V

A C++ GUI Framework

Bruce E. Wampler, Ph.D.

521 Springridge Dr.
Glenwood Springs, CO 81601

bruce@objectcentral.com
http://www.objectcentral.com

Version 1.18
March 15, 1998

The complete source for ***V*** (X Athena and Microsoft Windows versions) is available via anonymous ftp at `ftp.objectcentral.com`, or via the Object Central www home page (see above). This manual is also available as an html document on the web page.

Contents

Chapter 1

What is V?

V is a C++ Graphical User Interface Framework designed to provide an easy to use and program system for building GUI applications. The framework is small, elegant, and provides the tools required for building all but the most specialized applications.

The ***V*** framework has also been designed to be portable. Currently, versions for the X Windowing System (using a customized 3D Athena widget set), Microsoft Windows 3.1, and Microsoft WIN32 (Windows 95 and NT) are available. A version for OS/2 is in progress. A Macintosh port is under investigation. The ***V*** system is freely available for use by anyone under the terms of the GNU Library General Public License.

Why did I write ***V***, and why did I put it under the GNU license? I have been programming for over 20 years now, and building interactive applications for most of that time. During that time, I got tired of complicated, difficult to learn and use libraries for building interfaces, and wanted something easier.

I've also been successful in the software business, having founded two different software companies, Aspen Software and Reference Software International. I was the principle designer and author of the widely known and used grammar checker, Grammatik[1]. Basically, I see ***V*** as something of a public service; a way to give something back to the software industry that has been good to me. The concept of a portable GUI library is not original, but I think some of the design goals of ***V*** are significantly different than other similar libraries I've seen.

- The main design goal is for for ***V*** is ease of programming. I don't think that building the GUI part of an application should be the hardest part of the job as it is with most native GUI toolkits. ***V*** is small, easy to learn, easy to use, and provides the essentials of a good graphical user interface.

  I have some evidence that I have succeeded in this goal. ***V*** has been used for several semesters for large team projects in the software engineering class I taught at the University of New Mexico. While I get many questions from my students related to the projects they are doing, I got virtually no questions about using ***V*** itself. The small number of questions about ***V*** has been both startling and rewarding, and is good evidence that this design goal has been met. ***V*** has also been used successfully for a Junior level programming class. Previously, the high overhead of learning to write applications for X has prevented the students from writing small programs with interesting user interfaces. The simplicity of ***V*** has allowed them to do this for the first time.

[1] Grammatik is a trademark of Novell, Inc.

- ***V*** is designed to be portable. Over the years, I've programmed on a wide variety of interactive platforms. The main GUI platforms widely used today include the X Window System, Microsoft Windows (3.1, 95, and NT), OS/2, and the Macintosh. ***V*** has been designed to work on all those platforms, and present a look and feel that is consistent with native applications.

- ***V*** is not too big. It has less than 15 C++ classes that you will have to interact with. This is unlike many other frameworks that provide dozens and dozens of classes that you must learn and understand. The ***V*** framework only supports GUIs. It does not have templates, containers, and bunches of other C++ classes. If you need a good list class, use your favorite one from another class library. Use ***V*** for your interface.

- ***V*** has very good associated documentation. It is likely that part of the reason that ***V*** is easy to use is that it is accompanied by a better than average programming manual. I've tried to not only give a useful explanation of each ***V*** class and function, but to accompany each description with a short example that shows how to use the ***V*** feature in a useful way. There are also several examples provided with the ***V*** distribution to help you get started with a basic ***V*** application.

- ***V*** is an alternative for building *compiled* GUI applications. While interpretive solutions such as Tk/Tcl for building GUI applications are becoming popular, they don't allow fully compiled code on multiple platforms. As machines get faster and faster, I don't think the advantage of an edit/interpret cycle versus an edit/compile/run cycle is significant.

- The ***V*** library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or any later version.

  V is distributed in the hope that it will be useful, but *without any warranty*; without even the implied warranty of *merchantability* or *fitness for a particular purpose*. See the GNU Library General Public License for more details.

- The source code for ***V*** is of commercial quality, and I hope some of the easiest to read and understand code you will ever encounter (if you decide to look at the ***V*** source code).

There is, of course, a price to pay for the ease of programming with ***V***. The main constraint is that you are somewhat restricted to following ***V***'s (and thus my own) view of the world. The ***V*** model does not exactly conform to the native models of X, Windows, and the Mac, but it is a very good compromise. For the most part applications developed with ***V*** will in fact conform to the host look and feel, but may be lacking some of the bells and whistles of the most sophisticated commercial applications available for a given platform. For the vast majority of applications, this will not matter. You will end up with applications that look pretty good, and are likely to have a much cleaner and better interface than they would have otherwise.

If you are a C programmer, then the fact ***V*** is a C++ library might be a problem. While it is a fully object-oriented C++ framework, it can be used with C code if you know a bit about C++. Also, ***V*** does not allow you to do everything you could if you programmed in the native windowing library. You won't have every single conceivable control, and some controls are slightly restricted in how you can use them.

And finally, why the name ***V***? First of all, it is a simple name. It follows the tradition of *C* and *X*. It makes naming the classes easier. And, my son's name is Van, which starts with a V. So ***V*** it is.

Chapter 2

The V View of the World

Before getting into the details of ***V***, you might find it useful to read this overview of how the ***V*** view of the world was developed. If you are new at writing GUI applications, you should find this chapter especially useful.

2.1 A Generalized GUI Model

If you examine a large number of applications available on the major GUI platforms, you will find the interfaces typically have a great deal in common. While the visual details may differ, most applications have windows that show views of the data being manipulated, and use menus and dialogs for control interaction with the user. The user interacts with the program using a pointing device, usually a mouse, and the keyboard.

Windows

The *window* is usually the main interface object used by an application. The data being manipulated by the user (e.g., text, graphics, spreadsheet cells) is displayed in the window. Often, several windows may be open at the same time, each giving a different view of the data. There is usually a menu associated with the window for entering commands to manipulate data or to bring up dialogs.

The top level interface object used by ***V*** is a *Command Window*. Each command window consists of a *menu bar*, placed at the top of the window; a *canvas* area, used to draw text and graphics to display the data; and optional *command bars*, which include commands buttons and objects; and optional *status bars* to display state information.

Figure 2.1 represents, more or less, a typical top-level ***V*** window.

Dialog Boxes

Much control interaction with ***V*** applications takes place through one of two dialog objects: *modal* and *modeless* dialogs. In a modal dialog, interaction with any other window or dialog is locked out until the user interacts with it. In a modeless dialog, the user can continue to interact with other parts of the application while the dialog remains displayed. Modal dialogs will go away once the user enters a command. Modeless dialogs may or may not go away, depending on their purpose.

V supports a comprehensive set of controls for dialogs. These include command buttons, text labels, text input, list selection boxes, combo boxes, radio buttons, check boxes, spinners for value entry, sliders, and progress bars. These controls may be grouped into

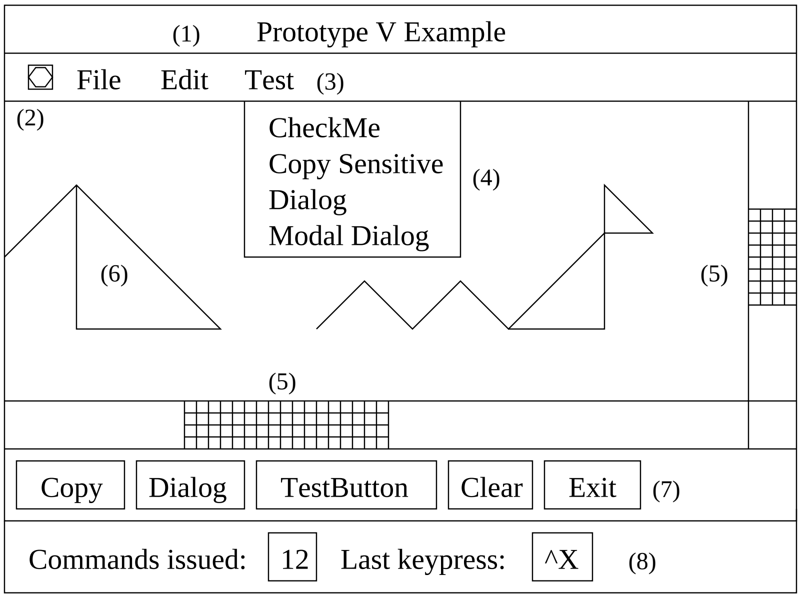


Figure 2.1: This top level consists of: (1) The title bar. (2) The close button. (3) The menu bar. (4) A pulldown menu. (5) Vertical and horizontal scroll bars. (6) The drawing canvas. (7) The command bar. (8) The status bar.

boxes. Layout of controls in a dialog is defined in the dialog definition list in the source code. Controls may be used in window command bars as well as dialogs.

Figure 2.2 represents, more or less, a typical ***V*** dialog.

Events

The structure of the code for user command processing in GUI applications is quite different from traditional C programs. The user input control model of traditional C programs is rather simple, usually using `printf` and `getc` or some variant for interaction. Logically, the program reaches a point where it needs input, and then waits for that input.

GUI applications deal with user input much differently. Interaction with an application from the user's viewpoint consists of a series of mouse movements and clicks, and text and command input through the keyboard. From the programmer's viewpoint, each of these is an event. The important thing about an event is that it can occur at any time, and the program cannot simply stop and wait for the event to happen.

Interaction with an application by the user can generate several different kinds of events. Consider mouse events. If the mouse is in the drawing area, each movement generates a *mouse movement* event. If the user clicks a mouse button, a *mouse button* event is generated. A keystroke from the keyboard will generate a *keyboard* event.

If the mouse pointer is in a dialog, or over a menu or command button, then movement events are not generated. Instead, button clicks generate *command* events.

Sometimes an application needs to track the passage of time. The application can call a system function that will generate a *timer* event at a given interval.

In a GUI environment, windows are usually not displayed alone. Often, other applications are running, each with its own windows. The host windowing system typically displays windows with various decorations that let the user manipulate the windows. Sometimes, these manipulations will generate events that require a response from the application code.

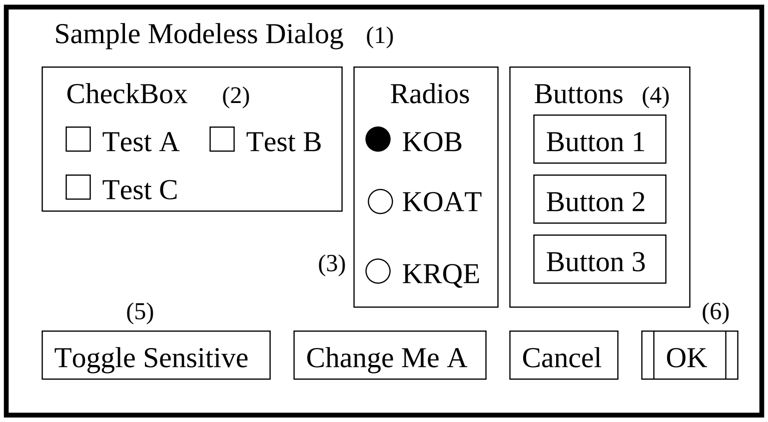


Figure 2.2: This dialog consists of: (1) Dialog label. (2) Three check boxes in a frame. (3) Three radio buttons in a frame. (4) Three buttons in a frame. (5) Four buttons, including (6) the default OK button.

For example, the user can use the mouse to change the size of a window causing a *resize* event. When multiple windows are displayed, some can be completely or partially covered by other windows. If the user moves a window so that a different part of the window is displayed, then an *expose* event is generated, which requires the program to redraw part of the canvas area.

All these events require a response from the application – to carry out the command, to draw something in the canvas area, or to redraw the canvas after a resize or expose event. Some events, however, are handled by the system, and not the application. This includes drawing menus and handling dialogs. For example, when a dialog is displayed, the system tracks mouse movements within the dialog, and handles redrawing the dialog for expose events. In general, the application is responsible for resize and expose events only for the canvas area.

All these events are asynchronous, and the application must be able to respond immediately to any of these events. Traditionally, handling events has been rather complicated. For each possible event, the program registers an *event handler* with the system. Then, the program runs in an *event loop*. The event loop receives an event, and then calls a function to dispatch the event to the proper event handler.

C++ makes dealing with events much easier. Each event can be considered a message, and the message is central to object-oriented programming. In ***V***, each object, such as a command window, has methods[1] that the system sends event messages to. For example, there is a `WindowCommand` method that responds to command events from the system. The application overrides the default ***V*** `WindowCommand` method to handle commands as needed by the application. All the details of the event loop and event handlers are hidden in the ***V*** implementation. If you have ever programmed with event handlers and loops, you will find the simplicity of overriding default methods incredibly easy in comparison!

[1] I use the general object-oriented term *method* to refer to what are called *member functions* in C++ terminology.

2.2 Easy to program

One of the main goals of the design of ***V*** was to make it easy to use to write real programs. Some of the factors that help ***V*** meet this goal are described in the following sections.

Hide the dirty details

One of the problems with using most native GUI libraries such as Xt or Windows is the huge amount of overhead and detail required to perform even the simplest tasks. You are typically coding at a very low level. While part of this complexity may be necessary to allow total flexibility in what you can do, the vast majority of applications just do not need total flexibility. ***V*** was designed to hide most of the details of the underlying GUI library. Things such as library initialization, specific window handles, and calls required to build low level controls are all hidden. Instead, you work at the much higher level of objects needed to build a typical GUI.

Easy to define GUI objects

It has always seemed to me that a GUI object such as a menu could most simply be thought of as a single object consisting of a list of items on that menu with their associated attributes. Rather than requiring a set of complicated calls to build that menu list, in ***V*** you can simply define a menu as a static C++ struct array – a list in other words. The same applies to dialogs. A dialog is a list of control objects with associated attributes, including how the controls are positioned in the dialog. This philosophy leads to very easy to maintain code. Menu and dialog lists are well defined in a single place in your code, and it is very easy to modify and change the list definitions. Actions for each menu or dialog command are defined in a single C++ method that responds to command events.

No resource editors

One data object used by most, but not all, native GUI libraries is what is usually called a resource file. A resource file is most often used to specify layout of dialogs and menus. One reason resource files are used is that specifying the layout of dialogs and menus directly in the code is often very difficult for the native libraries.

The combination of the way ***V*** lets you specify menus and dialogs, and the way C++ makes responding to event messages so easy has really removed the need for resource files. This in turn eliminates one of the more complicated aspects of portability across platforms.

Look and Feel

One of the limitations of ***V*** is that it has its own look and feel. While this may be a limitation, it is not necessarily bad. First, the look and feel is constrained so that applications will be portable across platforms and look like native applications on each platform. This means some things that are possible on one GUI platform, but not another, are not included in ***V***.

V also incorporates much of my own experience. I really like simplicity, and believe that just because you can do something, it is not necessarily a good idea to do so. Thus, for example, there are limitations on the number of menu items per menu, and how deeply you can nest pull down menus. These limits in fact help enforce good interface design.

2.3 Good Example of OO

While ***V*** has been designed to develop real and useful GUI applications, it also has been designed to be useful in a learning environment. Thus, ***V*** represents a good example of object-oriented design.

GUI systems are a natural for object orientation (OO). It is easy to understand the nature of each object – a window, a dialog, a command button, a menu bar, a canvas, and so on. Inheritance and aggregation of these objects is also very natural. Events are messages, and sending messages to methods is pure OO.

Since ***V*** is licensed under the terms of the GNU Library General Public License, the source code will always be available for study. It was written using the guidelines of Appendix B, and is very readable and easy to understand. Not only is the ***V*** source code a good example of OO programming, you may also find it interesting if you want to learn things about how the underlying GUI toolkits work. While good examples of freeware X source code are readily available, good examples of non-trivial Windows source code are nearly impossible to come by. I hope the ***V*** Windows source code will help fill this void.

2.4 The V Object Hierarchy

This manual contains several object hierarchy diagrams of the ***V*** framework, and of ***V*** applications. There are many graphical notations in varying degrees of widespread use, but I have found the Coad-Yourdon[2] notation one of the easiest to learn and simplest to use. The basic graphical elements of the notation are shown in Figure 2.3.

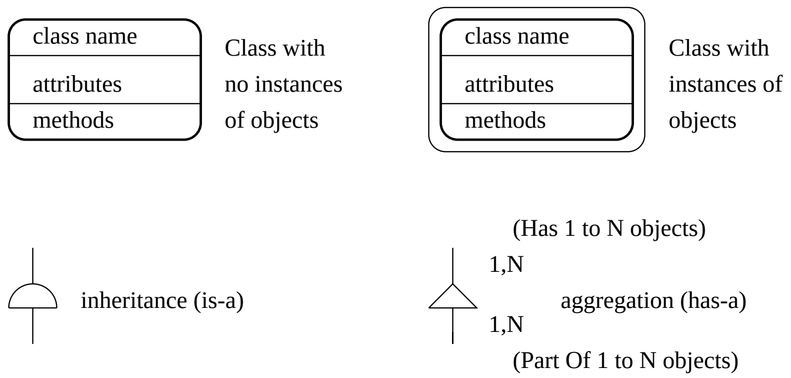


Figure 2.3: Coad-Yourdon OO Notation

An object is shown in a rectangular box. A single border indicates a generalized base class that will not have any instances, while a double border indicates that the named object can have instances. Generalization/specialization (inheritance, or is-a) relationships are shown with half circles. Whole/part (aggregation, or has-a) relationships are shown with triangles[3].

The "1,N" notation at the top of the aggregation triangle indicates that the object above can contain from 1 to N instances of the object below. The lower "1,N" indicates the lower

[2] Peter Coad and Edward Yourdon, *Object-Oriented Analysis*, 2nd ed. (Yourdon Press/Prentice Hall, 1990); and Edward Yourdon, *Object-Oriented System Design, An Integrated Approach* (Yourdon Press/Prentice Hall, 1994, ISBN 0-13-636325-3).

[3] Hint: It is sometimes hard to remember which shape is which. A triangle looks like a capital letter A as in Aggregation. The half circle shape is then inheritance.

object can be a part of 1 to N objects. The values can be changed to reflect reality. Thus, it is common to have "1,N" at the top, indicating that an object may contain many instances of the lower object, and just a "1" for the lower value, indication that an object is a part of exactly one of the upper objects.

When discussing a design at a high level, the attributes and methods boxes are often left blank. This leads to hierarchies such as the one for ***V*** in Figure 2.4 that shows the programming view of the ***V*** framework. In this case, there are no generalized base objects, and most of the relationships are whole/part.

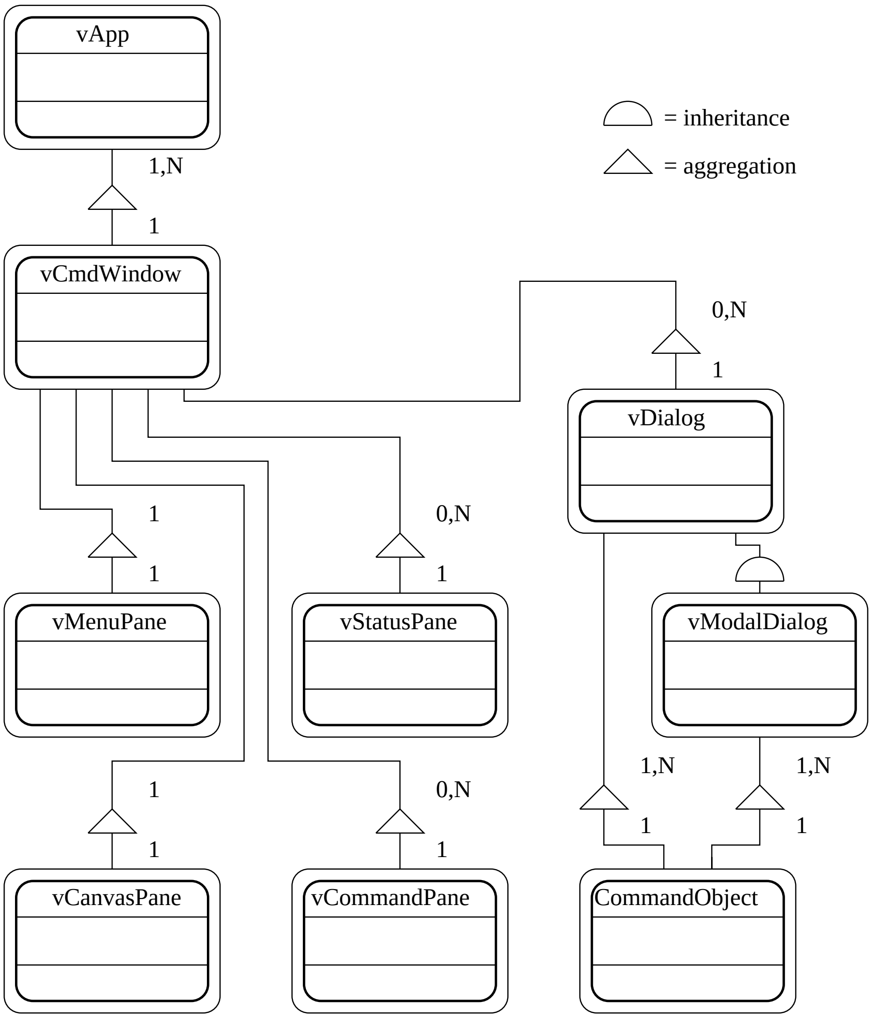


Figure 2.4: Programming View of V Classes

Figure 2.4 reveals some interesting things about ***V***'s look and feel. Note that a `vApp` class has 1 to N `vCmdWindows`, indicating that there will be at least one window. Each window, in turn, has exactly one menu and canvas, but zero to many command panes, status panes, and dialogs.

The version of the ***V*** hierarchy in the Appendix shows an implementation view of the hierarchy. Some of the classes that are never seen or used by the programmer are shown in that hierarchy.

Chapter 3

A V Tutorial

This chapter is intended to cover the elements that make up a ***V*** application. The first section covers the general organization of a "Standard ***V*** Application". Read this section to get an overview of a ***V*** application. Don't worry about the details yet – just the the main idea. Then read Section 3.4 and Appendix A, which has the source code of a small, complete ***V*** application, to get the details.

3.1 Getting Started with Your Own V Application

As with any new system, ***V*** has a learning curve before you can write applications of your own. ***V***'s learning curve is actually pretty short. The experience of the students using ***V*** has shown the best way to get started with ***V*** is to first read the first part of this reference manual, including this chapter. Then begin with an example ***V*** application.

The ***V*** application generator, `vgen`, included with the ***V*** distribution is the easiest way to begin building a ***V*** application. Run `vgen`, select the basic options you want to include in your application, select the directory to save the generated code in, and then generate the basic skeleton application. From the skeleton app, it is relatively easy to add your own functionality.

The tutorial application described in this chapter is also an excellent ***V*** example. Start by getting the example to compile. Then modify the code to add or remove features. Before long, you will have a good feel for ***V***, and be able to add all the features you need.

There are several other example programs provided with the ***V*** distribution. This tutorial is found in `~/v/tutor`. The VDraw program is found in `~/v/draw`. The program used to test all ***V*** functionality is found in `~/v/test`. It will have an example of how to use every ***V*** feature, although it is not as well structured as the other examples.

3.2 A Standard V Application

While the ***V*** framework is flexible enough to allow many different approaches to building an application, you should find it easier to base your applications on a model *Standard* **V** *Application*. The software organization described by a Standard ***V*** Application can support MVC (Model-View-Controller) object-oriented architecture paradigm.

Figure 3.1 shows the hierarchy of a standard ***V*** application. A standard ***V*** application consists of the parts described below. Each part consists of a pair of `.cpp` (or `.cxx`) and `.h` files (except the `makefile`).

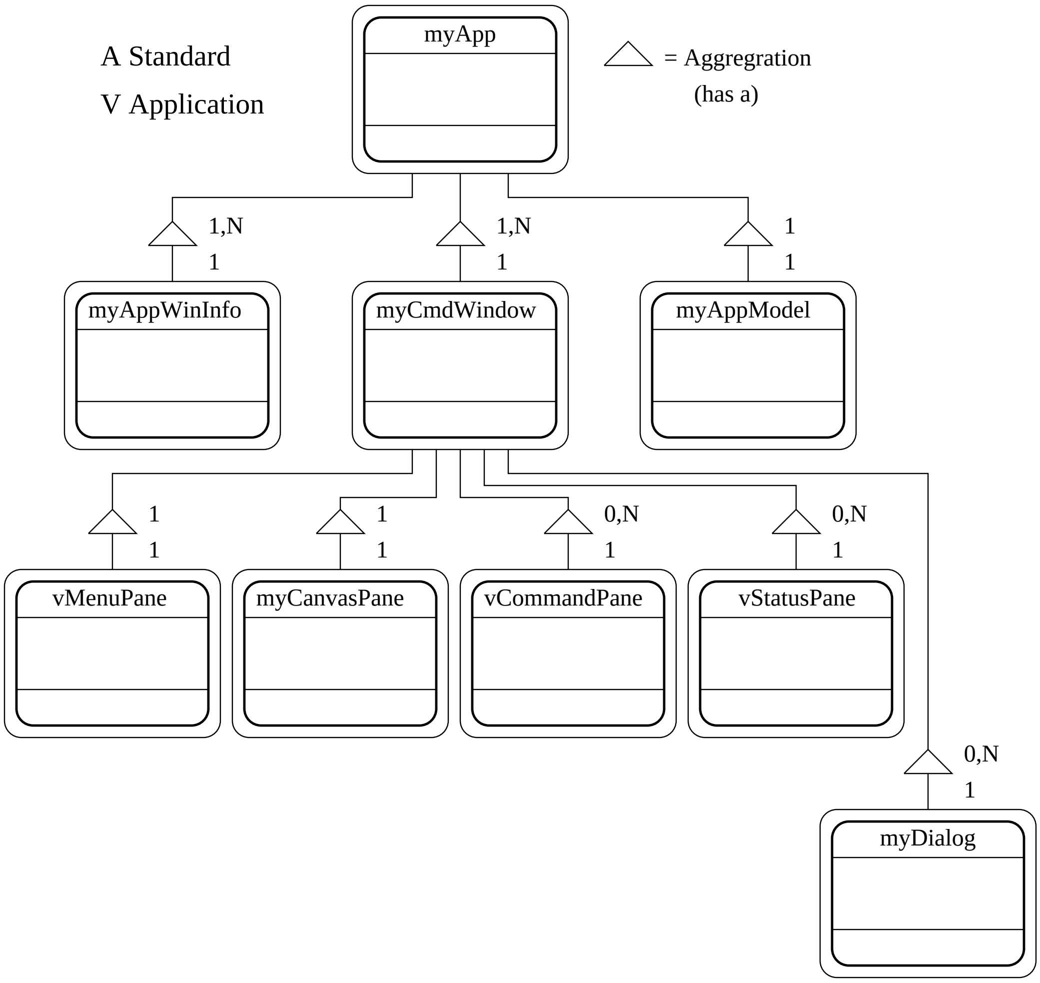


Figure 3.1: Standard V Application

The Application In many ways, the heart of a Standard ***V*** Application is the application class derived from the `vApp` class. By convention, this derived class is called `myApp` (but you can use a different name if you want.) There will always be exactly one instance of the `myApp` class. The `myApp` class acts as a coordinator between the windows that implement the user interface (the views) and the objects and algorithms that actually make up the application (the model). The `myApp` class will contain in a whole/part (or aggregation) relationship the windows defined by the application, as well as any classes needed to implement the application.

The `vApp` class has several utility methods that are usually used unmodified, plus several methods that are usually overridden by the `myApp` class. These are described in the section covering `vApp`. In addition, your `myApp` class will usually have several other programmer defined methods used to interface the command windows with the application model.

Windows and Canvases Each Standard ***V*** Application will have at least one top level window, and possible subwindows. These will usually be command windows derived from the `vCmdWindow` class. Your main derived class should be called `myCmdWindow`, and include a constructor that defines a menu bar, a canvas, and possible command and status bars. Of course, there will be a corresponding destructor. The `.cpp` file will contain the static definitions of the menu and any command and status bars. It will also override the `WindowCommand` method of `vCmdWindow` superclass. In your `WindowCommand` method, you will have a `switch` with a `case` for each menu item and button defined for the window.

Since a `vCmdWindow` contains different panes such as `vMenuPanes`, `vCanvasPanes`, `vCommandPanes`, and `vStatusPanes`, your top level command window object will usually define the appropriate pointers to each of these objects as required by the specific application. The `myCmdWindow` constructor will then have a `new` for each pane used.

Each instance of a window will be built using a call to the `vApp::NewAppWin` method. This allows the app object to track windows, and control interaction between the app model and the views represented by each window.

Some applications need to open subwindows. These windows may or may not use the same menu, command bar, and canvas as the top level window. If they do, then they can use the same static definitions used by the top level window. Subwindows may also have their own menu, button, and canvas definitions.

Canvases for Windows Since each window usually needs a canvas, you will usually derive a canvas object from the `vCanvasPane` class. At this point in the life of ***V***, there are only two possible kinds of canvas. The first is for graphics drawing, and is derived directly from the `vCanvasPane` class. The other kind is a text canvas derived from the `vTextCanvasPane` class. The derived class will define override methods required for the user to interact with the canvas.

Optional Dialogs Most applications will need dialogs – either modeless or modal. A Standard ***V*** dialog consists of a `.cpp` file with the static definition of the dialog commands, and the definitions of methods derived from the `vDialog` class. These will include a constructor and destructor, and a `DialogCommand` override with a `switch` with a `case` for each command defined for the dialog. Each `case` will have the code required to carry out useful work.

The top level window (or the subwindow that defines and uses the dialog) will create an instance of each dialog it needs (via `new`). The constructor for the dialog sets up the commands used for the dialog.

Typically, the top level window defines menu and button commands that result in the creation of a dialog. The top level window is thus usually responsible for invoking dialogs.

Optional Modal Dialogs Modal dialogs are almost identical to modeless dialogs. The main difference is how the dialog is invoked from the defining window.

Menu, Command and Status Bars By definition, the look and feel of a ***V*** application requires a menu bar on the command window. A ***V*** application also typically has a command bar and a status bar, but these are not required.

The Application Model Each application will need code to implement its data structures and algorithms. The design of the application model is beyond the scope of ***V***, but will usually be defined as a relatively independent hierarchy contained by the `myApp` object. Interaction between the application model and the various views represented by `myCmdWindows` can be coordinated with the `myAppWinInfo` class.

The Makefile Each ***V*** Standard Application should have an associated `makefile` that can be used to compile and link the application.

Please note that while ***V*** is object-oriented, the objects represent real screen windows. Thus, it makes no sense for most ***V*** objects to support copy constructors or object assignment. If you use one of these ***V*** objects in a way requiring a copy constructor or an assignment (fortunately, it is difficult to contrive such an example), the code will generate a run time error.

3.3 Special V Applications

Windows MDI/SDI

The basic standard ***V*** application includes a command window with a menu, a command bar, a canvas, and a status bar. While this model suits most applications, there are some special cases that ***V*** supports.

First, on Windows, ***V*** supports the standard Windows MDI model (Multiple Document Interface) by default. The MDI model consists of a parent window that can contain several children canvases, each with a different menu that changes in the main parent window when a child gets focus. In practice, the menus are usually the same for all children windows, and each window is used to hold a new document or data object. One of the main advantages of the MDI model is that each application has a main window to distinguish it from other Windows applications, and as many child windows as it needs to manipulate its data.

On X versions, there is no need for a special parent window. Each time you open a new command window, you get a new window on the X display.

The Windows MDI model forces some screen decorations that are not appropriate for all applications. Thus, ***V*** also supports the standard Windows SDI model. The SDI model allows only one canvas/command window combination. There is a parameter to the `vApp` constructor that tells ***V*** to use the SDI model. This parameter is not used on the X version.

Canvasless, menuless V Application

Sometimes an application needs just a command bar with no menu or canvas. By setting the `simSDI` parameter to 1, and supplying a width and height value to the `vApp` constructor, ***V*** allows this kind of simple interface. Instead of adding a menu and a canvas as is done for normal ***V*** apps, a menuless and canvasless app just defines a command pane for the command bar. The height and width are used to specify the height and width of the application, and require different values for Windows or X.

3.4 A Tutorial Example V Application

Now that you've read about the parts of a standard ***V*** application, it might be useful to go over a simple example of a ***V*** application. Appendix A contains the source code for a simple ***V*** application. The code is tutorial, and well commented. You can read the code directly and get a good understanding of what elements are required for a ***V*** application. This section will give a higher level overview of the code in Appendix A.

You should read this code, paying special attention to the comments. Most of the information you need to build a typical ***V*** application is explained in this code. This sample code is also available on line under the `~/v/tutor` directory. The source code of a slightly different standard ***V*** application is included the `~/v/examp` directory of the ***V*** distribution.

The previous section suggested using `myApp` for names. This tutorial uses a `t` prefix instead of `my`. You really can use whatever names you want. It will help to be consistent, however.

The code is broken down into five sections, corresponding to the main application, the main window, a simple canvas, and modal and modeless dialogs. The source code for each of these parts is included in Appendix A. The source code is extensively commented, and the comments contain much detail on how you should structure a ***V*** application, so please read them carefully. The following sections give a brief overview of each source file included in the tutorial example.

The Base Application Class

The file `tutapp.cpp` contains the overridden definitions of the classes `NewAppWin`, `Exit`, `CloseAppWin`, `AppCommand`, and `KeyIn` methods. These examples don't do much work, but are provided as a template for building complete applications.

The single definition of the application (`static tutApp tut_App("TutorApp")`, and the `AppMain` main program are also in this file. The initial window is created in `AppMain` by calling `NewAppWin`.

One thing that can be difficult to grasp when using a framework such as ***V*** is understanding where the program starts, and how you get things rolling. This happens in `tutapp.cpp`, so it is especially important to understand this piece of code. The essential thing to understand is that C++ will invoke the constructors of static objects before beginning execution of the program proper. Thus, you declare a static instance of the `vApp` object, and its constructor is used to initialize the native GUI library and get things going. Your program will *not* have a `main` function (see `AppMain` in the description of the `vApp` class for more details).

As with all files in the tutorial, each has a `.cpp` source file, and its associated `.h` header file. All ***V*** code has been written using the coding guidelines given in Appendix B. This includes the order of the declarations included in header files.

The Command Window

The file `tcmdwin.cpp` contains the code for the main command window. Of particular interest are the definitions of the main menu, command pane, and status pane. These panes are defined and added to the window in the constructor.

There is also code to demonstrate handling keyboard and window command events in the `KeyIn` and `WindowCommand` methods. There is also a simple example of using the `vFileSelect` utility class, as well as invoking modeless and modal dialogs.

The Canvas

The file `tcanvas.cpp` contains the code for the canvas. This is a really simple canvas example which supports drawing a few lines. This class handles redrawing after expose events very simply, but demonstrates what must be done in general.

A Modeless Dialog

The file `tdialog.cpp` contains the code for a modeless dialog. There are just a few example buttons, check boxes, and radio buttons. The `DialogCommand` methods demonstrates how to handle commands from a dialog.

A Modal Dialog

The file `tmodal.cpp` contains the code for a modal dialog. The definition of a modal dialog is nearly identical to a modeless dialog. The main difference is how they are invoked, which is shown in the `tcmdwin.cpp` code.

The Makefile

The file `makefile` contains a sample Unix-style make file. This version is for Gnu make, which has features different than some other flavors of make. It should still serve as a decent example.

Chapter 4

The Application

This chapter covers the top level classes used to build an application.

The classes covered in this chapter include:

vApp The base class for applications.

vAppWinInfo A utility class to interface views to models.

4.1 vApp

The base class for building applications.

Synopsis

Header: `<v/vapp.h>`

Class name: vApp

Contains: vCmdWindow, vAppWinInfo

Description

The `vApp` class serves as the base class for building applications. There must be exactly one instance of an object derived from the `vApp` class. The base class contains and hides the code for interacting with the host windowing system, and serves to simplify using the windowing system.

You will usually derive a class based on `vApp` that will serve as the main control center of the application, as well as containing the window objects needed for the user interface. The single instance of the application class is defined in the body of the derived application class code.

The `vApp` class has several utility methods of general usefulness, as well as several methods that are normally overridden to provide the control interface from the application to the command windows. The derived class will also usually have other methods used to interface with the application.

In order to simplify the control interface between the application and the windows, the `vAppWinInfo` class has been provided. The application can extend that class to keep track of relevant information for each window. When the `NewAppWin` method is used to create a window, it will create an appropriate instance of a `vAppWinInfo` object, and return a pointer to the new object. The base `vApp` then provides the method `getAppWinInfo` to retrieve the information associated with a given window.

Constructor

vApp(char* appName)

vApp(char* appName, simSDI = 0, int fh = 0, int fw = 0)

appName Default name for the application. This name will be used by default when names are not provided for windows. The name also appears on the "main window" for some platforms, including Microsoft Windows, but not X. The constructor also initializes some internal state information. There must be exactly one instance of the `vApp` object, and will usually represent your derived `myApp` object. See the code below with `AppMain` for an example of creating the single app instance.

simSDI This *optional* parameter is used to specify that ***V*** should start as a Windows SDI application if it is set to 1. This parameter has no effect for the X version.

fw, fh These are used to specify the size of a menuless and canvasless ***V*** application, and are optional.

Methods to Override

void AppCommand(vWindow* win, ItemVal val)

Any window commands not processed by the `vWindow` object are passed to `AppCommand`. You can override this method to handle any commands not processed in windows.

int AppMain(int argc, char** argv)

This is a global function (not a class member!) that is called once by the system at start up time with the standard command line arguments `argc` and `argv`. You provide this function in your code.

Your program will not have a C `main` function. The main reason for this is portability. While you would usually have a `main` in a Unix based program, MS-Windows does not use `main`, but rather `PASCAL WinMain`. By handling how the program gets started and providing the `AppMain` mechanism, ***V*** allows you to ignore the differences. You will still have all the capability to access the command line arguments and do whatever else you would do in `main` without having to know about getting the host windowing system up and running.

The windowing system will have been initialized before `AppMain` is called. You can process the command line arguments, and perform other required initializations. The top level command window should also created in `AppMain` by calling `NewAppWin`.

Before `AppMain` is called, the single instance of your derived `vApp` object must also be constructed, usually by instantiating a static instance with a statement such as `static myApp* MyApp = new myApp("ProtoApp")`. As part of the construction of the `myApp` object, the global pointer `vApp* theApp` is also pointed to the single instance of the `vApp` or derived `myApp` object. You can then use `theApp` anywhere in your code to access methods provided by the `vApp` class.

Your `AppMain` should return a 0 if it was successful. A nonzero return value will cause the ***V*** system to terminate with an exit code corresponding to the value you returned.

Example

```
// EVERY V application needs the equivalent of the following line
```

```
  static myApp* MyApp = new myApp("ProtoApp");  // Construct the app.

//==========================>>> AppMain <<<===========================
  int AppMain(int argc, char** argv)
  {
    // Use AppMain to perform special app initialization, and
    // to create the main window.  This example assumes that
    // NewAppWin knows how to create the proper window.

    (void) theApp->NewAppWin(0, "Prototype V Example", 350, 100, 0);
    return 0;
  }
```

void CloseAppWin(vWindow* win)

This is the normal way to close a window. Your derived `CloseAppWin` should first handle all housekeeping details, such as saving the contents of a file, and then call the default `vApp::CloseAppWin` method. The default method calls the window's `CloseWin` method and removes the window.

The `CloseAppWin` method is also called when the user clicks the close button of the window. This close button will correspond to the standard close window button depending on the native windowing system. On the X Window System, each window will have a close button in the menu bar. On Windows, this corresponds to a double click on the upper left box of the title bar, or the "X" box in Windows 95.

void Exit(void)

This is the normal way to exit from a standard ***V*** application. The overridden method can perform any special processing (e.g., asking "Are you sure?") required. The default `Exit` will call `CloseAppWin` for each window created with `NewAppWin`, and then exit from the windowing system.

void KeyIn(vWindow* win, vKey key, unsigned int shift)

Any input key events not handled by the `vWindow` object are passed to `VApp::KeyIn`. See `KeyIn` in the `vWindow` section for details of using keys.

vWindow* NewAppWin(vWindow* win, char* name, int w, int h, vAppWinInfo* winInfo)

The purpose of the `NewAppWin` method is to create a new instance of a window. Most likely, you will override `NewAppWin` with your own version, but you still *must* call the base `vApp::NewAppWin` method *after* your derived method has completed its initializations.

The default behavior of the base `NewAppWin` class is to set the window title to `name`, and the width `w` and height `h`. Note that the height and width are of the *canvas*, and not necessarily the whole app window. If you don't add a canvas to the command window, the results are not specified. Usually, your derived `NewAppWin` will create an instance of your derived `vCmdWindow` class, and you will pass its pointer in the `win` parameter. If the the `win` parameter is null, then a standard `vCmdWindow` will be created automatically, although that window won't be particularly useful to anyone.

Your `NewAppWin` class may also create an instance of your derived `vAppWinInfo` class. You would pass its pointer to the `winInfo` parameter. If you pass a null, then the base `NewAppWin` method also creates an instance of the standard `vAppWinInfo` class.

The real work done by the base `NewAppWin` is to register the instance of the window with the internal ***V*** run time system. This is why you must call the base `NewAppWin` method.

`NewAppWin` returns a pointer to the object just created. Your derived code can return the value returned by the base `vApp::NewAppWin`, or the pointer it created itself.

Example

The following shows a minimal example of deriving a `NewAppWin` method.

```
vWindow* myApp::NewAppWin(vWindow* win, char* name, int w, int h,
  vAppWinInfo* winInfo)
  {
    // Create and register a window. Usually this derived method
    // knows about the windows that need to be created, but
    // it is also possible to create the window instance outside.

    vWindow* thisWin = win;
    vAppWinInfo* theWinInfo = winInfo;

    if (!thisWin)   // Normal case: we will create the new window
        thisWin = new myCmdWindow(myname, w, h);  // create window

    // Now the application would do whatever it needed to create
    // a new view -- opening a file, tracking information, etc.
    // This information can be kept in the vAppWinInfo object.

    if (!theWinInfo)               // Create if not supplied
        vAppWinInfo* theWinInfo = new myAppWinInfo(name);

    // Now carry out the default actions
    return vApp::NewAppWin(thisWin, name, w, h, theWinInfo);
  }
```

Utility Methods

int DefaultHeight()

Returns a default window canvas height value in pixels corresponding to 24 lines of text in the default font.

int DefaultWidth()

Returns a default window canvas width value in pixels corresponding to 80 columns of text in the default font.

vFont GetDefaultFont(void)

This method returns a `vFont` object representing the default system font. It is a convenience method, and probably not overly useful to application programs.

vFont GetVVersion(int& major, int& minor)

Returns the current major and minor version of ***V***.

int IsRunning()

This method returns true if the windowing system is active and running. A false return means the program was started from a nonwindowing environment.

int ScreenHeight()

Returns the overall height of the physical display screen in pixels. Note that this value may or may not be overly useful. On X, the `vCommandWindows` are drawn on the full display. On the Windows MDI version, the command windows all fall inside the MDI frame, and thus knowing the size of the whole screen is less useful.

int ScreenWidth()

Returns the overall width of the physical display screen in pixels. See `ScreenHeight`.

void SendWindowCommandAll(ItemVal id, int val, CmdType ctype)

This method can be used to send a message to the `WindowCommand` method of *ALL* currently active windows. This method is most useful for sending messages to windows from modeless dialogs. While messages to the `WindowCommand` method usually originate with the system in response to menu picks or command object selection, it can be useful to send the messages directly under program control. The `vDraw` sample program contains a good example of using `SendWindowCommandAll` (and `SetValueAll`) in `vdrwdlg.cpp`. There is no way to send a message to a specific window. The message is sent to all active windows.

void SetAppTitle(char* title)

This method is used to set the title of the main application window. This currently only applies to the Microsoft Windows MDI version of ***V***. It is a no-op for the X version. It is still important that you choose a good title for your main window, and set it either with this method, or by providing a good name to the `vApp` initializer.

void SetValueAll(ItemVal itemId, int Val, ItemSetType what)

This method is similar to `vWindow::SetValue`, except that the control with the given `itemId` in *ALL* currently active windows is set. This is useful to keep control values in different windows in sync. The only difference between `vApp :: SetValueAll` and `vWindow :: SetValueAll` is that the `vApp` version can be easily called from dialogs as well as windows.

void SetStringAll(ItemVal itemId, char* title)

This method is similar to `vWindow::SetString`, except that the string with the given `itemId` in *ALL* currently active windows is set. This is useful to keep control strings in different windows in sync. The only difference between the `vApp::SetStringAll` version and the `vWindow::SetStringAll` version is that the `vApp` version can be easily called from dialogs as well as windows.

int ShowList()

This method is intended mostly for debugging, and will print on `stderr` the list of currently registered windows.

vAppWinInfo *getAppWinInfo(vWindow* win)

This method provides an easy way to retrieve the `vAppWinInfo` (or more typically, a derived class) object that is associated with a window. By convention, when a window is first created, it and its associated `vAppWinInfo` object are tracked by `NewAppWin`. When a user action in a window causes a method in `vApp` to be invoked, the `this` of that window is usually sent to the `vApp` method. You then use that `vWindow` pointer to call `getAppWinInfo` to get a pointer to the associated `vAppWinInfo` object. It will be up to you to determine what information that object has, and how to use it.

Tasking

Some applications may have extensive computation requirements. In traditional programming environments, this is usually no problem. However, for GUI based applications, the code cannot simply perform extensive computation in response to some command event (such as a "Begin Computation" menu command). GUIs make a basic assumption that the application will process events relatively quickly. While computation is in process, the application will not receive additional events, and may appear to hang if the computation is too long.

V provides two different approaches to handling compute bound applications. The most straight forward approach is to have the computation periodically call the ***V*** method `vApp::CheckEvents`. `CheckEvents` will process events, and pass the messages to the appropriate ***V*** method. This method may be the most appropriate for applications such as simulations.

The second technique is to have the ***V*** system call a work procedure periodically to allow some computation to be performed. This technique may be most appropriate for applications that have short computations that should be performed even if the user is not entering commands or interacting with the application. The technique is supported by the `WorkSlice` method.

CheckEvents()

Most ***V*** applications will *not* need this utility. However, it is possible for some compute bound applications to lock out system response to the events needed to update the screen. If you notice that your application stops responding to input, or fails to consistently update items in your window, then place calls to `vApp::CheckEvents()` in your code somewhere. You may have to experiment how often you need to call it. It does have some overhead, so you don't want it to slow down your app. But it does need to get called enough so the system can keep up with the screen updates. This function needs no parameters, and returns no value.

EnableWorkSlice(long slice)

For applications that need computations to be performed continuously or periodically, even while the user is not interacting with the program, ***V*** provides `EnableWorkSlice` and `WorkSlice`. After `EnableWorkSlice` has been called, ***V*** will call the app's `WorkSlice` method every `slice` milliseconds. The `WorkSlice` method of every open `vCommandWindow` will also be called. Calling `EnableWorkSlice` with a zero value will stop the calls to the `WorkSlice` methods.

V uses a standard ***V*** `vTimer` object to implement this behavior. Thus, all of the information about actual time intervals and limits on the number of timers discussed in the `vTimer` description apply to `EnableWorkSlice` and `WorkSlice`.

WorkSlice()

When a `EnableWorkSlice` has been called with a positive value, ***V*** calls `vApp::WorkSlice` at approximately the specified interval (or more likely, the overridden method in your app), as well as the `vWindow::WorkSlice` method of each open `vCommandWindow`. Your application can override the appropriate `WorkSlice` method to perform short, periodic computations. Theses computations should be shorter than the time interval specified for `EnableWorkSlice`. This may be difficult to ensure since different processors will work at different speeds. One simple way to be sure you don't get multiple calls to the `WorkSlice` method is to set a static variable on entry to the code. Note that `vCommandWindow` also has a `WorkSlice` method. The `WorkSlice` for the `vApp` is called first, followed by a call to each open `vCommandWindow` sequentially in no specific order.

See Also

vWindow, vAppWinInfo

4.2 vAppWinInfo

A utility class to interface views to models.

Synopsis

Header: `<v/vawinfo.h>`

Class name: vAppWinInfo

Description

This class is intended to be used as a base class for deriving your own `myAppWinInfo` class. The purpose of such a class is to serve as a controller data base for the MVC architecture. Typically, you would keep an `AppWinInfo` record for each view (open window) of the file or whatever represented by the model. For example, if your application were a text file viewer, the `AppWinInfo` record could track the file name, current line number, and viewing mode for each view of a file.

V makes using a `AppWinInfo` object easier by automatically tracking it when you create each new window with `NewAppWin`. You can then easily retrieve the `AppWinInfo` object associated with each window by using the `vApp::getAppWinInfo` method.

Constructor

vAppWinInfo(char* infoName = "", void* ptr = 0)

You can provide two values for the `vAppWinInfo` constructor. The first is a pointer to a character string which you can use to store some name meaningful to you application. The second is a `void *` pointer, and can be used to point to anything you want. The constructor makes a copy of the name string, but just copies the void pointer and does not copy the object pointed to.

Utility Methods

virtual char* infoName()

Returns a pointer to the name supplied to the constructor.

virtual void* getPtr()

Returns the value of the pointer name supplied to the constructor.

See Also

vApp

Chapter 5

Command Windows

This chapter covers the classes used to build windows and command windows.

The classes covered in this chapter include:

vCmdWindow A class to show a window with various command panes.

vCommandPane Used to define commands on a command bar.

vMenu Used to define pull down menus.

vPane Base class for various window panes.

vStatus Used to define label fields on a status bar.

vWindow A class to show a window on the display.

5.1 vCmdWindow

A class to show a window with various command panes.

Synopsis

Header: `<v/vcmdwin.h>`

Class name: vCmdWindow

Hierarchy: vBaseWindow → vWindow → vCmdWindow

Contains: vDialog, vPane

Description

The `vCmdWindow` class is derived from the `vWindow` class. This class is intended as a class that serves as a main control window containing various `vPane` objects such as menu bars, canvases, and command bars. The main difference between the `vCmdWindow` class and the `vWindow` class is how they are treated by the host windowing system. You will normally derive your windows from the `vCmdWindow` class.

Constructor

vCmdWindow(char* title)

vCmdWindow(char* title, int h, int w)

These construct a `vCmdWindow` with a title and a size specified in pixels. You can use `theApp->DefaultHeight()` and `theApp->DefaultWidth()` in the call to the constructor to create a "standard" size window. Note that the height and width are of the canvas area, and not the entire window.

Inherited Methods

See the section `vWindow` for details of the following methods.

virtual void KeyIn(vKey key, unsigned int shift)

virtual void MenuCommand(ItemVal itemId)

virtual void WindowCommand(ItemVal Id, ItemVal Val, CmdType Type)

virtual void AddPane(vPane* pane)

virtual void GetPosition(int& left, int& top, int& width, int& height)

virtual int GetValue(ItemVal itemId)

virtual void RaiseWindow(void)

virtual void ShowPane(vPane* wpane, int OnOrOff)

virtual void SetValue(ItemVal itemId, int Val, ItemSetType what)

virtual void SetString(ItemVal itemId, char* title)

virtual void SetTitle(char* title)

virtual void CloseWin()

See Also

vWindow

5.2 vCommandPane

Used to define commands on a command bar.

Synopsis

Header: `<v/vcmdpane.h>`

Class name: vCommandPane

Used by: vCmdWindow

Description

A command pane is a horizontal bar in a command window that holds `CommandObjects`. You can use any of the `CommandObjects`, although they all might not make sense to use on a command bar (a List, for example, is a bit large for the visual paradigm, but it would work). The layout is left to right, so you don't need to fill in the RightOf and Below fields. You can include Frames in a command bar, and commands contained in that frame do use the RightOf and Below attributes.

You define the commands on a command bar using a `CommandObject` array. You first create the command pane with `myCmdPane = new vCommandPane(CommandBar)`, and then add it to the window with `AddPane(myCmdPane)`.

You then handle the command objects in a command bar pretty much like the same way as in a dialog. The main difference is that you use the `vWindow` versions of `SetValue` and `WindowCommand` instead of the corresponding methods of the `vDialog` class. Other than the left to right ordering, things are pretty much the same.

Example

The discussion of `CommandObject` and `vDialog` contains several examples of defining command objects.

See the section `vPane` for a general description of panes.

See Also

vWindow, vStatus, CommandObject, vDialog, vPane

5.3 vMenu

Used to define pull down menus.

Synopsis

Header: `<v/v_menu.h>`

Type name: vMenu

Description

The `vMenu` structure is used to define pulldown menus, which includes the top level item on the menu bar, as well as the items contained in the pulldown menu. These are passed to the constructor of a `vCmdWindow` type object.

See the section `vPane` for a general description of panes.

Definition

```
typedef struct vMenu
  {
    char* label;        // The label on the menu
    ItemVal menuId;     // A User assigned unique id
    unsigned
     sensitive : 1,     // If item is sensitive or not
     checked : 1;       // If item is checked or not (*)
    char* keyLabel;     // Label for an accelerator key (*)
    vKey accel;         // Value of accelerator key
    vMenu* SubMenu;     // Ptr to a submenu
```

```
  unsigned int kShift; // Shift state of accelerator
} MenuItem;
```

Note that the items marked with an asterisk (`checked` and `keyLabel`) are not used when defining the top level menu bar items.

Structure Members

char* label The label on the menu. See the description of the `vWindow` class for information on setting the label of menu bar items.

For some platforms (Windows, but not Athena X), you can add a & to indicate a shortcut for the command. For example, specifying a label `&File` allows Windows users to pull down the `File` menu by pressing `Alt-F`, and specifying a submenu label as `&New` allows the user to use `Alt-N` to select the `New` command. The Athena version of ***V*** strips the &, so you can (and probably should) denote shortcuts for menu items even in Athena versions.

ItemVal MenuId A user assigned unique id. This id is passed to the `MenuCommand` (or `WindowCommand`) method when a menu item is selected. If a menu item with a submenu is selected, ***V*** will not return the id, but will cause the submenu to be displayed.

It will be common practice to use the same id for menu items and command objects defined on a command bar, and the same id value would then be passed to `WindowCommand` for either the menu selection or the equivalent button selection. Similarly, using the id to set the item's sensitivity will change both the menu and the button.

The values you use for your id in menus and controls should be limited to being less than 30,000. The predefined ***V*** values are all above 30,000, and are reserved. *There is no enforcement of this policy.* It is up to you to pick reasonable values.

If you want a separator line on a pulldown menu, you must use the predefined value `M_Line` for the `MenuId`.

int sensitive Controls if item is initially sensitive or not. Insensitive items are displayed grayed out. The predefined symbols `notSens` and `isSens` can be used to define the `MenuItem`. Note that ***V*** uses the static definition of the `MenuItem` to store the current sensitive state, and all menus (or windows) sharing the same static definition will have the same sensitive state. See the description of the `vWindow` class for information on setting the sensitivity of menu bar items.

int checked The user can put a check mark in front of the label of a menu item. This convention is often used to show a given setting is in effect. Like the sensitive member, this statically tracks the checked state. The predefined values `isChk` and `notChk` can be used to specify this value. This value is not used when defining the top level menu bar, and you can use the predefined value `notUsed` for that case. See the description of the `vWindow` class for information on setting checked state of menu items.

char* keyLabel Label for an accelerator key. The predefined symbol `noKeyLbl` can be used to indicate there is no `keyLabel`. This value is not used when defining the top level menu bar, and you can use the predefined value `notUsed` accelerator key.

vKey accel This is the value of the keystroke that is the accelerator for this menu item. When the user presses this key, the `vWindow::MenuCommand` method will be called just as though the user had used the mouse to select the menu item. This value may be used in combination with the `kShift` and `keyLabel` parameters. See the explanation of `vWindow::KeyIn` for a complete explanation of key codes.

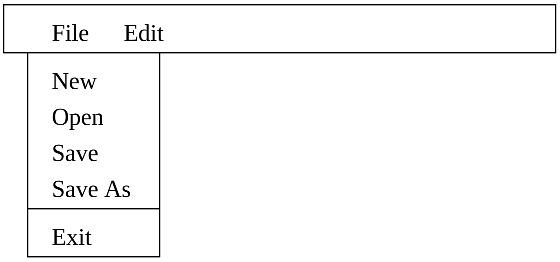
Note that the Windows version really doesn't support `Alt` key codes. The Windows system intercepts Alt keys and tries to interpret them as menu accelerators. Unfortunately, there is no simple way to override this behavior, so Alt keys are essentially unsupported on Windows. Using functions keys with combinations of Shift and Control is supported, as are regular control keys.

MenuItem* SubMenu Pointer to another `MenuItem` definition of a submenu. ***V*** will cause submenus to be shown automatically when selected. The predefined symbol `noSub` can be used to indicate there is no submenu.

unsigned int kShift This is the shift value to be used with the `accel` key definition. To use `Ctrl-D` as the accelerator key, you would specify the value for Control-D (easily specified as `'D'-'@'`) for `accel`, and leave `kShift` set to zero. If you use a Ctrl code, you must specify both the control code, and the `VKM_Ctrl` shift code. Note that this value is at the end of the `vMenu` structure because of it was forgotten in early implementations of ***V***. By placing it at the end, earlier versions of ***V*** code are compatible with no changes to the source. Sigh, I didn't get this one right.

Example

This example defines a menu bar with the items *File* and *Edit*. The `MenuBar` definition would be passed to the constructor of the appropriate `vCmdWindow` derived object.



Only the File submenu is shown here, and is an example of the menu as it might be included in a standard File menu. Note that this example menu includes items that can all be specified by using standard predefined values (see *Predefined ItemVals*). It also includes an optionally defined `Debug` item. A definition like this might be used for the `FileMenu` in the `Menu` example. Note that & is used to denote shortcuts for menu items.

```
static vMenu FileMenu[] =
  {
    {"&New", M_New, isSens,notChk,noKeyLbl,noKey,noSub},
    {"&Open", M_Open, isSens,notChk,noKeyLbl, noKey, noSub},
    {"&Save", M_Save, isSens,notChk,noKeyLbl,noKey,noSub},
    {"Save &As", M_SaveAs, isSens,notChk,noKeyLbl,noKey,noSub},
#ifdef vDEBUG
    {"-", M_Line, notSens,notChk,noKeyLbl,noKey,noSub},
    {"&Debug", M_SetDebug,isSens,notChk,noKeyLbl,noKey,noSub},
#endif
    {"-", M_Line, notSens,notChk,noKeyLbl,noKey,noSub},
    {"E&xit", M_Exit, isSens,notChk,noKeyLbl,noKey,noSub},
    {0}
  };

static vMenu EditMenu[] = {...};  // Define Edit pulldown

// Define menu bar, which includes the File and Edit pulldown
```

```
static vMenu MenuBar[] =
  {
    {"&File",M_File,isSens,notUsed,notUsed,noKey,&FileMenu[0]},
    {"&Edit",M_Edit,isSens,notUsed,notUsed,noKey,&EditMenu[0]},
    {0,0}                        // end of menubar
  };

  ...

  vMenuPane myMenuPane = new vMenuPane(MenuBar);  // construct pane
  AddPane(myMenuPane);
```

See Also

vWindow, vPane

5.4 vPane

The `vPane` class serves as a base class for various pane objects contained by the `vWindow` class. There are no methods or services provided by the `vPane` class that you need to use directly, but the class is used extensively by ***V*** internally.

There are four types of panes used by ***V*** in a `vCmdWindow`, including menu panes, canvas panes, command panes and status panes. To add a pane to a window, you will first define the contents of the pane (menu, commands, status info) using static arrays, then construct an instance of the pane with `new vWhateverPane`. Then you add the instance to the window using `AddPane`.

Note that the canvas panes are described in the *Drawing* chapter. The commands used with a command pane are described in the *Dialogs* chapter, while menus and status bars are covered in `vMenu` and `vStatus` in this chapter.

Canvas Pane

Header: `<v/vcanvas.h>`

Class name: vCanvasPane

Constructor: `userCanvasPane()`

Command Pane

Header: `<v/vcmdpane.h>`

Class name: vCommandPane

Constructor: `vCommandPane(CommandObject* cmdbar)`

Menu Pane

Header: `<v/vmenu.h>`

Class name: vMenuPane

Constructor: `vMenuPane(vMenu* menubar)`

Status Pane

Header: `<v/vstatusp.h>`

Class name: vStatusPane

Constructor: `vStatusPane(vStatus* sbar)`

See Also

CommandObject, vCanvasPane, vCmdWindow, vCommandPane, vMenu, vStatus

5.5 vStatus

Used to define label fields on a status bar.

Synopsis

Header: `<v/v_defs.h>`

Type name: vStatus

Used by: vWindow

Description

The `vStatus` structure is used to define the top level status bar included on a `vCmdWindow`, and the labels it contains. The `vStatus` array is usually passed to the `vStatusPane` constructor. See the section `vPane` for a general description of panes.

Definition

```
typedef struct vStatus      // for status bars
  {
    char* label;            // text label
    ItemVal statId;         // id
    CmdAttribute attrs;     // attributes - CA_NoBorder
    unsigned sensitive : 1; // if button is sensitive or not
    int width;              // to specify width (0 for default)
  } vButton;
```

Structure Members

char* label Text of label field. See the description of the `vWindow` class for information on changing the text of a label.

ItemVal id Id for the label. Use this value when changing value with `SetString` or `SetValue`.

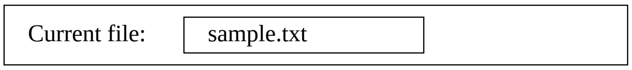
CmdAttribute attrs The current implementation only uses the `CA_NoBorder` attribute. If `CA_NoBorder` is supplied, the label will be drawn on the command bar without a border or box around it. Not supplying `CA_NoBorder` (e.g., `CA_None`) will result in a label with a border or box around it. In general, unbordered labels don't change, and bordered labels are used to show changing status.

int sensitive If label is sensitive or not. Use predefined symbols `isSens` and `notSens` to specify the initial state. On some implementations, the label will be grayed if it is insensitive. The sensitivity can be changed using `vWindow::SetValue` as described in the section `vWindow`.

int width This can be used to specify a fixed width for a label. Normally, the label will be sized to fit the length of the text. If you provide a non-zero width, then the label field will remain constant size.

Example

This shows a sample status bar with two fields. It is added to a `vCmdWindow` using `AddPane`. The value of the file name would be changed by calling `SetString(m_curFile, filename)` somewhere in your program.




```
static vStatus sbar[] =
  {
    {"Current file:", m_curMsg,CA_NoBorder,isSens,0},
    {" ", m_curFile,CA_None,isSens,100},
    {0,0,0,0,0}
  };
  ...
  vStatusPane myStatusPane = new vStatusPane(sbar); // construct
  AddPane(myStatusPane);
```

See Also

vWindow, vPane

5.6 vWindow

A class to show a window on the display.

Synopsis

Header: `<v/vwindow.h>`

Class name: vWindow

Hierarchy: vBaseWindow → vWindow

Contains: vDialog, vPane

Description

The `vWindow` class is an aggregate class that usually has associated `vPane` objects – window panes, in other words. There several kinds of panes, including menu panes, command bar panes, status panes, and drawing canvas panes. As you would expect, classes derived from `vWindow` also include panes.

The `vWindow` class will probably never be used by your application - it serves primarily as a superclass for the `vCmdWindow` class. This class may be more useful in future versions of

V, but for now it is not really useful by itself. You will typically derive your own class from `vCmdWindow`, and override several of the methods defined by `vWindow` and `vCmdWindow`.

Menus and commands in the panes send messages to the `WindowCommand` and `MenuCommand` methods when the user clicks on a command or menu item contained in the window. The application program can also change attributes of the various menu items and commands associated with a window. Canvas panes are designed to handle their own interaction with the user (mouse events, etc.).

Constructor

vWindow()

vWindow(char* title)

vWindow(char* title, int h, int w)

vWindow(char* title, int h, int, WindowType wintype)

title Title to place in title bar.

h,w The height and width of the window.

wintype CMDWINDOW or WINDOW type for window.

The constructor for `vWindow` is normally called with a name, size, and possibly a window type. The name will be displayed in the window's title bar by default. The size is the initial size of the window's *canvas* work area in pixels. The type may be `CMDWINDOW` or `WINDOW`. The constructor for `vCmdWindow` invokes the proper `vWindow` constructor.

Methods to Override

virtual void KeyIn(vKey key, unsigned int shift)

`KeyIn` is invoked when a key is pressed while a window has focus. The `key` value is the `vKey` value of the key pressed, and `shift` indicates the shift state of the key.

Handling the keystroke is not necessarily trivial. Regular ASCII characters in the range from a Space (0x40) up to a tilde (~) are passed to `KeyIn` directly, and shift will be 0, even for upper case letters. The current version of ***V*** does not have explicit support for international characters, so values between 0x80 and 0xFF are undefined, and correspond to whatever might be the local convention for the character set. (This will be one thing for X and another for Windows - but you can count on the values for each platform. Thus, you can use non-English characters on each platform, even though they won't be the same values on X and Windows. I would like a portable solution for this. If any non-English users of ***V*** have any ideas about this problem, I'd like to hear. The choice seems to be between the standard MS-DOS code page solution and the ANSI character set used on X platforms. I'm not ready to support multibyte characters for some time yet.) Values between 0xFF00 and 0xFFFF correspond to the various function keys and keypad keys found on a typical keyboard. The standard set by IBM PCs has determined what function keys are supported by ***V***. The file `<v/vkeys.h>` has the definitions for the key codes supported.

Besides getting a keycode for the non-ASCII keys, `KeyIn` also gives a shift code corresponding to the Control, Shift, and Alt modifier keys. (These are defined as `VKM_Ctrl`, `VKM_Shift`, and `VKM_Alt`.) Pressing the F4 key would return the code for F4 (vk_F4), while the keystroke Alt-F4 will return the code for the F4 key, and the shift code set to `VKM_Alt`.

More than one bit of the shift code can be set – the shift values are really bit values. Control keys from the normal character set (Ctrl-A, etc.) are passed as their true control code, but *not* the `VKM_Ctrl` shift set.

In addition, you also need to check for the `VKM_Alt` modifier applied to regular Ascii keys. The keystroke Alt-K will be mapped to a *lower case* Ascii 'k' with the `VKM_Alt` bit set in `shift`. The top row keys (1,2, etc.) can also be pressed with the `VKM_Ctrl` bit set in `shift`, and your program will need to deal with these. It will quite often be the case that your program simply ignores many of these values.

`KeyIn` will also return a value when only a modifier key is pressed. For example, pressing the Alt key returns a key value of `vk_Alt`. A macro defined in `<v/vkeys.h>` called `vk_IsModifer(x)` can be used to determine if a key `x` is a modifier. Your program can usually ignore modifier keys.

If you have defined any keystroke combinations to be accelerators for menu commands, your program will never see those keystrokes in `KeyIn`. Instead, they are intercepted by the system and mapped to the appropriate command to pass to the `MenuCommand` method.

Note that the keystrokes are not displayed by the system. It is up to your program to handle keystrokes and to do something useful with them.

You should call `vWindow::KeyIn` from your derived method with any keystrokes you don't handle. The `vWindow::KeyIn` method passes these unhandled keystrokes up to the `vApp::KeyIn` method. Thus, you will have the choice of either handling keystrokes in the window or in the app class.

virtual void MenuCommand(ItemVal itemId)

`MenuCommand` is called when a menu command is selected. This virtual function allows menu commands to be distinguished from other commands in a window, although it is not usually necessary to do so. The default method simply passes the menu command along to the `WindowCommand` method, so you don't need to override this method if you don't distinguish between menu and command events.

virtual void WindowCommand(ItemVal Id, ItemVal Val, CmdType Type)

This method is invoked when a user activates a command object in a command pane. The `Id` of the command object is passed in in the `Id` field, and the value and type (e.g., C_Button or C_CheckBox) of the command are passed in in the `Val` and `Type` parameters. Note that command objects in a command pane are really no different than the command objects in a dialog. Most of the discussion for handling these commands is covered in the sections on dialogs. See `vCommandPane` and `vDialog::DialogCommand` for more details about the values passed to `WindowCommand`.

`WindowCommand` is also called by the default `MenuCommand` in response to menu picks. The `Id` is the id of the item that generated the call.

The default behavior of `WindowCommand` is to call the `AppCommand` method. However, you will almost always override the default `WindowCommand` method.

virtual void WorkSlice()

See `vApp::WorkSlice` for a description of this method.

Utility Methods

virtual void AddPane(vPane* pane)

This method is used to add the pane `pane` to a window. Panes will be displayed in the order they are added. You can add exactly one menu pane, plus canvas, command, and status panes. You typically first create a given pane (e.g., `myPane = new XPane(PaneDef))`, and then add the pane to the window with `AddPane(myPane)`.

void GetPosition(int& left, int& top, int& width, int& height)

Returns the position and size of `this` window. These values reflect the actual position and size on the screen of the window. On X, this is the whole `vCommandWindow` frame. On the Windows MDI version, it is the size and position of just the drawing canvas and its scroll bars. The intent of this method is to allow you to find out where the active window is so you can move a window, or position a dialog so that it doesn't cover a window. It is most useful when used in conjunction with `SetDialogPosition`.

virtual int GetValue(ItemVal itemId)

This method is used to retrieve the value of a menu or command object in a menu or command pane. The `itemId` is the id of the item as defined in the menu or command bar definition. For menu items, this will return the menu checked state. For other command objects, the value returned will be appropriate as described in the *Dialog Commands* section.

virtual void RaiseWindow(void)

This method will raise the window to top of all windows on the display. Raising a window is often a result of mouse actions of the user, but this method allows a buried window to be moved to the top under program control. You will need to track which window instance you want raised, possibly through the `vAppWinInfo` object.

virtual void SetValue(ItemVal itemId, int Val, ItemSetType what)

This method is used to change the state of command window items. The item with `itemId` is set to `Val` using the `ItemSetType` parameter to control what is set. Not all command items can use all types of settings. See `vWindow::GetValue` and `vDialog::SetValue` for a more complete description.

If a menu item and a command item in the same window share the same id, they will both be set to the same value (this usually applies to sensitivity). Only the controls in the window that sent this message are changed.

virtual void SetValueAll(ItemVal itemId, int Val, ItemSetType what)

This method is similar to `SetValue`, except that the control with the given `itemId` in *ALL* currently active windows is set. This is useful to keep control values in different windows in sync.

virtual void SetPosition(int left, int top)

Moves `this` window to the location `left` and `top`. This function is of limited usefulness. `SetDialogPosition` is more useful.

virtual void SetString(ItemVal itemId, char* title)

This can be used to change the label on a command bar button, status bar label, or menu item. The item identified by `itemId` will have its label changed to `title`.

virtual void SetStringAll(ItemVal itemId, char* title)

This method is similar to `SetString`, except that the string with the given `itemId` in *ALL* currently active windows is set. This is useful to keep control strings in different windows in sync.

virtual void SetTitle(char* title)

Set the name of the window shown on its title bar to `title`.

virtual void ShowPane(vPane* wpane, int OnOrOff)

You can show or hide a command, status, or canvas pane with this method. The pane must first be defined, created, and added to the command window (which will show the pane). You can then hide the pane later by calling this method with the pointer to the pane and `OnOrOff` set to 0. A 1 will show the pane. Note that in some environments (e.g., X), the window may show up again in a different position in the window. For example, if you had a command bar above a status bar, and then hide the command bar, it will be placed under the status bar when you show it again. This is a "feature" of X.

virtual void ShowWindow(void)

You *must* call `ShowWindow()` after you have added all the panes to the window. You usually call `ShowWindow()` in the constructor to your `vCmdWindow` class after you have created all the panes and have used `AddPane` to add them to the window.

Other Methods

virtual void CloseWin()

This method is called by the `vApp::CloseAppWin` method as part of closing down a window. The default `vWindow::CloseWin()` method's behavior is to take care of some critical housekeeping chores. You will normally never override this method. However, it is remotely conceivable that there will be an occasion you need to do something really low level after a window has been destroyed by the host GUI environment. In that case, your method *must* call the immediate superclass `vWindow::CloseWin()`, and then do whatever it has to do. Normally, you handle such details in your class's `CloseAppWin` method.

See Also

vCmdWindow

Chapter 6

Dialogs

This chapter covers the classes used to build dialogs, and the various kinds of command objects that can be included in a dialog.

The classes and command objects covered in this chapter include:

CmdAttribute A type describing attributes of various command objects.

CommandObject Main type used to define commands to dialogs and command panes.

Commands Command items used in building dialogs.

vIcon Used to define ***V*** icons.

vDialog Class to build a modeless dialog.

vModalDialog Used to show modal dialogs.

6.1 CmdAttribute

A type describing attributes of various command objects.

Synopsis

Header: `<v/v_defs.h>`

Type name: CmdAttribute

Description

These attributes are used when defining command items. They are used to modify default behavior. These attributes are bit values, and some can be combined with an *OR* operation. Note that not all attributes can be used with all commands.

Attributes

CA_DefaultButton Used with a `C_Button` to indicate that this button will be the default button. The user can activate the default button by pressing the Enter key as well as using the mouse. It will most often be associated with the OK button.

CA_Hidden Sometimes you may find it useful to have a command object that is not displayed at first. By using the `CA_Hidden` attribute, the command object will not be displayed. The space it will require in the dialog or dialog pane will still be allocated, but the command will not be displayed. You can then unhide (or hide) the command using the `SetValue` method: `SetValue(CmdID, TrueOrFalse, Hidden)`.

CA_Horizontal Command will have horizontal orientation. This attribute is used with Sliders and Progress Bars.

CA_Large The object should be larger than usual. It can be used with Lists, Progress Bars, Sliders, Text Ins, and Value Boxes.

CA_MainMsg Used with a `C_Label` to indicate that its string will be replaced with the message supplied to the `ShowDialog` method.

CA_NoBorder Used for frames and status bar labels, `CA_NoBorder` specifies that the object is to be displayed with no border.

CA_NoLabel Used for progress bars to suppress display of the value label.

CA_NoNotify Used for combo boxes and lists. When specified, the program will not be notified for each selection of a combo box item or a list item. When specified, the program is notified only when the combo box button is pressed, and must then use `GetValue` to retrieve the item selected in the combo box list. For lists, you will need another command button in the dialog to indicate list selection is done.

CA_NoSpace Used for frames, this attribute causes the command objects within the frame to be spaced together as tightly as possible. Normally, command objects have a space of several pixels between them when laid out in a dialog. The `CA_NoSpace` attribute is especially useful for producing a tightly spaced set of command buttons.

CA_None No special attributes. Used as a symbolic filler when defining items, and is really zero.

CA_Percent Used with progress bars to add a % to the value label.

CA_Size The `size` element of the `CommandObject` is being used to specify a size for the control. This is used with buttons, spin controls, and lists.

CA_Small The object should be smaller than usual. It can be used with Progress Bars and Text Ins. On Progress Bars, `CA_Small` means that the text value box will not be shown.

CA_Text Used for Spinners to specify that a text list of possible values has been supplied.

CA_Vertical Command will have vertical orientation. This attribute is used with Sliders and Progress Bars.

6.2 CommandObject

Used to define commands to dialogs and command panes.

Synopsis

Header: `<v/v_defs.h>`

Type name: CommandObject

Part of: vDialog, vCommandPane

Description

This structure is used to define command items in dialogs and command panes. You will define a static array of `CommandObject` items. This array is then passed to the `AddDialogCmds` method of a dialog class such as `vDialog` or `vModalDialog`, or the constructor of a `vCommandPane` object, or more typically, a class derived from one of those.

Definition

```
typedef struct CommandObject
  {
    CmdType cmdType;    // what kind of item is this
    ItemVal cmdId;      // unique id for the item
    ItemVal retVal;     // initial value of object
    char* title;        // string
    void* itemList;     // used when cmd needs a list
    CmdAttribute attrs; // list of attributes
    int Sensitive;      // if item is sensitive or not
    ItemVal cFrame;     // Frame used for an item
    ItemVal cRightOf;   // Item placed left of this id
    ItemVal cBelow;     // Item placed below this one
    int size;           // Used for size information
    char* tip;          // ToolTip string
  } CommandObject;
```

Structure Members

CmdType cmdType This value determines what kind of command item this is. The types of commands are explained in the section *Commands*.

ItemVal cmdId This unique id for the command defined by the programmer. Each command item belonging to a dialog should have a unique id, and it is advisable to use some scheme to be sure the ids are unique. The ***V*** system does not do anything to check for duplicate ids, and the behavior is undefined for duplicate ids. The id for a command is passed to the `DialogCommand` method of the dialog, as well as being used for calls to the various `SetX` and `GetX` methods. There are many predefined values that can be used for ids as described in the chapter *Standard V Values*.

The values you use for your id in menus and controls should be limited to being less than 30,000. The predefined ***V*** values are all above 30,000, and are reserved. *There is no enforcement of this policy.* It is up to you to pick reasonable values.

The type `ItemVal` exists for historical reasons, and is equivalent to an int, and will remain so. Thus, the easiest way to assign and maintain unique ids for your controls is to use a C++ `enum`. As many as possible examples in this manual will use `enums`, but examples using the old style `const` codeItemVal declarations may continue to exist. There is more discussion of assigning ids in the following example.

int retVal The use of this value depends on the type of command. For buttons, for example, this value will be passed (along with the `cmdId`) to the `DialogCommand` method. The `retVal` is also used for the initial on/off state of check boxes and radio buttons. For some commands, `retVal` is unused. Note that the static storage provided in the declaration is *not* used to hold the value internally. You should use `GetValue` to retrieve the current value of a command object.

char* title This is used for the label or text string used for command items.

void* itemList This is used to pass values to commands that need lists or strings. The ListCmd is an example. Note the `void *` to allow arbitrary lists.

CmdAttribute attrs Some command items use attributes to describe their behavior. These attributes are summarized in Section 6.1, *CmdAttribute*.

int Sensitive This is used to determine if an item is sensitive or not. Note that the static storage provided in the declaration is used by the ***V*** system to track the value, and should be changed by the `SetValue` method rather than directly. Thus dialogs sharing the same static declaration will all have the same value. This is usually desired behavior.

ItemVal cFrame Command items may be placed within a frame. If this value is 0 (or better, the symbol `NoFrame`), the command will be placed in the main dialog area. If a value is supplied, then the command will be placed within the frame with the id `cFrame`.

ItemVal cRightOf, ItemVal cBelow These are used to describe the placement of a command within a dialog. Ids of other commands in the same dialog are used to determine placement. The current command will be placed to the right of the command `cRightOf`, and below the command `cBelow`. The commands left and above don't necessarily have to be adjacent. By careful use of these values, you can design very attractive dialogs. You can control the width of command objects by padding the label with blanks. Thus, for example, you can design a dialog with all buttons the same size.

You can also use the `CA_Hidden` attribute to selectively hide command objects that occupy the same location in the dialog. Thus, you might have a button labeled `Hide` right of and below the same command object as another button labeled `UnHide`. By giving one of the two buttons the `CA_Hidden` attribute, only one will be displayed. Then you can use `SetValue` at runtime to switch which button is displayed in the same location. The bigger of the two command objects will control the spacing.

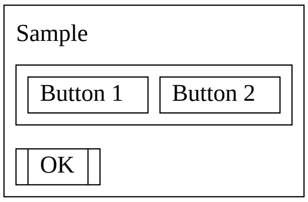
int size The size parameter can be used for some command objects to specify size. For example, for labeled Button commands, the `size` specifies the minimum width in pixels of the button. It is also used in various other command objects as needed. A value of zero for `size` always means use the default size. Thus, you can take advantage of how C++ handles declarations and write `CommandObject` declarations that leave off the `size` values, which default to zero. Many of the examples in this reference do not specify these values.

char* tip The tip parameter is used to specify an optional ToolTip string for use with a command object. If you provide a string here, that string will be automatically displayed after the user holds the mouse over that control. The exact delay before the tip is shown, and the format of the tip box is somewhat platform dependent, and all platforms might not support tool tips. (Currently, only OS/2 does not support tips.) Note that if you use a tip, you must be sure to include a value (usually 0) for the size parameter!

Example

The following example defines a simple dialog with a message label on the top row, a check box on the second row, two buttons in a horizontally organized frame on the third row, and an OK button on the bottom row. The ids in this example are defined using an `enum`. Remember that your ids must be less than 30,000, and using 0 is not a good idea. Thus, the `enum` in this example gives the ids values from 101 to 106. An alternative used in ***V*** code prior to release 1.13 was to provide `const` declarations to define meaningful symbolic values for the ids. Many examples of this type of id declaration will likely persist.

It also helps to use a consistent naming convention for ids. The quick reference appendix lists suggested prefixes for each control type under the `CmdType` section. For example, use an id of the form `btnXXX` for buttons. Predefined ids follow the form `M_XXX`.




```
enum {lbl1 = 101, frm1, btn1, btn2}
static CommandObject Sample[] =
  {
    {C_Label, lbl1, 0,"Sample",NoList,CA_MainMsg,isSens,NoFrame,0,0},
    {C_Frame, frm1, 0, "", NoList,CA_None,isSens,NoFrame,0,lbl1},
    {C_Button, btn1, 0, "Button 1", NoList, CA_None, isSens,frm1,0,0,0,
         "Tip for Button 1"},
    {C_Button, btn2, 0, "Button 2", NoList, CA_None, isSens,frm1,btn1,0,0,
         "Tip for Button 2"},
    {C_Button, M_OK, M_OK, " OK ", NoList, CA_DefaultButton,
        isSens, NoFrame,0,frm1},
    {C_EndOfList,0,0,0,0,CA_None,0,0,0}
  };
```

See Also

vWindow, Predefined ItemVals, CmdAttribute, Commands

6.3 Commands

This section describes how each of the command objects available in ***V*** is used to build dialogs.

Synopsis

Header: `<v/v_defs.h>`

Type name: CmdType

Description

V provides several different kinds of command items that are used in dialogs. The kind of command is specified in the `cmdType` field of the `CommandObject` structure when defining a dialog. This section describes current dialog commands available with ***V***. They will be constructed by ***V*** to conform to the conventions of the host windowing system. Each command is named by the value used to define it in the `CommandObject` structure.

Commands

C_Blank

A Blank can help you control the layout of your dialogs. The Blank object will occupy the space it would take if it were a `C_Label`, but nothing will be displayed. This is especially useful for leaving space between other command objects, and getting nice layouts with RightOfs and Belows. You control the size of the Blank by providing a string with an appropriate number of blanks for the `title` field.

C_BoxedLabel

Special Label Information

This command object is just like a `C_Label`, but drawn with a surrounding box. See `C_Label`.

C_Button

Save

A Button is one of the primary command input items used in dialog boxes. When the user clicks on a Button, the values set in the `cmdId` and `retVal` fields are passed to the `DialogCommand` method. In practice, the `retVal` field is not really used for buttons – the `cmdId` field is used in the `switch` statement of the `DialogCommand` method.

A button is defined in a `CommandObject` array. This is a typical definition:

```
{C_Button, btnId, 0,"Save",NoList,CA_None,isSens,NoFrame,0,0}
```

The `retVal` field can be used to hold any value you wish. For example, the predefined color button frame (see `vColor`) uses the `cmdId` field to identify each color button, and uses the `retVal` field to hold the index into the standard ***V*** color array. If you don't need to use the `retVal`, a safe convention is to a 0 for the `retVal`. You can put any label you wish in the `title` field.

If you provide the attribute `CA_DefaultButton` to the `CmdAttribute` field, then this button will be considered the default button for the dialog. The default button will be visually different than other buttons (usually a different border), and pressing the Return key is the same as clicking on the button.

The size of the button in pixels can be controlled by using the `CommandObject` element `size`. By specifying the attribute `CA_Size` and providing a value for the `size` element, you can control the size of the button. Note the that the `size` element is the last one of a `CommandObject`, and can left out of a declaration, which results in the compiler generating a zero value.

You can change the label of a button with: `SetString(btnId, "New Label")`. You can change the sensitivity of a button with `SetValue(btnID, OnOrOff, Sensitive)`.

C_CheckBox

☒ Show Details

A CheckBox is usually used to set some option on or off. A CheckBox command item consists of a check box and an associated label. When the user clicks on the check box, the `DialogCommand` method is invoked with the `Id` set to the `cmdId` and the `Val` set to the current state of the CheckBox. The system takes care of checking and unchecking the displayed check box – the user code tracks the logical state of the check box.

A CheckBox is defined in a `CommandObject` array. This is a typical definition:

```
{C_CheckBox, chkId, 1,"Show Details",NoList,CA_None,isSens,NoFrame,0,0}
```

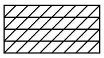
The `retVal` is used to indicate the initial state of the check box. You should use the `GetValue` method to get the current state of a check box. You can also track the state dynamically in the `DialogCommand` method. You can put any label you wish in the `title` field.

You can change the label of a check box with: `SetString(chkId, "New Label")`. You can change the sensitivity of a check box with `SetValue(chkID, OnOrOff,Sensitive)`. You can change the checked state with `SetValue(chkID, OnOrOff, Checked)`.

If the user clicks the Cancel button and your code calls the default `DialogCommand` method, ***V*** will automatically reset any check boxes back to their original state, and call the `DialogCommand` method an additional time with the original value if the state has changed. Thus, your code can track the state of check boxes as the user checks them, yet rely on the behavior of the Cancel button to reset changed check boxes to the original state.

The source code for the ***V*** `vDebugDialog` class provides a good example of using check boxes (at least for the X version). It is found in `v/src/vdebug.cxx`.

C_ColorButton



A color command button. This works exactly the same as a `C_Button` except that the button may be colored. You use `C_ColorButton` for the `cmdType` field, and provide a pointer to a `vColor` structure in the `itemList` field using a `(void*)` cast. The label is optional.

The `retVal` field of a color button is not used. You can generate a square color button of a specified size by specifying an empty label (`""`) *and* a `size` value greater than 0. When you specify the `size` field, the color button will be a colored square `size` pixels per side. When used within a `CA_NoSpace` frame, this feature would allow you to build a palette of small, tightly spaced color buttons. In fact, ***V*** provides a couple of such palettes in `v/vcb2x4.h` and `v/vcb2x8.h`. These include files, as well as the other details of the `vColor` class are described in the section `vColor` in the *Drawing* chapter.

There are two ways to change to color of a button. The most direct way is to change each of the RGB values in three successive calls to `SetValue` using `Red`, `Green`, and finally `Blue` as the `ItemSetType` to change the RGB values. The call with `Blue` causes the color to be updated. I know this isn't the most elegant way to do this, but it fits with the `SetValue` model.

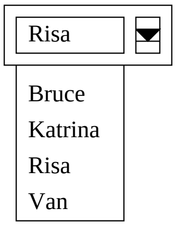
An alternate way is to change the value of the original `vColor` used to define the initial color of the control, and then call `SetValue` with the `ChangeColor` set type.

This is a short example of defining a red button, and then changing it.

```
    static vColor btncolor(255,0,0};  // define red
    ...

    // part of a CommandObject definition
    {C_ColorButton, cbt1, 0, "", (void*)&btncolor,
        CA_None, isSens, NoFrame, 0, btnXXX},

    ...
    // Code to change the color by some arbitrary values
    btncolor.Set(btncolor.r()+127, btncolor.g()+63, btncolor.b()+31);
#ifdef ByColor    // by vColor after changing btncolor
```

```
    SetValue(cbt1,0,btncolor);
#else           // by individual colors
    SetValue(cbt1,(ItemVal)btncolor.r(),Red);
    SetValue(cbt1,(ItemVal)btncolor.g(),Green);
    // This final call with Blue causes color to update in dialog
    SetValue(cbt1,(ItemVal)btncolor.b(),Blue);
#endif
    ...
```

C_ComboBox



A combo box is a drop-down list. It normally appears as box with text accompanied by some kind of down arrow button. You pass a list of alternative text values in the `itemList` field of the `CommandObject` structure. You also must set the `retVal` field to the index (starting at 0) of the item in the list that is the default value for the combo box text title.

If the user clicks the arrow, a list pops up with a set of alternative text values for the combo box label. If the user picks one of the alternatives, the popup closes and the new value fills the text part of the combo box. ***V*** supports up to 32 items in the combo box list. You need to use a `C_List` if you need more than 32 items.

With default attributes, a combo box will send a message to `DialogCommand` whenever a user picks a selection from the combo box dialog. This can be useful for monitoring the item selected. If you define the combo box with the attribute `CA_NoNotify`, the dialog in not notified on each pick. You can use `GetValue` to retrieve the index of the item shown in the combo box text field.

You can preselect the value by using `SetValue`. You can change the contents of the combo list by using `vDialog::SetValue` with either `ChangeList` or `ChangeListPtr`. See `vDialog::SetValue` for more details.

Example

The following is a simple example of using a combo box in a modal dialog. This example does not process items as they are clicked, and does not show code that would likely be in an overridden `DialogCommand` method. The code interface to a list and a combo box is very similar – the interaction with the user is different. This example will initially fill the combo box label with the text of `comboList[2]`.

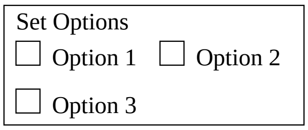
```
enum { cbxId = 300 };
char* comboList[] =
  {
    "First 0",   // The first item in the list
     ...
    "Item N",    // The last item in the list
    0            // 0 terminates the list
  };
  ...
CommandObject ComboList[] =
  {
    {C_ComboBox, cbxId, 2, "A Combo Box", (void*)comboList,
       CA_NoNotify,isSens,NoFrame,0,0},
```

```
    {C_Button, M_OK, M_OK, " OK ", NoList,
       CA_DefaultButton, isSens, NoFrame, 0, ListId},
    {C_EndOfList,0,0,0,0,CA_None,0,0,0}
  };
    ...
    vModalDialog cd(this);    // create list dialog
    int cid, cval;
    ...
    cd.AddDialogCmds(comboList);   // Add commands to dialog
    cid = ld.ShowModalDialog("",cval);  // Wait for OK
    cval = ld.GetValue(cbxId);  // Retrieve the item selected
```

C_EndOfList

This is not really a command, but is used to denote end of the command list when defining a `CommandObject` structure.

C_Frame



The frame is a line around a related group of dialog command items. The dialog window itself can be considered to be the outermost frame. Just as the placement of commands within the dialog can be controlled with the `cRightOf` and `cBelow` fields, the placement of controls within the frame use the same fields. You then specify the id of the frame with the `cFrame` field, and then relative position within that frame.

The `title` field of a frame is not used.

You may supply the `CA_NoBorder` attribute to any frame, which will cause the frame to be drawn without a border. This can be used as a layout tool, and is especially useful to force buttons to line up in vertical columns.

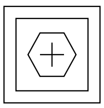
See the section *CommandObject* for an example of defining a frame.

C_Icon

A display only icon. This works exactly the same as a `C_Label` except that an icon is displayed instead of text. You use `C_Icon` for the `cmdType` field, and provide a pointer to the `vIcon` object in the `itemList` field using a `(void*)` cast. You should also provide a meaningful label for the `title` field since some versions of ***V*** may not support icons.

You can't dynamically change the icon.

C_IconButton



A command button Icon. This works exactly the same as a `C_Button` except that an icon is displayed for the button instead of text. You use `C_IconButton` for the `cmdType` field, and provide a pointer to the `vIcon` object in the `itemList` field using a `(void*)` cast.

You should also provide a meaningful label for the `title` field since some versions of ***V*** may not support icons.

You can't dynamically change the icon. The button will be sized to fit the icon. Note that the `v/icons` directory contains quite a few icons suitable for using on command bars.

C_Label

C_ColorLabel

Select Options

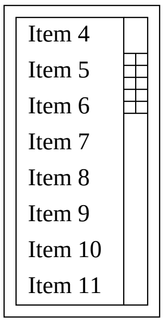
This places a label in a dialog. A label is defined in a `CommandObject` array. This is a typical definition:

```
{C_Label, lblId,0,"Select Options",NoList,CA_None,isSens,NoFrame,0,0, 0,0}
```

While the value of a label can be changed with `SetString(lblId, "New Label")`, they are usually static items. If the label is defined with the `CA_MainMsg` attribute, then that label position will be used to fill the the message provided to the `ShowDialog` method.

A `C_ColorLabel` is a label that uses the List parameter of the `CommandObject` array to specify a `vColor`. You can specify the color and change the color in the same fashion as described in the `C_ColorButton` command.

C_List



A list is a scrollable window of text items. The list can be made up of any number of items, but only a limited number are displayed in the list scroll box. The default will show eight items at a time. The number of rows can be controlled as explained later.

The user uses the scroll bar to show various parts of the list. Normally, when the user clicks on a list item, the `DialogCommand` is invoked with the id of the List command in the `Id` parameter, and the index into the list of the item selected in the `Val` parameter. This value may be less than zero, which means the user has unselected an item, and your code should properly handle this situation. This only means the user has selected the given item, but not that the selection is final. There usually must be a command Button such as OK to indicate final selection of the list item.

If the List is defined with the attribute `CA_NoNotify`, `DialogCommand` is not called with each pick. You must then use `GetValue` to get which item in the list was selected.

It is possible to preselect a given list item with the `SetValue` method. Use the `GetValue` to retrieve the selected item's index after the OK button is selected. A value less than zero means no item was selected.

The number of rows displayed can be controlled by using the `CommandObject` element `size`. By specifying the attribute `CA_Size` and providing a value for the `size` element, you can specify how many rows to show. If you don't specify a size, 8 rows will be displayed.

V will support between 1 and 32 rows. Note the that the `size` element is the last one of a `CommandObject`, and can left out of a declaration, which results in the compiler generating a zero value, giving the default 8 rows.

The width in pixels (approximately) of the list can be controlled by specifying the `CA_ListWidth` attribute and providing a value to the `retVal` parameter, which is otherwise unused for a list object. This implementation isn't perfect - you may have to play with the interaction between the width you specify, and the font used in a list control.

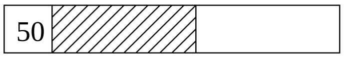
Change the contents of the list with `vDialog::SetValue` using either `ChangeList` or `ChangeListPtr`. See `vDialog::SetValue` for more details.

Example

The following is a simple example of using a list box in a modal dialog. This example does not process items as they are clicked. This list will be displayed in 12 rows.

```
enum {lstId = 200 };
char* testList[] =
  {
    "First 0",   // The first item in the list
     ...
    "Item N",    // The last item in the list
    0            // 0 terminates the list
  };
  ...
CommandObject ListList[] =
  {
    {C_List, lstId, 0, "A List", (void*)testList,
       CA_NoNotify | CA_Size,isSens,NoFrame,0,0,12},
    {C_Button, M_OK, M_OK, " OK ", NoList,
       CA_DefaultButton, isSens, NoFrame, 0, lstId},
    {C_EndOfList,0,0,0,0,CA_None,0,0,0}
  };
    ...
    vModalDialog ld(this);    // create list dialog
    int lid, lval;
    ...
    ld.AddDialogCmds(ListList);   // Add commands to dialog
    ld.SetValue(lstId,8,Value);  // pre-select 8th item
    lid = ld.ShowModalDialog("",lval);  // Wait for OK
    lval = ld.GetValue(lstId);  // Retrieve the item selected
```

C_ProgressBar



Bar to show progress. Used with `CA_Vertical` or `CA_Horizontal` attributes to control orientation. You change the value of the progress bar with `SetValue(ProgID, val, Value)`, where `val` is a value between 0 and 100, inclusive. Normally, the progress bar will show both a graphical indication of the value, and a text indication of the value between 0 and 100.

If you don't want the text value (for example, your value represents something other than 0 to 100), then define the progress bar with the `CA_NoLabel` attribute. Use the `CA_Percent` attribute to have a % added to the displayed value. You can also use `CA_Small` or `CA_Large` to make the progress bar smaller or larger than normal. If you need a text value display for ranges other than 0 to 100, you can build a `CA_NoSpace` frame with a progress bar and a text label that you modify yourself.

Example

The following shows how to define a progress bar, and how to set its value.

```
enum{frm1 = 200, lbl1, pbrH, pbrV, ... };
  static CommandObject Cmds[] =
  {
    ...
    // Progress Bar in a frame
    {C_Frame, frm1, 0, "",NoList,CA_None,isSens,NoFrame, 0,0},
    {C_Label, lbl1, 0, "Progress",NoList,CA_None,isSens,frm1,0,0},
    {C_ProgressBar, pbrH, 50, "", NoList,
        CA_Horizontal,isSens,frm1, 0, lbl1},  // Horiz, with label

    {C_ProgressBar, pbrV, 50, "", NoList,  // Vertical, no value
      CA_Vertical | CA_Small, isSens,NoFrame, 0, frm1},
    ...
  };
  ...
  // Set the values of both bars to same
  SetValue(pbrH,retval,Value);    // The horizontal bar
  SetValue(pbrV,retval,Value);    // The vertical bar
```

C_RadioButton

○ KOB ● KOAT ○ KRQE ○ KASA

Radio buttons are used to select one and only one item from a group. When the user clicks on one button of the group, the currently set button is turned off, and the new button is turned on. Note that for each radio button press, *two* events are generated. One a call to `DialogCommand` with the id of the button being turned off, and the other a call with the id of the button being turned on. The order of these two events is not guaranteed. The `retVal` field indicates the initial on or off state, and only one radio button in a group should be on.

Radio buttons are grouped by frame. You will typically put a group of radio buttons together in a frame. Any buttons not in a frame (in other words, those just in the dialog window) are grouped together.

Radio buttons are handled very much like check boxes. Your code should dynamically monitor the state of each radio button with the `DialogCommand` method. Selecting Cancel will automatically generate calls to `DialogCommand` to restore the each of the buttons to the original state.

You can use `SetValue` with a `Value` parameter to change the settings of the buttons at runtime. `SetValue` will enforce a single button on at a time.

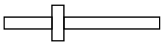
Example

The following example of defining and using radio buttons was extracted from the sample file `v/examp/mydialog.cpp`. It starts with the button `RB1` pushed.

```
enum {
    frmV1 = 200, rdb1, rdb2, rdb3, ...
...
  };
...
static CommandObject DefaultCmds[] =
  {
    {C_Frame, frmV1, 0,"Radios",NoList,CA_Vertical,isSens,NoFrame,0,0},
```

```
    {C_RadioButton, rdb1, 1, "KOB",  NoList,CA_None,isSens, fmV1,0,0},
    {C_RadioButton, rdb2, 0, "KOAT", NoList,CA_None, isSens,frmV1,0,0},
    {C_RadioButton, rdb3, 0, "KRQE", NoList,CA_None, isSens,frmV1,0,0},
    {C_Button, M_Cancel,M_Cancel,"Cancel",NoList,CA_None,
        isSens, NoFrame, 0, frmV1},
    {C_Button, M_OK, M_OK, " OK ", NoList, CA_DefaultButton,
        isSens, NoFrame, M_Cancel, frmV1},
    {C_EndOfList,0,0,0,0,CA_None,0,0,0}
  };
...
void myDialog::DialogCommand(ItemVal Id, ItemVal Val, CmdType Ctype)
  {
    switch (Id)              // switch on command id
      {
        case rdb1:            // Radio Button KOB
            // do something useful - current state is in retval
            break;
        ...
        // cases for other radio buttons

      }
    // let the super class handle M_Cancel and M_OK
    vDialog::DialogCommand(id,retval,ctype);
  }
```

C_Slider



Used to enter a value with a slider handle. The slider will provide your program with a value between 0 and 100, inclusive. Your program can then scale that value to whatever it needs.

V will draw sliders in one of three sizes. Use `CA_Small` for a small slider (which may not be big enough to return all values between 0 and 100 on all platforms), `CA_Large` to get a larger than normal slider, and no attribute to get a standard size slider that will return all values between 0 and 100. Use the `CA_Vertical` and `CA_Horizontal` attributes to specify orientation of the slider.

When the user changes the value of the slider, the `DialogCommand` method is called with the id of the slider for the `Id` value, and the current value of the slider for the `Retval` value. You can use `SetVal` to set a value for the slider.

Example

The following example shows the definition line of a slider, and a code fragment from an overridden `DialogCommand` method to get the value of the dialog and update a `C_Text` item with the current value of the slider. The slider starts with a value of 50.

```
enum { frm1 = 80, sld1, txt1 };
CommandObject Commands[] =
  {
    ...
    {C_Frame, frm1, 0, "",NoList,CA_None,isSens,NoFrame,0,0},
    {C_Slider, sld1, 50, "",NoList,CA_Horizontal,isSens,frm1,0,0},
    {C_Text, txt1, 0, "", "50",CA_None,isSens, frm1, sld1, 0},
    ...
  };
  ...
void testDialog::DialogCommand(ItemVal id,
  ItemVal retval, CmdType ctype)
  {
```

```
    ...
    switch (id)     // Which dialog command item?
      {
        ...
        case sld1:    // The slider
          {
            char buff[20];
            sprintf(buff,"%d",retval);  // To string
            SetString(txt1,buff);      // Show value
          }
        ...
      }
    ...
  }
```

C_Spinner

This command item is used to provide an easy way for the user to enter a value from a list of possible values, or in a range of values. Depending on the attributes supplied to the `CommandObject` definition, the user will be able to select from a short list of text values, from a range of integers, or starting with some initial integer value. As the user presses either the up or down arrow, the value changes to the next permissible value. The `retVal` field specifies the initial value of the integer, or the index of the initial item of the text list. You use the `GetValue` method to retrieve the final value from the `C_Spinner`.

You can change the contents of the spinner list by using `vDialog::SetValue` with either `ChangeList` or `ChangeListPtr`. See `vDialog::SetValue` for more details.

The size of the spin value field in pixels can be controlled by using the `CommandObject` element `size`. By specifying the attribute `CA_Size` and providing a value for the `size` element, you can control the size of the value field . Note the that the `size` element is the last one of a `CommandObject`, and can left out of a declaration, which results in the compiler generating a zero value.

Example

This example shows how to setup the `C_Spinner` to select a value from a text list (when supplied with a list and the `CA_Text` attribute), from a range of integers (when supplied a range list), or from a starting value (when no list is provided). The definitions of the rest of the dialog are not included.

```
  static char* spinList[] =    // a list of colors
    {
      "Red","Green","Blue", 0
    };
  static int minMaxStep[3] =  // specify range of
    {                         // -10 to 10
      -10, 10, 2              // in steps of 2
    };
  enum { spnColor = 300, spnMinMax, spnInt, ... };
  CommandObject SpinDialog[] =
    {
      ...
      {C_Spinner,spnColor,0,"Vbox", // A text list.
        (void*)spinList,CA_Text,     // the list is CA_Text
        isSens,NoFrame, 0,0},
```

```
    {C_Spinner,spnMinMax,0,"Vbox", // a range -10 to 10
      (void*)minMaxStep,CA_None,  // by 2's starting at 0
      isSens,NoFrame, 0,0},
    {C_Spinner,spnInt,32,"Vbox",  // int values step by 1
      NoList,CA_None,             // starting at 32
      isSens,NoFrame, 0,0},
    ...
  };
```

C_Text

This is an example
of a two line text.

This draws boxed text. It is intended for displaying information that might be changed, unlike a label, which is usually constant. The text may be multi-line by using a `'\n'`. The `retVal` and `title` fields are not used. The text to display is passed in the `itemList` field.

You can use the `CA_NoBorder` attribute to suppress the border.

A definition of a `C_Text` item in a `CommandObject` definition would look like:

```
{C_Text, txtId, 0, "", "This is an example\nof a two line text.",
        CA_None,isSens,NoFrame, 0, 0, 0,0},
```

You can change the label of text box with: `SetString(txtId, "New text to show.")`.

C_TextIn

Editable input text _

This command is used for text entry from the user. The text input command item will typically be boxed field that the user can use to enter text.

The strategy for using a TextIn command item is similar to the List command item. You need an OK button, and then retrieve the text after the dialog has been closed.

You can provide a default string in the `title` field which will be displayed in the TextIn field. The user will be able to edit the default string. Use an empty string to get a blank text entry field. The `retVal` field is not used.

There are two ways to control the size of the TextIn control. If you specify `CA_None`, you will get a TextIn useful form most simple input commands. Using `CA_Large` gets a wider TextIn, while `CA_Small` gets a smaller TextIn. You can also use the `size` field of the `CommandObject` to explicitly specify a width in characters. When you specify a size, that number of characters will fit in the TextIn, but the control does *not* enforce that size as a limit.

If you specify the attribute `CA_Password`, then the user's input will either be echoed as asterisks (MS-Windows), or not echoed (X).

Example

The following example demonstrates how to use a TextIn.

```
CommandObject textInList[] =
  {
    ...
    {C_TextIn, txiId,0,"",NoList,CA_None,isSens,NoFrame,0,0},
    ...
    {C_EndOfList,0,0,0,0,CA_None,0,0,0}
```

```
 };
...
   vModalDialog md(this);      /// make a dialog
   int ans, val;
   char text_buff[255];        // get text back to this buffer
...
   md.AddDialogCmds(textInList);  // add commands
   ans = md.ShowModalDialog("Enter text.", val);  // Show it
   text_buff[0] = 0;          // make an empty string
   (void) md.GetTextIn(txiId, text_buff, 254); // get the string
...
```

C_ToggleButton

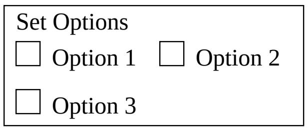
A `C_ToggleButton` is a combination of a button and a checkbox. When the toggle button is pressed, the `vCmdWindow::WindowCommand` method is called, just as with a regular command button. However, the system will change the look of the toggle button to indicate it has been pressed. Each click on a `C_ToggleButton` will cause the button to appear pressed in or pressed out.

The `retVal` field of the `CommandObject` definition is used to indicate the initial state of the toggle.

The behavior of a toggle button is like a check box, and not a radio button. This is more flexible, but if you need exclusive radio button like selection, you will have to enforce it yourself using `SetValue(toggleId,val,Value)`.

```
 // Define a toggle button with id tbtToggle and
 // an initial state of 1, which means pressed in
 {C_ToggleButton,tbtToggle, 1,"", NoList,CA_None,
     isSens, NoFrame, 0, 0},
 ...

 // The case in WindowCommand should be like this:

    case tbtToggle:
      {
// Always safest to retrieve current value
        ItemVal curval = GetValue(tbtToggle);
        // Now, do whatever you need to
        if (curval)
           ... it is pressed
        else
           ... it is not pressed
        break;
      }
```

C_ToggleFrame



A `C_ToggleFrame` is ***V***'s answer to the Windows Tab control. While ***V*** doesn't have real Tab controls, using a combination of `C_ToggleFrames` and either radio buttons or toggle buttons, you can design very nice multi-frame dialogs.

A Toggle Frame works just like a regular `C_Frame` except that you can use `SetValue` with a type `Value` to hide or make visible all controls contained or nested in the toggle frame. (Note: setting the `Value` of a toggle frame is *not* the same as setting its `Hidden` attribute.)

The strategy for using toggle frames follows. First, you will usually use two or more toggle frames together. In the dialog `CommandObject` definition, you first define one radio button or one toggle button for each toggle frame used in the dialog. You then define a regular bordered `C_Frame` positioned below the radio/toggle buttons. Then place `CA_NoBorder` toggle frames inside that outer frame. The outer frame will be the border for all the toggle frames. Inside each toggle frame, you define controls in the normal way.

You must select just *one* of the toggle frames to be initially visible. This will correspond to the checked radio button or pressed toggle button. The remaining toggle frames *and* their controls should all be defined using the `CA_Hidden` attribute.

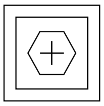
You then hide and unhide toggle frames by responding to the `vDialog::DialogCommand` messages generated when a radio button or toggle button is pressed. You `SetValue(togID, 1, Value)` to show a toggle pane and all its controls, and `SetValue(togID, 0, Value)` to hide all its controls.

The following example shows how to define and control toggle frames:

```
enum {lbl1 = 400, tbt1, tbt2, tbt3, frm1, tfr1, tfr2,
      btnA1, btnB1, btnA2, btnB2 };
static CommandObject DefaultCmds[] =
  {
    // A label, then 2 toggle buttons to select toggle frames
    {C_Label,lbl1,0,"Tab Frame Demo",NoList,CA_None,isSens,
            NoFrame,0,0},
    {C_ToggleButton,tbt1,1,"Tab 1",NoList, CA_None, isSens,
            lbl1, 0, 0},
    {C_ToggleButton,tbt2,0,"Tab 2",NoList, CA_None, isSens,
            lbl1, tbt, 0},
    {C_ToggleButton,tbt3,0,"Tab 3",NoList, CA_None, isSens,
            lbl1, tbt2 0},

    // A Master frame to give uniform border to toggle frames
    {C_Frame,frm1,0, "", NoList,CA_None,isSens,lbl1,0,tbt1},

    // Toggle Frame 1 - default frame on
    {C_ToggleFrame, tfr1,1,"",NoList, CA_NoBorder,isSens,frm1,0,0},
    {C_Button,btnA1,0,"Button A(1)",NoList,CA_None,isSens,tfr1,0,0},
    {C_Button,btnB1,0,"Button B(1)",NoList,CA_None,isSens,tfr1,
             0,btnA1},

    // Toggle Frame 2 - default off (CA_Hidden!)
    {C_ToggleFrame,tfr2,0,"",NoList,CA_NoBorder | CA_Hidden,
           isSens,frm1,0,0},
    {C_Button,btnA2,0,"Button A(2)",NoList,CA_Hidden,isSens,tfr2,0,0},
    {C_Button,btnB2,0,"Button B(2)",NoList,CA_Hidden,isSens,tfr2,
             btnA2,0},
```

```
    {C_EndOfList,0,0,0,0,CA_None,0,0,0}
  };


...

// In the DialogCommand method:

switch (id)          // We will do some things depending on value
  {
    case tbt1:        // For toggle buttons, assume toggle to ON
      {
        SetValue(id,1,Value);     // turn on toggle button
        SetValue(tbt2,0,Value);    // other one off
        SetValue(tfr2,0,Value);    // Toggle other frame off
        SetValue(tfr1,1,Value);    // and ours on
        break;
      }

    case tbt2:        // Toggle 2
      {
        SetValue(id,1,Value);     // turn on toggle button
        SetValue(tbt1,0,Value);    // other off
        SetValue(tfr1,0,Value);    // Toggle other off
SetValue(tfr2,1,Value);    // and ours on
        break;
      }

  }
// All commands should also route through the parent handler
vDialog::DialogCommand(id,retval,ctype);
}
```

C_ToggleIconButton



A C_ToggleIconButton is a combination of an icon button and a checkbox. When the toggle icon button is pressed, the vCmdWindow::WindowCommand method is called, just as with a regular icon button. However, the system will change the look of the toggle icon button to indicate it has been pressed. This is useful for good looking icon based interfaces to indicate to a user that some option has been selected. An additional press will change the appearance back to a normal icon button. The retVal field of the CommandObject definition is used to indicate the initial state of the toggle.

The behavior of a toggle icon button is like a check box, and not a radio button. This is more flexible, but if you need exclusive radio button like selection, you will have to enforce it yourself using SetValue(toggleId,val,Value).

```
// Define a toggle icon button with id tibToggle and
// an initial state of 1, which means pressed
```

```
{C_ToggleIconButton,tibToggle, 1,"", &anIcon,CA_None,
    isSens, NoFrame, 0, 0},
...

// The case in WindowCommand should be like this:

   case tibToggle:
     {
       // Always safest to retrieve current value
       ItemVal curval = GetValue(tibToggle);
       // Now, do whatever you need to
       if (curval)
          ... it is pressed
       else
          ... it is not pressed
       break;
     }
```

6.4 vIcon

Used to define ***V*** icons.

Synopsis

Header: `<v/v_icon.h>`

Class name: vIcon

Description

Icons may be used for simple graphical labels in dialogs, as well as for graphical command buttons in dialogs and command bars. See the sections `vButton` and *Dialog Commands* for descriptions of using icons.

Presently, ***V*** supports monochrome icons which allow an on or off state for each pixel, and color icons of either 256 or 2^{24} colors. The format of ***V*** monochrome icons is identical to the X bitmap format. This is a packed array of unsigned characters (or bytes), with each bit representing one pixel. The size of the icon is specified separately from the icon array. The ***V*** color icon format is internally defined, and allows easy conversion to various color file formats used by X and Windows.

Definition

```
class vIcon     // an icon
  {
  public:             //---------------------------------------- public
    vIcon(unsigned char* ic, int h, int w, int d = 1);
    ~vIcon();
    int height;             // height in pixels
    int width;              // width in pixels
    int depth;              // bits per pixel (1,8, or 24)
    unsigned char* icon;    // ptr to icon array

  protected:        //--------------------------------------- protected
```

```
private:          //------------------------------------- private
};
```

Constructor

vIcon(unsigned char* icon, int height, int width, int depth = 1)

The constructor for a `vIcon` has been designed to allow you to easily define an icon. The first parameter is a pointer to the static icon array. (Note: `vIcon` does not make a copy of the icon - it needs to be a static or persistent definition in your code.) The second and third parameters specify the height and width of the icon. The last parameter specifies depth.

Class Members

int height This is the height in pixels of the icon.

int width This is the width in pixels of the icon. A icon will thus require (height * width) pixels. These bits are packed into bytes, with 0's padding the final byte if needed.

int depth For monochrome icons, this will be one. For color icons, the value is either 8 (for 2^8 or 256 colors) or 24 (for 2^{24} colors).

unsigned char* icon This is a pointer to the array of bytes that contain the icon. ***V*** basically uses the format defined by X (`.XBM`) bitmaps for monochrome bitmaps. It uses an internal format consisting of a color map followed by a one byte per pixel color icon description, or a three bytes per pixel color icon description.

Defining Icons

The easiest way to define an icon is to include the definition of it in your code (either directly or by an `#include`). You then provide the address of the icon data plus its height and width to the initializer of the `vIcon` object.

The ***V*** distribution includes a simple icon editor that can be used to create and edit icons in standard `.vbm` format, as well as several other formats. You can also generate monochrome icons is with the X `bitmap` utility. That program allows you to draw a bitmap, and then save the definition as C code. This code can be included directly in your code and used in the initialization of the `vIcon` object. If you follow the example, you should be able to modify and play with your icons very easily.

A simple converter that converts a Windows `.bmp` format file to a ***V*** `.vbm` ***V*** bitmap format is also included in the standard ***V*** distribution. There are many utilities that let you generate `.bmp` files on both Windows and X, so this tool easily lets you add color icons of arbitrary size. Chapter 9 has more details on `bmp2vbm`.

The standard ***V*** distribution also contains a directory (`v/icons`) with quite a few sample icons suitable for using in a command bar.

Once you have a `.vbm` file, the easiest way to add an icon to your program is to include code similar to this in your source:

```
#include "bruce.vbm"    // Picture of Bruce
  static vIcon bruceIcon(&bruce_bits[0], bruce_height,
                         bruce_width,8);
```

The following sections describe the format of the `unsigned char* icon` data for 1, 8, and 24 bit ***V*** icons.

1 Bit Icons

Icon definitions are packed into bytes. A bit value of 1 represents Black, a 0 is White. The bytes are arranged by rows, starting with the top row, with the bytes padded with leading zeros to come out to whole bytes. The bytes are scanned in ascending order (`icon[0]`, `icon[1]`, etc.). Within bytes, the bits are scanned from LSB to MSB. A 12 bit row with the pattern BBBWWBBBWBWBW would be represented as `unsigned char row[ ] = { 0x67, 0x05 };`. This is the format produced by the X `bitmap` program.

8 Bit Icons

Eight bit icons support 256 colors. Each pixel of the icon is represented by one byte. Bytes are arranged in row order, starting with the top row. Each byte represents an index into a color map. The color map consists of RGB byte entries. While an 8 bit icon can only have 256 colors, it can map into 2^{24} possible colors. Thus, each 8 bit icon must also include the color map as part of its data. The very first byte of the `icon` data is the number of entries in the color map *minus one*[1] (you don't have to define all 256 colors), followed by the color map RGB bytes, followed by the icon pixels. The following is a very simple example of an icon:

```
//vbm8
#define color_width 16
#define color_height 12
#define color_depth 8
static unsigned char color_bits[] = {
       2,       // 3 colors in color map (2 == 3-1)
       255,0,0, // byte value 0 maps to red
       0,255,0, // 1 -> green
       0,0,255, // 2 -> blue
       // Now, the pixels: an rgb "flag", 3 16x4 rows
       0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, // RRRRRRRRRRRRRRRR
       0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0, // RRRRRRRRRRBBBBBR
       0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0, // RRRRRRRRRRBBBBBR
       0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, // RRRRRRRRRRRRRRRR
       1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, // GGGGGGGGGGGGGGGG
       1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, // GGGGGGGGGGGGGGGG
       1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, // GGGGGGGGGGGGGGGG
       1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, // GGGGGGGGGGGGGGGG
       2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2, // BBBBBBBBBBBBBBBB
       2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2, // BBBBBBBBBBBBBBBB
       2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2, // BBBBBBBBBBBBBBBB
       2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2  // BBBBBBBBBBBBBBBB
     };

static vIcon colorIcon(&color_bits[0], color_height, color_width,
     color_depth);
```

24 Bit Icons

Twenty-four bit icons are arranged in rows, staring with the top row, of three bytes per pixel. Each 3 byte pixel value represents an RGB value. There is no color map, and the RGB pixel values start immediately in the `unsigned char* icon` data array. This is a simple example of a 24 bit icon.

```
//vbm24
#define c24_height 9
#define c24_width 6
```

[1] This is necessary keep things as `chars` and still allow a possible 256 entries, since 256 is $2^8 + 1$, and a color map with 0 entries doesn't make sense.

```
#define c24_depth 24
    static unsigned char c24_bits[] = {
     255,0,0,255,0,0,255,0,0,255,0,0,0,255,0,0,255,0, //RRRRGG
     255,0,0,255,0,0,255,0,0,255,0,0,0,255,0,0,255,0, //RRRRGG
     255,0,0,255,0,0,255,0,0,255,0,0,255,0,0,255,0,0, //RRRRRR
     0,255,0,0,255,0,0,255,0,0,255,0,0,255,0,0,255,0, //GGGGGG
     0,255,0,0,255,0,0,255,0,0,255,0,0,255,0,0,255,0, //GGGGGG
     0,255,0,0,255,0,0,255,0,0,255,0,0,255,0,0,255,0, //GGGGGG
     0,0,255,0,0,255,0,0,255,0,0,255,0,0,255,0,0,255, //BBBBBB
     0,0,255,0,0,255,0,0,255,0,0,255,0,0,255,0,0,255, //BBBBBB
     0,0,255,0,0,255,0,0,255,0,0,255,0,0,255,0,0,255  //BBBBBB
    };
    static vIcon c24Icon(&c24_bits[0], c24_height, c24_width,
        c24_depth);
```

Example

This example uses the definition of the checked box used by the Athena checkbox dialog command.

```
// This code is generated by the V Icon Editor:
//vbm1
#define checkbox_width 12
#define checkbox_height 12
#define checkbox_depth 1
static unsigned char checkbox_bits[] = {
   0xff, 0x0f, 0x03, 0x0c, 0x05, 0x0a, 0x09, 0x09,
   0x91, 0x08, 0x61,  0x08, 0x61, 0x08, 0x91, 0x08,
   0x09, 0x09, 0x05, 0x0a, 0x03, 0x0c, 0xff, 0x0f};

// This code uses the above definitions to define an icon
// in the initializer of checkIcon to vIcon.

static vIcon checkIcon(&checkbox_bits[0],
    checkbox_height, checkbox_width, checkbox_depth);
```

See Also

vButton, Dialog Commands C_Icon and C_IconButton

6.5 vDialog

Class to build a modeless dialog.

Synopsis

Header: `<v/vdialog.h>`

Class name: vDialog

Hierarchy: (vBaseWindow,vCmdParent) → vDialog

Contains: CommandObject

Description

The `vDialog` class is used to build modeless dialogs. Since most dialogs will require a response to the commands they define, you will almost always derive your own subclass based on `vDialog`, and override the `DialogCommand` method to handle those commands. Note that `vDialog` is multiply derived from the `vBaseWindow` and the `vCmdParent` classes.

Constructor

vDialog(vBaseWindow* parent)

vDialog(vApp* parent)

vDialog(vBaseWindow* parent, int isModal = 0, char* title = "")

vDialog(vApp* parent, int isModal = 0, char* title = "")

A dialog is constructed by calling it with a pointer to a vBaseWindow or vApp, which is usually the 'this' of the object that creates the `vDialog`. The `isModal` parameter indicates if the dialog should be modal or modeless. You would usually use the default of 0. The modal flag is used by the derived `vModalDialog` class. The `title` parameter can be used to set a title for your dialog (see `SetDialogTitle` for information on titles). If you create a derived dialog class, you might provide a `parent` and a `title` in your constructor, and provide the 0 for the `isModal` flag in the call to the `vDialog` constructor.

The constructor builds an empty dialog. The `AddDialogCmds` method must be called in order to build a useful dialog, which you would usually do from within the constructor of your derived dialog class.

IMPORTANT! When you derive your own `vDialog` objects, you should write constructors for both the `vBaseWindow*` and `vApp*` versions. These two different constructors allow dialogs to be used both from windows directly, and from the `vApp` code as well. Normally, you would construct a dialog from a window. Occasionally, it will be useful to build a dialog from the vApp that applies to all windows, and not just the window that constructed it.

void vDialog::AddDialogCmds(CommandObject* cList)

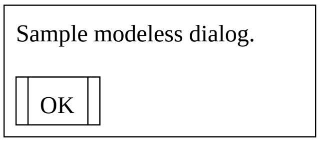
This method is used to add a list of commands to a dialog. It is called after the dialog object has been created. You can usually do this in the constructor for your derived Dialog class. This method is passed an array of `CommandObject` structures.

void vDialog::SetDialogTitle(char* title)

This can be used to dynamically change the title of any object derived from a `vDialog` object. Note that the title will not always be displayed. This depends on the host system. For example, the user can set up their X window manager to not show decorations on transient windows, which is how dialogs are implemented on X. You should write your applications to provide a meaningful title as they are often helpful when displayed.

Example

This example shows the steps required to use a dialog object. Note that the example uses the `vDialog` class directly, and thus only uses the default behavior of responding to the `OK` button.




```
#include <v/vdialog.h>
    CommandObject cmdList[] =           // list of the commands
      {
        {C_Label, lbl1, 0, "Label",NoList,CA_MainMsg,isSens,0,0},
        {C_Button, M_OK, M_OK, " OK ", NoList,
            CA_DefaultButton, isSens,lbl1,0},
        {C_EndOfList,0,0,0,0,CA_None,0,0}  // This ends list
      };
    ...
    vDialog curDialog(this,0,"Sample Dialog"); // create dialog instance

    curDialog.AddDialogCmds(cmdList);          // add the commands

    curDialog.ShowDialog("Sample modeless dialog."); // invoke
    ...
```

This example creates a simple modeless dialog with a label and an OK button placed below the label (see the description of layout control below). `ShowDialog` displays the dialog, and the `vDialog::DialogCommand` method will be invoked with the id (2) and value (`M_OK`) of the OK button when it is pressed.

Use the `vModalDialog` class to define modal dialogs.

The `CommandObject` structure includes the following:

```
    typedef struct CommandObject
      {
        CmdType cmdType;    // what kind of item is this
        ItemVal cmdId;      // unique id for the item
        ItemVal retVal;     // initial value
                            //  depends on type of command
        char* title;        // string for label or title
        void* itemList;     // a list of stuff to use for the cmd
                            //  depends on type of command
        CmdAttribute attrs; // list of attributes of command
        unsigned
             Sensitive:1;   // if item is sensitive or not
        ItemVal cFrame;     // if item part of a frame
        ItemVal cRightOf;   // Item placed left of this id
        ItemVal cBelow;     // Item placed below this one
        int size;           // Used for size information
char* tip;          // tool tip string
      } CommandObject;
```

Placements of command objects within the dialog box are controlled by the `cRightOf` and `cBelow` fields. By specifying where an object goes in relation to other command objects in the dialog, it is simple to get a very pleasing layout of the dialog. The exact spacing of command objects is controlled by the `vDialog` class, but the application can used `C_Blank` command objects to help control spacing.

The various types of command objects that can be added include (with suggested id prefix in parens):

```
    C_EndOfList:   Used to denote end of command list
    C_Blank:       filler to help RightOfs, Belows work (blk)
    C_BoxedLabel:  a label with a box (bxl)
    C_Button:      Button (btn)
```

```
C_CheckBox:    Checked Item (chk)
C_ColorButton: Colored button (cbt)
C_ColorLabel:  Colored label (clb)
C_ComboBox:    Popup combo list (cbx)
C_Frame:       General purpose frame (frm)
C_Icon:        a display only Icon (ico)
C_IconButton:  a command button Icon (icb)
C_Label:       Regular text label (lbl)
C_List:        List of items (lst)
C_ProgressBar: Bar to show progress (pbr)
C_RadioButton: Radio button (rdb)
C_Slider:      Slider to enter value (sld)
C_Spinner:     Spinner value entry (spn)
C_TextIn:      Text input field (txi)
C_Text:        wrapping text out (txt)
C_ToggleButton: a toggle button (tbt)
C_ToggleFrame: a toggle frame (tfr)
C_ToggleIconButton:  a toggle Icon button (tib)
```

The use of these commands is described in the `DialogCommand` section.

virtual void CancelDialog()

This method is used to cancel any action that took place in the dialog. The values of any items in the dialog are reset to their original values, and the This method is automatically invoked when the user selects a button with the value `M_Cancel` and the `DialogCommand` method invoked as appropriate to reset values of check boxes and so on. `CancelDialog` can also be invoked by the application code.

virtual void CloseDialog()

The `CloseDialog` is used to close the dialog. It can be called by user code, and is automatically invoked if the user selects the `M_Done` or `M_OK` buttons and the the user either doesn't override the `DialogCommand` or calls the default `DialogCommand` from any derived `DialogCommand` methods.

virtual void DialogCommand(ItemVal Id, ItemVal Val, CmdType Type)

This method is invoked when a user selects some command item of the dialog. The default `DialogCommand` method will normally be overridden by a user derived class. It is useful to call the default `DialogCommand` from the derived method for default handling of the `M_Cancel` and `M_OK` buttons.

The `Id` parameter is the value of the `cmdId` field of the `CommandObject` structure. The `Val` parameter is the `retVal` value, and the `Type` is the `cmdType`.

The user defined `DialogCommand` is where most of the work defined by the dialog is done. Typically the derived `DialogCommand` will have a `switch` statement with a `case` for each of the command `cmdId` values defined for items in the dialog.

void DialogDisplayed()

This method is called by the ***V*** runtime system after a dialog has actually been displayed on the screen. This method is especially useful to override to set values of dialog controls with `SetValue` and `SetString`.

It is important to understand that the dialog does not get displayed until `ShowDialog` or `ShowModalDialog` has been called. There is a very important practical limitation implied by this, especially for modal dialogs. The values of controls *cannot* be changed until the dialog has been displayed, even though the `vDialog` object may exist. Thus, you can't call `SetValue` or `SetString` until after you call `ShowDialog` for modeless dialogs, or `ShowModalDialog` for modal dialogs. Since `ShowModalDialog` does not return until the user has closed the dialog, you must override `DialogDisplayed` if you want to change the values of controls in a modal dialog dynamically.

For most applications, this is not a problem because the static definitions of controls in the `CommandObject` definition will be usually be what is needed. However, if you need to create a dialog that has those values changed at runtime, then the easiest way is to include the required `SetValue` and `SetString` calls inside the overridden `DialogDisplayed`.

void GetDialogPosition(int& left, int& top, int& width, int& height)

Returns the position and size of `this` dialog. These values reflect the actual position and size on the screen of the dialog. The intent of this method is to allow you to find out where a dialog is so position it so that it doesn't cover a window.

virtual int GetTextIn(ItemVal Id, char* str, int maxlen)

This method is called by the application to retrieve any text entered into any `C_TextIn` items included in the dialog box. It will usually be called after the dialog is closed. You call `GetTextIn` with the `Id` of the TextIn command, the address of a buffer (`str`), and the size of `str` in `maxlen`.

virtual int GetValue(ItemVal Id)

This method is called by the user code to retrieve values of command items, usually after the dialog is closed. The most typical use is to get the index of any item selected by the user in a `C_List` or `C_ComboBox`.

int IsDisplayed()

This returns true if the dialog object is currently displayed, and false if it isn't. Typically, it will make sense only to have a single displayed instance of any dialog, and your code will want to create only one instance of any dialog. Since modal dialogs allow the user to continue to interact with the parent window, you must prevent multiple calls to `ShowDialog`. One way would be to make the command that displays the dialog to be insensitive. `IsDisplayed()` is provided as an alternative method. You can check the `IsDisplayed()` status before calling `ShowDialog`.

virtual void SetDialogPosition(int left, int top)

Moves `this` dialog to the location `left` and `top`. This function can be used to move dialogs so they don't cover other windows.

virtual void SetValue(ItemVal Id, ItemVal val, ItemSetType type)

This method is used to change the state of dialog command items. The `ItemSetType` parameter is used to control what is set. Not all dialog command items can use all types of settings. The possibilities include:

Checked The `Checked` type is used to change the checked status of check boxes. ***V*** will normally handle checkboxes, but if you implement a command such as *Check All*, you can use `SetValue` to change the check state according to `ItemVal val`.

Sensitive The `Sensitive` type is used to change the sensitivity of a dialog command.

Value The `Value` type is used primarily to preselect the item specified by `ItemVal val` in a list or combo box list.

ChangeList, ChangeListPtr Lists, Combo Boxes, and Spinners use the `itemList` field of the defining `CommandObject` to specify an appropriate list. `SetValue` provides two ways to change the list values associated with these controls.

The key to using `ChangeListPtr` and `ChangeList` is an understanding of just how the controls use the list. When a list type control is instantiated, it keeps a private copy of the pointer to the original list as specified in the `itemList` field of the defining `CommandObject`.

So if you want to change the original list, then `ChangeList` is used. The original list may be longer or shorter, but it must be in the same place. Remember that a NULL entry marks the end of the list. So you could allocate a 100 item array, for example, and then reuse it to hold 0 to 100 items.

Call `SetValue` with `type` set to `ChangeList`. This will cause the list to be updated. Note that you must not change the `itemList` pointer used when you defined the list or combo box. The contents of the list can change, but the pointer must be the same. The `val` parameter is not used for `ChangeList`.

Sometimes, especially for regular list controls, a statically sized list just won't work. Using `ChangeListPtr` allows you to use dynamically created list, but with a small coding penalty. To use `ChangeListPtr`, you must first modify the contents of the `itemList` field of the original `CommandObject` definition to point the the new list. Then call `SetValue` with `ChangeListPtr`. Note that this will both update the pointer, and update the contents of the list. You *don't* need to call again with `ChangeList`.

The following illustrates using both types of list change:

```
char* comboList[] = {
  "Bruce", "Katrina", "Risa", "Van", 0 };
char* list1[] = {"1", "2", "3", 0};
char* list2[] = {"A", "B", "C", "D", 0};

// The definition of the dialog
CommandObject ListExample[] = {
  {C_ComboBox,100,0,"",(void*)comboList,CA_None,isSens,0,0,0},
  {C_List,200,0,"",(void*)list1,CA_None,isSens,0,0,0},
  ...
  };
 ...

  // Change the contents of the combo list
  comboList[0] = "Wampler";  // Change Bruce to Wampler
  SetValue(200,0,ChangeList);
 ...
  // Change to a new list entirely for list
  // Note that we have to change ListExample[1], the
  // original definition of the list control.
  ListExample[1].itemList = (void*)list2;  // change to list2
  SetValue(100,0,ChangeListPtr);
 ...
```

Note that this example uses static definitions of lists. It is perfectly fine to use completely dynamic lists: you just have to dynamically fill in the appropriate `itemList` field of the defining `CommandObject`.

Please see the description of `DialogDisplayed` for an important discussion of setting dialog control values.

virtual void SetString(ItemVal Id, char* str)

This method is called to set the string values of dialog items. This can include the labels on check boxes and radio buttons and labels, as well as the text value of a Text item.

Please see the description of `DialogDisplayed` for an important discussion of setting dialog control values.

virtual void ShowDialog(char* message)

After the dialog has been defined, it must then be displayed by calling the `ShowDialog` method. If a `C_Label` was defined with a `CA_MainMsg` attribute, then the message provided to `ShowDialog` will be used for that label.

`ShowDialog` returns to the calling code as soon as the dialog is displayed. It is up to the `DialogCommand` method to then handle command input to the dialog, and to close the dialog when done.

Please see the description of `DialogDisplayed` for an important discussion of setting dialog control values.

Derived Methods

None.

Inherited Methods

None.

See Also

vModalDialog

6.6 vModalDialog

Used to show modal dialogs.

Synopsis

Header: `<v/vmodald.h>`

Class name: vModalDialog

Hierarchy: (vBaseWindow,vCmdParent) → vDialog → vModalDialog

Contains: CommandObject

Description

This class is an implementation of a modal dialog. This means that the dialog grabs control, and waits for the user to select an appropriate command from the dialog. You can use any of the methods defined by the `vDialog` class, as well as the new `ShowModalDialog` method.

Constructor

vModalDialog(vBaseWindow* parent, char* title)

vModalDialog(vApp* parent, char* title)

There are two versions of the constructor, one for constructing dialogs from windows, the other from the vApp object. See the description of the `vDialog` constructor for more details.

The default value for the title is an empty string, so you can declare instances of modal dialogs without the title string if you wish. The dialog title will always show in Windows, but in X is dependent on how the window manager treats decorations on transient windows.

New Methods

virtual ItemVal ShowModalDialog(char* message, ItemVal& retval)

This method displays the dialog, and does not return until the modal dialog is closed. It returns the id of the button that caused the return, and in `retval`, the value of the button causing the return as defined in the dialog declaration.

Please see the description of `DialogDisplayed` for an important discussion of setting dialog control values.

There are a couple of ways to close a modal dialog and make `ShowModalDialog` return, all controlled by the `DialogCommand` method. The default `DialogCommand` will close the modal dialog automatically when the user clicks the `M_Cancel`, `M_Done`, or `M_OK` buttons.

All command actions are still passed to the virtual `DialogCommand` method, which is usually overridden in the derived class. By first calling `vModalDialog::DialogCommand` to handle the default operation, and then checking for the other buttons that should close the dialog, you can also close the dialog by calling the `CloseDialog` method, which will cause the return.

The following code demonstrates this.

```
void myModal::DialogCommand(ItemVal id, ItemVal val,
    CmdType ctype)
  {
    // Call the parent for default processing
    vModalDialog::DialogCommand(id,val,ctype);
    if (id == M_Yes || id == M_No) // These close, too.
        CloseDialog();
  }
```

Derived Methods

virtual void DialogCommand(ItemVal Id, ItemVal val, CmdType type)

Adds a little functionality for handling this modally.

Inherited Methods

vDialog(vBaseWindow* parent)

vDialog(vBaseWindow* parent, int modalflag)

vDialog(vApp* parent)

vDialog(vApp* parent, int modalflag)

void vDialog::AddDialogCmds(CommandObject* cList)

virtual void CancelDialog()

virtual void CloseDialog()

virtual int GetTextIn(ItemVal Id, char* str, int maxlen)

virtual int GetValue(ItemVal Id)

virtual void SetValue(ItemVal Id, ItemVal val, ItemSetType type)

virtual void SetString(ItemVal Id, char* str)

virtual void ShowDialog(char* message)

See Also

vDialog

Chapter 7

Drawing

This chapter covers classes and utility functions needed to draw text and graphics.

The classes and objects covered in this chapter include:

Introduction to Drawing Basic ***V*** drawing model.

Fonts Various screen fonts are available in ***V***.

vBrush A brush for filling areas.

vCanvasPaneDC The canvas pane drawing canvas.

vCanvasPane A base class to build graphical and text canvas panes.

vBaseGLCanvasPane A specialized class to support OpenGL.

vColor A class for specifying colors.

vDC A base class describing drawing canvas methods.

vMemoryDC A memory drawing canvas.

vPen A drawing pen.

vPrintDC A printer drawing canvas.

vPrinter A printer setup dialog.

vTextCanvasPane A class for drawing text on a canvas.

vTextEditor A class for editing text.

vBaseGLCanvasPane A class to support OpenGL.

7.1 Introduction to Drawing

The basic ***V*** model of drawing is a canvas. ***V*** supports several kinds of drawing canvases. The most obvious canvas is the screen drawing canvas. This will often be the main or even only canvas you use. ***V*** also supports printing canvases. Each kind of canvas has identical drawing methods, so you can write code to draw that is mostly independent of which kind of canvas is being used.

There is also a specialized drawing canvas to support OpenGL. This class differs somewhat from the other drawing canvases.

Drawing with the vDC Class

You draw to the various canvases using a `vDC` class, the general ***V*** Drawing Canvas Class (the OpenGL canvas does not use the codevDC class). The `vDC` class for drawing to the screen is `vCanvasPaneDC`. The class `vPrintDC` is the platform independent class to draw to a printer. For X, `vPrintDC` supports PostScript printing. The Windows version supports standard Windows printers. (You can also use the PostScript DC independently on Windows.) If you write your drawing code to use a `vDC` pointer, you will be able to draw to several canvases just by changing the value of the pointer.

Each `vDC` supports the methods described in the `vDC` section. Because the `vCanvasPane` class is so central to most applications, it duplicates all the `vDC` methods so you can call them directly from your `vCanvasPane` object. In fact, all the methods in `vCanvasPane` are just calls to the corresponding `vDC` using the `vCanvasPaneDC` of the canvas pane. You can get the `vCanvasPaneDC` pointer with the `GetDC` method.

There are three kinds of drawing methods supported by ***V***. The simplest methods draw lines of various widths and colors using the current `vPen`. You change the color and width of the lines being drawn by setting the current `vPen` with the `SetPen` method.

The second type of drawing includes filling the space surrounded by a shape such as a polygon. The edges of the shape are drawn using the current `vPen`. The filled area is drawn using the current `vBrush`. You can set various attributes of the brush, and use `SetBrush` to change how the shapes will be filled, as well as changing the attributes of the `vPen` used to draw the surrounding line. Both the pen and the brush can be transparent, allowing you to draw unfilled outline shaped, or to fill a shape without an outline.

Finally, ***V*** supports drawing of text on a canvas using various `vFonts` and text attributes. The canvas pane will start out using the default system font (`vfSystemDefault`). If you need a different initial font, use `vFont::SetFontValues` to select the font you want, then `vCanvasPane::SetFont` to set the new font.

7.1.1 Coordinates

All ***V*** drawing canvas classes use integer physical coordinates appropriate to the canvas. All devices call the upper left corner x,y coordinate of the drawing canvas 0,0. The x values increase to the right, and y values increase down.

It it up to each application to provide appropriate mapping from the coordinates used for the particular model being used (often called the world coordinate system) to the physical mapping used by each ***V*** drawing canvas. Each drawing canvas will have a physical limit for the maximum x and maximum y, usually imposed by the particular canvas (a screen or a paper size, for example). You can set a scale factor for each drawing canvas which can be helpful for using different kinds of drawing canvases. ***V*** also supports setting an x,y translation. This will allow you to more easily use the scroll bars and set margins on printers. Your application can usually use the messages received from the scroll bars to set the translation coordinates to map your the canvas to a different drawing area. The system will handle clipping.

However, the application is for the most part responsible for determining all coordinate mapping – translations of a viewport of the drawing, determining the scaling for various drawing canvases, and any mapping from the world to the physical coordinates. The application will have to map the mouse input accordingly, too.

7.2 Fonts

Various screen fonts are available in ***V***.

Synopsis

Class: vFont

Header: <v/vfont.h>

Description

Fonts are difficult to make portable. ***V*** has adopted a font model that is somewhat portable, yet allows you to take advantage of various fonts available on different platforms. In fact, it is possible to write your programs to use the `vFontSelect` dialog class, and pretty much ignore many of the details of selecting fonts. The main characteristics of fonts your program will have to deal with are the height and width of text displayed on a canvas. These values are provided by `vDC::TextHeight` and `vDC::TextWidth`. Use these values to calculate how much space a text string will take up on the screen or page.

Fonts are associated with drawing canvases. For example, the `vCanvasPane::SetFont` method is used to set the font used by the canvas pane. The sizes of the actual fonts will probably differ on different kinds of canvases. Specifically, your program should not depend on getting the same `TextWidth` value for screen and printer canvases for the same font.

The class `vFont` is used to define font objects, and the characteristics of the font are set either by the class constructor when the font is instantiated, or by using the `vFont :: SetFontValues` method. The utility class `vFontSelect` can be used to interactively set font characteristics. The characteristics associated with a font are described in the following sections. Remember, however, that `vFontSelect::FontSelect` can be used to set these attributes.

Font Family

Each font belongs to a font family. There are eight font families defined by ***V*** with the `vFontID` attribute of the font object. Font families typically correspond to some typeface name such as *Helvetica* or *Times Roman*, but use more generic names. There are three system fonts, `vfDefaultSystem`, `vfDefaultFixed`, and `vfDefaultVariable`. These default fonts are defined by the specific platform. `vfDefaultSystem` will usually be a fixed space font, and is often settable by the user. On X, for example, the default system font can be changed by using a `-fn fontname` switch when starting the application. The `vfDefaultSystem` font will have fixed attributes, and will not be changeable by the program. The `vfDefaultFixed` (fixed spacing) and `vfDefaultVariable` (variable spacing) fonts are also system specified, but can usually have their attributes, such as size and weight changed.

V also supports five other font families. The `vfSerif` font is a seriffed font such as `Times Roman`. The `vfSanSerif` is a serifless font such as `Swiss` or `Lucidia`. Both of these are variable spaced fonts. The `vfFixed` is a fixed space font, often called `Courier` on the host platform. The `vfDecorative` font usually contains symbols or other drawing characters. It is not very portable across platforms. Finally, ***V*** supports a font family called `vfOther`. This is used when the system supports other fonts that are selectable via the `vFontSelect` dialog class. Windows supports a wide variety of fonts, while X does not support any additional fonts.

Font Style

V supports two kinds of font styles: `vfNormal` for normal fonts, and `vfItalic` for italic fonts.

Font Weight

V supports two kinds of font weights: `vfNormal` for normal weight fonts, and `vfBold` for boldface fonts.

Point Size

V supports a wide range of point size, usually ranging from 8 point to 40 or 72 point fonts. Not all point sizes are supported on each platform. How each point size maps to space on the screen or page also vary from platform to platform.

Underlining

You can also specify that a font is underlined.

Angled text

You can specify that a font is to be drawn at something other than horizontally, left to right. If you need a vertical font, for a graph perhaps, you can specify an angle in the font constructor. This means you must use one of V's standard fonts, and can't use the font select dialog. You also can't dynamically change the angle on the fly. If you need text at more than one angle, you need to create multiple instances of a vFont.

You specify the angle in degrees, with 0 representing standard horizontal text. Using 90 degrees gives vertical text, reading from bottom to top. Using 180 gives upside down horizontal text, and 270 gives vertical text, top to bottom. A simple example:

```
vFont font90(vfSansSerif,10,vfNormal,vfNormal,0,90);
myCanvas->SetFont(font90);
myCanvas->DrawText(200,150,"Vertical Text");
```

The Windows version supports any arbitrary angle. The X version only supports 90, 180, and 270. Because X does not provide native support for non-horizontal text, the initial implementation of angled text (V Version 1.18) simulates angled text by drawing standard horizontal characters vertically or backwards. It doesn't look too bad, and is better than having to do it yourself. Real vertical text will probably be supported someday, and I will probably forget to remove this note when that happens, so go by the release notes.

Methods

vFont(vFontID fam = vfDefaultFixed, int size = 10, vFontID sty = vfNormal, vFontID wt = vfNormal, int und = 0, int angle = 0)

The constructor is used to declare a font with the specified family, size, style, weight, underline, and angle.

vFontID GetFamily()

Returns the family of the font object.

int GetPointSize()

Returns the point size of the font object.

vFontID GetStyle()

Returns the style of the font object.

vFontID GetWeight()

Returns the weight of the font object.

int GetUnderlined()

Returns the underline setting of the font object.

void SetFontValues(vFontID fam = vfDefaultFixed, int size = 10, vFontID sty = vfNormal, vFontID wt = vfNormal, int und = 0)

Changes the attributes of the font object. For example, the font selection dialog uses this method to change the font attributes. Note that you can't use this method to set font angles.

7.3 vBrush

A class to specify the brush used to fill shapes.

Synopsis

Header: `<v/vbrush.h>`

Class name: vBrush

Description

Brushes are used to fill shapes. Brushes have two attributes, including color and style.

Methods

vBrush(unsigned int r = 0, unsigned int g = 0, unsigned int b = 0, int style = vSolid)

The brush constructor allows you to set the initial color and style of the brush. The default constructs a solid black brush.

int operator == , !=

You can use the operators `==` and `!=` for comparisons.

vColor GetColor()

This method returns the current color of the brush as a `vColor` object.

int GetFillMode()

This method returns the fill mode of the brush (either `vAlternate` or `vWinding`).

int GetStyle()

This method returns the current style of the brush.

void SetColor(vColor& c)

You can use this method to set the brush color by passing in a `vColor` object.

int SetFillMode(int fillMode)

This method sets the fill mode of the brush. The fillMode parameter specifies one of two alternative filling algorithms, `vAlternate` or `vWinding`. These algorithms correspond to the equivalent algorithms on the native platforms.

void SetStyle(int style)

This method is used to set the style of the brush. Brush styles include:

vSolid The brush fills with a solid color.

vTransparent The brush is transparent, which allows you to draw unfilled shapes.

vHorizontalHatch The brush fills with a horizontal hatch pattern in the current color.

vVerticleHatch The brush fills with a vertical hatch pattern.

vLeftDiagonalHatch The brush fills with a left leaning diagonal hatch pattern.

vRightDiagonalHatch The brush fills with a right leaning diagonal hatch pattern.

vCrossHatch The brush fills with a vertical and horizontal cross hatch pattern.

vDiagonalCrossHatch The brush fills with a diagonal cross hatch pattern.

7.4 vCanvasPaneDC

The drawing canvas class for CanvasPanes.

Synopsis

Header: `<v/vcpdc.h>`

Class name: vCanvasPaneDC

Hierarchy: vDC → vCanvasPaneDC

Description

This class is normally automatically used by the `vCanvasPane` class. It provides the actual implementation of the screen drawing canvas class.

7.5 vCanvasPane

A base class to build graphical and text canvas panes.

Synopsis

Header: `<v/vcanvas.h>`

Class name: vCanvasPane

Hierarchy: vPane → vCanvasPane

Description

This is the base drawing class. You use it to build more complicated drawing canvases, either for graphical drawing or text drawing. The `vCanvasPane` class has all the basic methods needed to interact with the drawing canvas. It does not, however, know how to handle repainting the screen on `Redraw` or `Resize` events. It provides utility methods for drawing on the canvas, and several other methods that are normally overridden by your application.

See the section `vPane` for a general description of panes.

Utility Methods

The following methods provide useful service without modification. Sometimes you will want to override some of these, but you will then usually call these methods from your derived class.

Drawing

The `vCanvasPane` normally creates a `vCanvasPaneDC` to use for drawing, and class provides direct support by including direct calls for the drawing methods described in the `vDC` section. If your drawing will only be to the screen, then you can use the methods of the `vCanvasPane` class directly. Each of these methods is really an inline function that expands to `_cpDC->DrawWhatever()`.

If your drawing code might want to draw to both a screen and a printer, you might want to use a parameter to the appropriate drawing canvas. You can get the `vDC` used by the `vCanvasPane` by calling `GetDC()`.

virtual void CreateDC(void)

This method is called when the `vCanvasPane` is initialized. The default is to create a drawing canvas using `_cpDC = new vCanvasPaneDC(this);`. If you want to derive a different canvas pane class from `vCanvasPane` perhaps using a more sophisticated drawing canvas derived from the `vCanvasPaneDC` class, you can override the `CreateDC` method and set the protected `vDC* _cpDC` pointer to an instance of your new drawing canvas (e.g., `_cpDC = new myCanvasPaneDC(this)` instead.

vDC* GetDC()

Returns a pointer to the `vDC` of the current drawing canvas. The `vDC` can be used for most of the drawing methods to achieve drawing canvas independence. If your code draws via a `vDC` pointer, then the same code can draw to the screen canvas or the printer canvas depending on what the `vDC` points to.

VCursor GetCursor()

Returns the id of the current cursor being used in the canvas. See `SetCursor`.

virtual int GetHeight()

Returns the height of the current drawing canvas in pixels.

virtual int GetHScroll(int& Shown, int& Top)

Get the status of the Horizontal Scroll bar. Returns 1 if the scroll bar is displayed, 0 if not. Returns in `Shown` and `Top` the current values of the scroll bar. See `SetVScroll` for a description of the meanings of parameters.

virtual int GetVScroll(int& Shown, int& Top)

Get the status of the Vertical Scroll bar. See `GetHScroll` for details.

virtual int GetWidth()

Returns the width of the current drawing canvas in pixels. This is either the initial size of the window, or the size after the user has resized the window.

void SetCursor(VCursor id)

This method sets the cursor displayed while the mouse in in the current canvas area. The default cursor is the standard arrow cursor used on most host platforms. You can change the cursor displayed within the canvas area only by calling this method.

The cursors currently supported include:

VC_Arrow The standard arrow cursor.

VC_CenterArrow An upward point arrow.

VC_CrossHair A cross hair cursor.

VC_EWArrows Double ended horizontal arrows (EastWest).

VC_Hand A hand with a pointing finger (NOT ON WINDOWS).

VC_IBar An I bar cursor.

VC_Icon A cursor representing an icon.

VC_NSArrows Double ended vertical arrows (NorthSouth).

VC_Pencil A pencil (NOT ON WINDOWS).

VC_Question A question mark cursor (NOT ON WINDOWS).

VC_Sizer The cursor used for sizing windows.

VC_Wait A cursor that symbolizes waiting, usually an hour glass.

VC_X An X shaped cursor (NOT ON WINDOWS).

void SetWidthHeight(int width, int height)

This will set the size of the drawing canvas to `height` and `width` in pixels. It will also cause a `Resize` event message to be sent to the window.

virtual void SetHScroll(int Shown, int Top)

Set the horizontal scroll bar. See `SetVScroll` for a description of the parameters.

virtual void SetVScroll(int Shown, int Top)

Set the vertical scroll bar. The `Shown` parameter is a value from 0 to 100, and represents the percent of the scroll bar shows of the view in the canvas. For example, the canvas might be displaying text from a file. If the file was 100 lines long, and the window could show 20 lines, then the value of `Shown` would be 20, meaning that the canvas is showing 20 percent of the file. As the size of the data viewed in the canvas changes, your program should change the scroll bar to corresponding values.

The `Top` parameter represents where the top of the scroll indicator should be placed. For example, if the first line displayed in the canvas of a 100 line file was line 40, then `Top` should be 40, representing 40 percent.

This model of a scroll bar can be mapped to all the underlying windowing systems supported by ***V***, but the visual appearance of the scroll bar will vary.

virtual void ShowHScroll(int OnOrOff)

virtual void ShowVScroll(int OnOrOff)

When a canvas is first displayed, it will begin with both horizontal and scroll bars not shown by default. `ShowHScroll` and `ShowVScroll` can be used to selectively turn on and off the canvas scroll bars. When a scroll bar is turned off or on, the size of the canvas may changes, so you should also call `Resize` after you have set the scroll bars.

You must not call either of these methods until the canvas has actually been instantiated on the screen. This means if your application needs to start with scroll bars, you should have the calls to `ShowVScroll` and `ShowHScroll` in the code of your `vCmdWindow` class constructor (or other initialization code) *after* calling `vWindow::ShowWindow` in your class constructor.

Methods to Override

virtual void FontChanged(int vf)

Called when the font is changed. This usually means your application needs to resize the window and recalculate the number of rows and columns of text that can be displayed.

virtual void HPage(int Shown, int Top)

When the user moves the horizontal scroll bar, it generates an `HPage` event. It is up to your program to intercept (override) this method, and provide proper interpretation. This event usually is used for large movements. The meaning of `Shown` and `Top` represent the state of the scroll bar as set by the user. It is then up to your program to display the correct portion of the data shown in the canvas to correspond to these values. Your program uses `SetHScroll` to set appropriate values, and they are explained there. The `Shown` value supplied here will correspond to the value you program set for the scroll bar. The `Top` value should indicate the meaningful change as input by the user.

virtual void HScroll(int step)

This method is called when the user enters a single step command to the scroll bar. The value of `step` will be positive for right or negative for left scroll. These scrolls are usually interpreted as discreet steps - either a line or screenful at a time. It is up to your application to give an appropriate interpretation.

virtual void MouseDown(int x, int y, int button)

This is called when the user clicks a button on the mouse. The `x` and `y` indicates the position of the mouse in the canvas when the button was clicked. Mouse events in vCanvasPane are no-ops, and your subclass of `vCanvasPane` will need to handle proper interpretation of mouse clicks.

Sorry, but thanks to the Macintosh, handling of buttons is a bit nonportable. The `button` parameter will have a value of 1, 2, or 3. On X based systems, 1 is the left button, 2 is the middle button, and 3 is the right button. On Windows, 1 is the left button, and 3 is the right button. Thus, applications using the left and right buttons are portable from X to Windows. The single Macintosh button will return a value of 1.

If you intend your applications to port to all three platforms, you will have to account for the single Macintosh button. If you ignore X's middle button, then your applications can be directly portable from X to Windows.

virtual void MouseMotion(int x, int y)

This is called when the mouse moves while a button is *not* pressed, and gives the current `x` and `y` of the mouse. Most applications will ignore this information.

virtual void MouseMove(int x, int y, int button)

This is called when the mouse moves while a button is pressed, and gives the new `x`, `y`, and `button` of the mouse. Mouse events in vCanvasPane are no-ops, and your subclass needs to interpret them. Note that scaling applies to output only. The mouse events will provide unscaled coordinates, and it is up to your code to scale mouse coordinates appropriately. Mouse coordinate *do* have the translation added.

virtual void MouseUp(int x, int y, int button)

This is called when the user releases the mouse button, and gives the final location of the mouse. Mouse events in vCanvasPane are no-ops, and your subclass needs to interpret them.

virtual void Redraw(int x, int y, int width, int height)

`Redraw` is called when the canvas needs to be redrawn. The first redraw is generated when the canvas is first created. Other redraws are generated when the canvas is covered or uncovered by another window, and means the contents of the canvas must be repainted. The `vCanvasPane` does not know how to repaint the contents of the canvas, so you must override this method to be able to keep the canvas painted.

The parameters of `Redraw` represent the rectangular area that needs to be repainted. This areas is not always the whole canvas, and it is possible that many `Redraw` events will be generated in a row as the user drags a covering window off the canvas.

The default `Redraw` in `vCanvasPane` is a no-op, and your subclass needs to override `Redraw`.

virtual void Resize(int newW, int newH)

A `Resize` event is generated when the user changes the size of the canvas using the resize window command provided by the host windowing system.

The default `Resize` in `vBaseGLCanvasPane` is a no-op, and your subclass needs to override `Redraw`.

virtual void VPage(int Shown, int Top)

See `HPage`.

virtual void VScroll(int step)

This method is called when the user enters a single step command to the vertical scroll bar. The value of `step` will be positive for down or negative for up scroll. These scrolls are usually interpreted as discreet steps - either a line or screenful at a time. It is up to your application to give an appropriate interpretation.

See Also

vTextCanvasPane

7.6 vTextEditor

A complete text editing canvas pane.

Synopsis

Header: `<v/vtexted.h>`

Class name: vTextEditor

Hierarchy: vCanvasPane → vTextCanvasPane → vTextEditor

Description

This class is a completely functional line oriented text editor. It can edit any file with lines less than 300 characters wide that use a linefeed, carriage return, or combination of those to mark the end of each line.

While you need to create your own class derived from `vTextEditor`, your class can be very minimal. You will need to provide some service methods for the parent `vCmdWindow`, such as methods to open, read, save, and close files. Other than actually working with the real text source and providing that source to `vTextEditor`, you can get a fully functional text editor with no additional work.

However, `vTextEditor` has been designed to allow you to extend and add functionality to the editor if you need to. The `vTextEditor` also sends messages that will allow you to place various status messages on a status bar if you wish. The hard stuff is done for you. You don't need to worry about mouse movements, scroll bars or scroll messages, updating the screen, handling keystrokes, or anything else associated with actual editing. The `vTextEditor` class takes care of all those details, and provides a standard editing interface.

The following steps are required to use `vTextEditor`. First, you create an instance of your derived class from your `vCmdWindow` class, something like this:

```
...

// The Text Editor Canvas
 vedCanvas = new vedTextEditor(this);
 AddPane(vedCanvas);
...

// Show Window
```

```
    ShowWindow();
    vedCanvas->ShowVScroll(1);  // Show Vert Scroll for vTextEditor

    ...
```

Your derived `vTextEditor` class should provide the methods needed for opening and reading the text file you want to edit. (Actually, you can edit any text source you wish.) `VTextEditor` doesn't actually read or write any text itself. It maintains an internal line buffer. (The default version of the internal buffer is essentially limited by the amount of memory your system can provide. The buffer methods can be overridden to provide totally unlimited file size, if you wish.) The idea is to have your application control where the text comes from, and then add it a line at a time to the `vTextEditor` buffer. You retrieve the text a line at a time when you want to save the edited text. Thus, your if your code is working with disk files, it can read the text a line at a time, and let `vTextEditor` worry about the buffering.

The following code shows how to add the contents of a text file to the `vTextEditor` buffer, and display it in the canvas for the first time. Calls to `vTextEditor` methods are marked with **.

```
//==================>>> vedTextEditor::ReadFile <<<===================
  int vedTextEditor::ReadFile(char* name)
  {
    const int maxBuff = 300;    // Line length
    char buff[maxBuff];

    if (!name || !*name)
        return 0;
    ifstream inFile(name);      // Open the file

    if (!inFile)
        return 0;               // file not there

    resetBuff();                // ** Tell vTextEditor to init buffer

    while (inFile.getline(buff,maxBuff))  // read file
      {
        if (!addLine(buff))     // ** Add the line to the buffer
          {
            ERROR_MESSAGE("File too big -- only partially read.");
            break;
          }
      }
    inFile.close();             // Close the file
    displayBuff();              // ** Now, display the buffer
    return 1;
  }
```

To load text into the editor buffer, you first call `resetBuff` to initialize the buffer, then add a line at a time with calls to `addLine`, and finally display the text by calling `displayBuff`.

When your are editing (e.g., the user enters a Close command), you retrieve the text from the `vTextEditor` buffer with calls to `getLine`.

Then, to use the editor, you pass keystrokes from the `KeyIn` method of your `vCmdWindow` to the `EditKeyIn` method of the `vTextEditor`. `EditKeyIn` interprets the conventional meanings of the arrow keys, etc., and lets you edit the text in the buffer. You will also probably implement other commands, such as Find, by using the `EditCommand` method.

`VTextEditor` also calls several methods to notify of text state changes, such as current line, insert or overtype, etc. You can receive these messages by overriding the default methods, and display appropriate information on a status bar.

While `vTextEditor` is very complete, there are some things missing. The major hole is cut and paste support. This will be added when cut and paste support is added to ***V***. There is also no real undo support. Maybe someday.

Constructor

vTextEditor(vBaseWindow* parent)

The `vTextEditor` constructor requires that you specify the parent `vCmdWindow`. Since you usually create the text editor object in your `vCmdWindow` object, this is easy. You will probably need to cast the `this` to a `vBaseWindow*`.

Utility Methods

resetBuff()

Before you load new text into the buffer, you must first call this method. It initializes the internal state of the text buffer.

virtual int addLine(char* line)

This method is called repeatedly to add lines to the text buffer. The default method is limited by the amount of memory available on the system, and this method return 0 when it runs out of memory.

Note that the entire text buffer package can be overridden if you need to provide unlimited file size handling. You should examine the source code for `vTextEditor` to determine the specifications of the methods you'd need to override.

virtual void displayBuff()

After you have added the complete file, call `displayBuff` to display the text in the window.

virtual int getLine(char* line, int maxChars, long lineNum)

virtual int getFirstLine(char* line, int maxChars)

virtual int getNextLine(char* line, int maxChars)

These are used to retrieve the edited text from the buffer. You can use `getFirstLine` with `getNextLine` for easy sequential retrieval, or `getLine` for specific lines. These methods return -1 when all lines have been recovered.

virtual int EditCommand(int id, long val)

This method provides a complete interface to the functions provided by `vTextEditor`. While the basic editing functions are also handled by `EditKeyIn`, `EditCommand` gives access to functions that typically are either usually invoked from a menu command (such as Find), or don't have a standard mapping to a functions key (such as lineGoto). If you want the functionality of these commands in your application, you will have to provide an appropriate menu or command pane item to support them.

Each function supported by `vTextEditor` has an associated id (symbolically defined in `v/vtexted.h`), each beginning with `ed`. Many of the functions also take an associated value. Many editors allow a repetition count to be specified with many commands. For example, it is sometimes useful to be able to specify a command to move right some specific number of characters. The `val` parameter can be used to specify a value as desired. The only function that really need a value other than 1 (or -1 in the case of directional movement commands) is `edLineGoto`.

`EditCommand` returns 1 if the command was executed successfully, 0 if the command was recognized, but not successful (the find fails, for example), and -1 if the command was not recognized as valid.

At the time this manual was written, the following commands are supported. Because `vTextEditor` is evolving, it is likely more commands will be added. Check the `v/vtexted.h` file for specification of new editor commands. In the following descriptions, the note "no val" means that the `val` parameter is not used. A notation of "+/-" means the sign of `val` indicates direction.

edBalMatch find matching paren (if val ¿ 1, up to val lines away, otherwise within a reasonable range)

edBufferBottom move to bottom of file (no val)

edCharDelete delete +/- val chars

edCharFoldCase swap case of +/- val letters

edCharInsert insert char val

edCharRight move +/- val chars right

edFind invoke TextEd's find dialog (no val)

edFindNext find next occurrence of prev (no val)

edLineBeginning move to line beginning (no val)

edLineDown move down +/- val lines in column

edLineDownBeg move down +/- val lines

edLineDelete delete +/- val lines

edLineDeleteFront delete to beginning of line (no val)

edLineDeleteToEnd delete to end of line (no val)

edLineEnd move to end of line (no val)

edLineGoto move cursor to line val

edLineOpen open val new blank lines

edScrollDown scroll +/- val lines without changing cursor

edVerify force repaint of screen (no val)

edWordRight move cursor +/- val words right

For a basic editor, the simplest way to use `EditCommand` is to use the `ed*` id's to define the associated menu items and controls, and then call `EditCommand` as the default case of the `switch` in the `WindowCommand` method of your `vCmdWindow`. Thus, you might have code that looks like this:

```
  ...
 static vMenu EditMenu[] = {
 ...
   {"Find", edFind, isSens,notChk,noKeyLbl,noKey,noSub},
   {"Find Next", edFindNext, isSens,notChk,noKeyLbl,noKey,noSub},
   {"Find Matching Paren", edBalMatch, isSens,notChk,
     noKeyLbl,noKey,noSub},
 ...
 };

  ...

//===========>>> vedCmdWindow::WindowCommand <<<====================
 void vedCmdWindow::WindowCommand(ItemVal id, ItemVal val,
      CmdType cType)
 {
   switch (id)
    {
     ...

      default:  // route unhandled commands through editor
       {
         if (vedCanvas->EditCommand(id, 1) < 0)
            vCmdWindow::WindowCommand(id, val, cType);
        break;
       }

    }
   ...
 }

//===================>>> vedCmdWindow::KeyIn <<<====================
  void vedCmdWindow::KeyIn(vKey keysym, unsigned int shift)
  {
    if (vedCanvas->EditKeyIn(keysym, shift) < 0)
        vCmdWindow::KeyIn(keysym, shift);
  }
```

virtual int EditKeyIn(vKey key, unsigned int shift)

This method is usually called from the `KeyIn` method of your derived `vCmdWindow` class. See the above code example.

The default implementation of `EditKeyIn` handles most of the standard keys, such as the arrow keys, the page keys, backspace, home, delete, insert, and end keys. It will also insert regular character keys into the text. It ignores function keys and non-printing control key values except tab and newline.

You can override this method to provide your own look and feel to the editor.

edState GetEdState()

void SetEdState()

`VTextEditor` maintains a state structure with relevant state information associated with various operating options of `vTextEditor`. It is defined in `v/vtexted.h`, and has the following fields:

```
    typedef struct edState
      {
        long changes,           // count of changes
             cmdCount;          // how many times to repeat command
        int
            findAtBeginning,    // leave find at beginning of pattern
```

```
        fixed_scroll,       // flag if using fixed scroll
        ins_mode,           // true if insert mode
        counter,            // counter for + insert
        echof,              // whether or not to echo action
        tabspc,             // tab spacing
        wraplm;             // right limit
} edState;
```

You can query and set the state with `GetEdState` and `SetEdState`.

long GetLines()

Returns the number of lines in the current buffer.

Methods to Override

virtual void ChangeLoc(long line, int col)

This method is called by `vTextEditor` whenever the current line or current column is changed. This information could be displayed on a status bar, for example.

virtual void ChangeInsMode(int IsInsMode)

This method is called by `vTextEditor` whenever the insert mode is changed. If `IsInsMode` is true, then the editor is in insert mode. Otherwise, it is in overtype mode. The editor starts in insert mode. This information could be displayed on a status bar, for example.

virtual void StatusMessage(char* Msg)

The editor will call this message with a non-critical message such as "Pattern Not Found" for certain operations. This information could be displayed on a status bar, for example.

virtual void ErrorMessage(char* Msg)

The editor will call this message with a critical error message such as "Bad parameter value" for certain operations. This information could be displayed in a warning dialog, for example.

See Also

vTextCanvasPane

7.7 vBaseGLCanvasPane

A specialized base class to support OpenGL graphics.

Synopsis

Header: `<v/vbglcnv.h>`

Class name: vBaseGLCanvasPane

Hierarchy: vPane → vBaseGLCanvasPane

Description

This is a specialized class to provide very basic support for the OpenGL graphics package. Unlike other ***V*** canvas panes, this class does not use a `vDC` class. Instead, it has a few features designed to support OpenGL.

This is a basic class. It does not provide many convenience methods to support OpenGL at a high level, but it does hide all the messy details of interfacing with the host GUI environment, and provides the first really easy way to generate sophisticated interfaces for OpenGL applications. A more sophisticated class called `vGLCanvasPane` that will provide a number of convenience operations is under development, but the base class is still very useful.

By following a standard convention to structure V/OpenGL code, it is relatively easy to generate applications. The details of this convention are explained in the tutorial section of this description.

See the section `vPane` for a general description of panes.

Constructor

vBaseGLCanvasPane(unsigned int vGLmode)

The `vBaseGLCanvasPane` constructor allows you to specify certain attributes of the visual used by OpenGL. The options, which can be ORed together, include:

vGL_Default Use the default visual, which includes `vGL_RGB` and `vGL_DoubleBuffer`. ***V*** will use this default if you don't provide a value to the constructor.

vGL_RGB This is the standard RGBA mode used by most OpenGL programs. The size of the RED, GREEN, and BLUE planes are maximized according to the capabilities of the host machine. An ALPHA plane is not included unless the `vGL_Alpha` property is also specified.

vGL_Alpha Used to include an APLHA plane. Not all machines support ALPHA planes.

vGL_Indexed Use indexed rather than RGB mode. ***V*** will attempt to maximize the usefulness of the palette. You should not specify both RGB and Indexed.

vGL_DoubleBuffer Use Double buffering if available. Single buffering is assumed if `vGL_DoubleBuffer` is not specified.

vGL_Stereo Use a Stereo buffer if available.

vGL_Stencil Use Stencil mode if available.

vGL_Accum Use accumulation buffers if available.

vGL_Depth Use Depth mode if available.

Not all of these attributes are available on all OpenGL implementations, and ***V*** will attempt to get a reasonable visual based on your specifications. For now, the `vGL_Default` mode works well for many OpenGL applications.

V supports only one visual per application, and the first `vBaseGLCanvasPane` created determines the attributes of the visual used.

Utility Methods

The following methods provide useful service without modification. Sometimes you will want to override some of these, but you will then usually call these methods from your derived class. Most of these methods are the equivalent of the normal ***V*** `vCanvasPane` class.

VCursor GetCursor()

Returns the id of the current cursor being used in the canvas. See `SetCursor`.

virtual int GetHeight()

Returns the height of the current drawing canvas in pixels.

virtual int GetHScroll(int& Shown, int& Top)

Get the status of the Horizontal Scroll bar. Returns 1 if the scroll bar is displayed, 0 if not. Returns in `Shown` and `Top` the current values of the scroll bar. See `SetVScroll` for a description of the meanings of parameters.

virtual int GetVScroll(int& Shown, int& Top)

Get the status of the Vertical Scroll bar. See `GetHScroll` for details.

virtual int GetWidth()

Returns the width of the current drawing canvas in pixels. This is either the initial size of the window, or the size after the user has resized the window.

void SetCursor(VCursor id)

This method sets the cursor displayed while the mouse in in the current canvas area. See the description of `vCanvasPane` for details.

void SetWidthHeight(int width, int height)

This will set the size of the drawing canvas to `height` and `width` in pixels. It will also cause a `Resize` event message to be sent to the window.

virtual void SetHScroll(int Shown, int Top)

Set the horizontal scroll bar See the description of `vCanvasPane` for details.

virtual void SetVScroll(int Shown, int Top)

Set the vertical scroll bar. See the description of `vCanvasPane` for details.

virtual void ShowHScroll(int OnOrOff)

virtual void ShowVScroll(int OnOrOff)

See the description of `vCanvasPane` for details.

Methods to Override

virtual void HPage(int Shown, int Top)

When the user moves the horizontal scroll bar, it generates an `HPage` event. See the description of `vCanvasPane` for details.

virtual void HScroll(int step)

This method is called when the user enters a single step command to the scroll bar. See the description of `vCanvasPane` for details.

virtual void MouseDown(int x, int y, int button)

This is called when the user clicks a button on the mouse.

It is important to remember that all mouse coordinates are in screen pixels, and use 0,0 as the upper left corner. You will probably have to map them to the actual coordinates in use by your OpenGL graphic.

See the description of `vCanvasPane` for details.

virtual void MouseMotion(int x, int y)

This is called when the mouse moves while a button is *not* pressed. See the description of `vCanvasPane` for details.

virtual void MouseMove(int x, int y, int button)

This is called when the mouse moves while a button is pressed. See the description of `vCanvasPane` for details.

virtual void MouseUp(int x, int y, int button)

This is called when the user releases the mouse button. See the description of `vCanvasPane` for details.

virtual void Redraw(int x, int y, int width, int height)

`Redraw` is called when the canvas needs to be redrawn. The first redraw is generated when the canvas is first created. Other redraws are generated when the canvas is covered or uncovered by another window, and means the contents of the canvas must be repainted. Normally, you will put a call to the code that redraws your OpenGl picture here.

The parameters of `Redraw` represent the rectangular area that needs to be repainted. This areas is not always the whole canvas, and it is possible that many `Redraw` events will be generated in a row as the user drags a covering window off the canvas.

The default `Redraw` in `vBaseGLCanvasPane` is a no-op, and your subclass needs to override `Redraw`.

virtual void Resize(int newW, int newH)

A `Resize` event is generated when the user changes the size of the canvas using the resize window command provided by the host windowing system.

The default `Resize` in `vBaseGLCanvasPane` is a no-op, and your subclass needs to override `Redraw`.

virtual void VPage(int Shown, int Top)

See the description of `vCanvasPane` for details.

virtual void VScroll(int step)

See the description of `vCanvasPane` for details.

Specific OpenGL methods

virtual void graphicsInit(void)

This method is called after the OpenGL drawing canvas has been created, and *must* be overridden by your code. You use this method to set up whatever you would usually do to initialize OpenGL. In practice, this is a very convenient way to get things started.

It is critical that you call the `graphicsInit` method in the base `vBaseGLCanvasPane` class *first*, then whatever OpenGL calls you need. See the example in the OpenGL tutorial section for more details.

void vglMakeCurrent(void)

This method should be called by your program before you call any OpenGL drawing code. Normally, this is called first thing in `Redraw`, or whatever code you use to draw with. It is essential to call this, and since it is cheap to call this for an already current drawing canvas, it is better to be safe.

virtual void vglFlush(void)

Call this method after you are finished calling OpenGL to draw a picture. It automatically handles the details of displaying your picture in the window, including double buffering and synchronization. It is normally found in your `Redraw` method.

virtual XVisualInfo* GetXVisualInfo()

This method is specific to X, and will return a pointer to the `XVisualInfo` structure currently being used. There will be an equivalent method available for MS-Windows.

Tutorial

A minimal V/OpenGL application will consist of a class derived from `vApp`, a class derived from `vCmdWindow`, and a canvas pane class derived from `vBaseGLCanvasPane`. Most of your drawing code will be in or called from your derived canvas pane.

Within that class, you will minimally need to override the `graphicsInit` method, and the `Redraw` method. The following code fragment, adapted directly from the example code in Mark J. Kilgard's book, *OpenGL, Programming for the X Window System*, shows how simple it can be to draw a picture. The full code can be found in the `opengl/shapes` directory in the ***V*** distribution.

```
  static int initDone = 0;

  ......

//==========>>> testGLCanvasPane::graphicsInit <<<=================
  void testGLCanvasPane::graphicsInit(void)
  {
```

```
    // Always call the superclass first!
    vBaseGLCanvasPane::graphicsInit();

    // Example from Mark Kilgard
    glEnable(GL_DEPTH_TEST);
    glClearDepth(1.0);
    glClearColor(0.0, 0.0, 0.0, 0.0);  /* clear to black */
    glMatrixMode(GL_PROJECTION);
    gluPerspective(40.0, 1.0, 10.0, 200.0);
    glMatrixMode(GL_MODELVIEW);
    glTranslatef(0.0, 0.0, -50.0);
    glRotatef(-58.0, 0.0, 1.0, 0.0);

    initDone = 1;
  }

//============>>> testGLCanvasPane::Spin <<<=======================
  void testGLCanvasPane::Spin()
  {
    // Called from the parent CmdWindow for animation
    vglMakeCurrent();              // Call this FIRST!
    glRotatef(2.5, 1.0, 0.0, 0.0);
    Redraw(0,0,0,0);
  }

//============>>> testGLCanvasPane::Redraw <<<=====================
  void testGLCanvasPane::Redraw(int x, int y, int w, int h)
  {
    static int inRedraw = 0;

    if (inRedraw || !initDone)  // Don't draw until initialized
  return;

    inRedraw = 1;               // Don't allow recursive redraws.

    vglMakeCurrent();           // Call this to make current

    // Code taken directly from Mark J. Kilgard's example
    // Draws 3 intersecting triangular planes
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glBegin(GL_POLYGON);
    glColor3f(0.0, 0.0, 0.0); glVertex3f(-10.0, -10.0, 0.0);
    glColor3f(0.7, 0.7, 0.7); glVertex3f(10.0, -10.0, 0.0);
    glColor3f(1.0, 1.0, 1.0); glVertex3f(-10.0, 10.0, 0.0);
    glEnd();

    glBegin(GL_POLYGON);
    glColor3f(1.0, 1.0, 0.0); glVertex3f(0.0, -10.0, -10.0);
    glColor3f(0.0, 1.0, 0.7); glVertex3f(0.0, -10.0, 10.0);
    glColor3f(0.0, 0.0, 1.0); glVertex3f(0.0, 5.0, -10.0);
    glEnd();

    glBegin(GL_POLYGON);
    glColor3f(1.0, 1.0, 0.0); glVertex3f(-10.0, 6.0, 4.0);
    glColor3f(1.0, 0.0, 1.0); glVertex3f(-10.0, 3.0, 4.0);
    glColor3f(0.0, 0.0, 1.0); glVertex3f(4.0, -9.0, -10.0);
    glColor3f(1.0, 0.0, 1.0); glVertex3f(4.0, -6.0, -10.0);
    glEnd();

    vglFlush();    // Call when done drawing to display

    inRedraw = 0;  // Not in here any more
  }
```

....

Note that this example includes a method called `Spin`. It is used to animate the intersecting planes. In a ***V*** OpenGL application, the easiest way to implement animation is with the timer. Create a timer in the Command Window class, and then call the animation code in the canvas in response to timer events. You should keep code to prevent recursive redraws if the timer events end up occurring faster than the picture can be rendered, which might happen for complex pictures or heavily loaded systems. See the example code in the `v/opengl` directory for a complete example of animation using the timer.

See Also

vCanvasPane

7.8 vColor

A class for handling and specifying colors.

Synopsis

Header: `<v/vcolor.h>`

Class name: vColor

Description

The ***V*** color model allows you to specify colors as an RGB value. The intensity of each primary color, red, green, and blue are specified as a value between 0 and 255. This allows you to specify up to 2^{24} colors. Just how many of all these colors you can see and how they will look on your display will depend on that display. Even so, you can probably count on (255,0,0) being something close to red on most displays. Given this 24 bit model, the `vColor` class allows you to define colors easily.

In order to make using colors somewhat easier, ***V*** has defined a standard array of 16 basic colors that you can access by including `v/vcolor.h>`. This array is called `vStdColors`. You index the array using the symbols `vC_Black`, `vC_Red`, `vC_DimRed`, `vC_Green`, `vC_DimGreen`, `vC_Blue`, `vC_DimBlue`, `vC_Yellow`, `vC_DimYellow`, `vC_Magenta`, `vC_DimMagenta`, `vC_Cyan`, `vC_DimCyan`, `vC_DarkGray`, `vC_MedGray`, and `vC_White`. For example, use the standard color `vStdColors[vC_Green]` to represent green. You can also get a `char` for the color by using the symbol to index the `char* vColorName[16]` array.

The file `<v/vcb2x4.h>` contains definitions for 8 color buttons in a 2 high by 4 wide frame. The file `<v/vcb2x8.h>` has a 2 by 8 frame of all 16 standard colors. You can specify the size of each button in the frame by defining `vC_Size`. The default is 8. You can also specify the location in a dialog of the color button frame by defining the symbols `vC_Frame,` `vC_RightOf,` and `vC_Below`. The ids of each button in the frame correspond to the color indexes, but with a `M` prefix (e.g., `M_Red` for `vC_Red`). See the example in `v/examp` for and example of using the standard color button frames.

Also note that unlike most other ***V*** objects, it makes perfect sense to assign and copy `vColor` values. Thus, assignment, copy constructor, and equality comparison operators are provided.

Constructor

vColor(unsigned int rd 0, unsigned int gr = 0, unsigned int bl = 0)

The class has been defined so you can easily initialize a color either by using its constructor directly, or indirectly via an array declaration. Each color has a red, green, and blue value in the range of 0 to 255.

```
// Declare Red via constructor
vColor btncolor(255, 0 , 0);   // Red

// Declare array with green and blue
vColor GreenAndBlue[2] =
  {
    (0, 255, 0),              // Green
    (0, 0, 255)               // Blue
  };
```

Utility Methods

BitsOfColor()

This method returns the number of bits used by the machine to display to represent color. A value of 8, for example, means the computer is using 8 bits to show the color.

ResetColor(unsigned int rd = 0, unsigned int gr = 0, unsigned int bl = 0)

ResetColor(vColor& c)

Like the `Set` method, this method will set all three values of the color at once. However, ***V*** tries to preserve entries in the system color palette or color map with `ResetColor`. You can also pass a `vColor` object.

Consider the following code excerpt:

```
vColor aColor;        // A V Color
vBrush aBrush;
int iy;

...

for (iy = 0 ; iy < 128 ; ++iy)
  {
    aColor.Set(iy,iy,iy);     // Set to shade of gray
    aBrush.SetColor(aColor);  // Set brush
    canvas.DrawLine(10,iy+100,200,iy+100);  // Draw line
  }
...
```

This example will use up 128 color map entries on some systems (X, for example). Once a system has run out of entries, ***V*** will draw in black or white. When these systems run out of new color map entries, the color drawn for new colors will be black or white.

```
vColor aColor;        // A V Color
vBrush aBrush;
int iy;

...

for (iy = 0 ; iy < 128 ; ++iy)
  {
```

```
    aColor.ResetColor(iy,iy,iy);     // Set to shade of gray
    aBrush.SetColor(aColor);  // Set brush
    canvas.DrawLine(10,iy+100,200,iy+100);  // Draw line
  }
...
```

This example accomplishes the same as the first, but does not use up color map entries. Instead, the entry used for `aColor` is reused to get better use of the color map. If your application will be working with a large number of colors that will vary, using `ResetColor` will minimize the number of color map accesses.

On some systems, and systems with a full 24 bits of color, `ResetColor` and `Set` work identically.

WARNING: If you intend to use `ResetColor` on a `vColor` object, then `ResetColor` is the only way you should change the color of that object. You should not use the color assignment operator, or `Set`. `ResetColor` needs to do some unconventional things internally to preserve color palette entries, and these can be incompatible with regular assignment or `Set`. You can, however, safely use such a `vColor` object with any other `vColor` object. For example:

```
vColor c1, c2;

c1.ResetColor(100,100,100);    // You can use c1 with others.
c2 = c1;                       // OK, but this = now makes c2
                               // incompatible with ResetColor.
c2.ResetColor(200,200,200);    // DON'T DO THIS
```

Set(unsigned int rd = 0, unsigned int gr = 0, unsigned int bl = 0)

Set all three values of the color at once.

void SetR(unsigned int rd = 0)

Set the Red value.

void SetG(unsigned int gr = 0)

Set the Green value.

void SetB(unsigned int bl = 0)

Set the Blue value.

unsigned int r()

Get the Red value.

unsigned int g()

Get the Green value.

unsigned int b()

Get the Blue value.

int operator ==

Compare two color objects for equality.

int operator !=

Compare two color objects for inequality.

Notes about color

The color model used by ***V*** attempts to hide most of the details for using color. However, for some applications you may end up confronting some of the sticky issues of color.

Most machines in use in 1996 will not support all 2^{24} colors that can be represented by the RGB color specification. Typically, they devote 8 or 16 bits to each pixel. This means that the 24-bit RGB colors must be mapped to the smaller 8-bit or 16-bit range. This mapping is usually accomplished by using a palette or colormap.

V tries to use the default system color palette provided by the machine it is running on. On some systems, such as X, it is possible to run out of entries in the color map. Others, like Windows, map colors not in the color palette to dithered colors. ***V*** provides two methods to help with this problem. First, `vColor::BitsOfColor()` tells you how many bits are used by the running system to represent color. The method `vColor::ResetColor(r,g,b)` can be used to change the value of a color without using up another entry in the system color map. For now, these methods should allow you to work with color with pretty good flexibility. Eventually, ***V*** may include more direct support for color palettes.

See Also

C_ColorButton, vCanvas

7.9 vDC

This is the base class that defines all the drawing methods provided by the various drawing canvases.

Synopsis

Header: `<v/vdc.h>`

Class name: vDC

Description

All drawing classes such as `vCanvasPaneDC` and `vPostScriptDC` are derived from this class. Each drawing class will support these methods as needed. Not all drawing classes have the same scale, and printer drawing canvases provide extra support for paging. Your code will not normally need to include `vdc.h`.

See the specific sections for details of drawing classes.

Utility Methods

virtual void BeginPage()

Supported by printer canvases. Call to specify a page is beginning. Bracket pages with `BeginPage` and `EndPage` calls.

virtual void BeginPrinting()

Required by printer canvases. Call to specify a document is beginning. You *must* bracket documents with `BeginPrinting` and `EndPrinting` calls. `BeginPrinting` includes an implicit call to `BeginPage`.

virtual void Clear()

Clear the canvas to the background color. No op on printers.

virtual void ClearRect(int x, int y, int width, int height)

Clear a rectangular area starting at x,y of height and width. No op on printers.

void CopyFromMemoryDC(vMemoryDC* memDC, int destX, int destY, int srcX = 0, int srcY = 0, int srcW = 0, int srcH = 0)

This method is used to copy the image contained in a `vMemoryDC` to another drawing canvas. The parameter `memDC` specifies the `vMemoryDC` object, and `destX` and `destY` specify where the image is to be copied into `this` drawing canvas (which will usually be 0,0). If you use the default values for `srcX=0`, `srcY=0`, `srcW=0`, and `srcH=0`, the entire source canvas will be copied.

Beginning with ***V*** release 1.13, `CopyFromMemoryDC` provides the extra parameters to specify an area of the source to copy. You can specify the source origin, and its width and height. The default values for these allow backward call and behavior compatibility.

One of the most useful uses of this is to draw both the canvas pane drawing canvas, and to a memory drawing canvas, and then use `CopyFromMemoryDC` to copy the memory canvas to the canvas pane for `Redraw` events.

virtual void DrawAttrText(int x, int y, char* text, const ChrAttr attr)

Draw text using the current font with specified attributes at given x, y.

`ChrAttr attr` is used to specify attributes to override some of the text drawing characteristics normally determined by the pen and font. Specifying `ChNormal` means the current pen and font will be used. `ChReverse` is used to specify the text should be drawn reversed or highlighted, using the current font and pen. You can also specify 16 different standard colors to override the pen color. You use `ORed` combinations the basic color attributes `ChRed`, `ChBlue`, and `ChGreen`. Most combinations are also provided as `ChYellow`, `ChCyan`, `ChMagenta`, `ChWhite`, and `ChGray`. These colors can be combined with `ChDimColor` can be used for half bright color combinations (or you can use `ChDimRed`, etc.). You can combine color attributes with `ChReverse`. Attributes such as boldface, size, and underlining are attributes of the font.

virtual void DrawColorPoints(int x, int y, int nPts, vColor* pts)

Draw an array of `nPts` `vColors` as points starting at x,y. This method is useful for drawing graphical images, and bypasses the need to set the pen or brush for each point. Typically, `DrawColorPoints` will be significantly faster than separate calls to `DrawPoint`.

virtual void DrawEllipse(int x, int y, int width, int height)

Draw an ellipse inside the bounding box specified by x, y, width, and height. The current Pen will be used to draw the shape, and the current Brush will be used to fill the shape.

virtual void DrawIcon(int x, int y, vIcon& icon)

A `vIcon` is drawn at x,y using the current Pen. Note that only the location of an icon is scaled. The icon will retain its original size.

virtual void DrawLine(int x, int y, int xend, int yend)

Draw a line from x,y to xend,yend. The current Pen will be used to draw the line.

virtual void DrawLines(vLine* lineList, int count)

Draws the `count` lines contained in the list `lineList`.

The current Pen will be used to draw the lines.

The type `vLine` is defined in `v_defs.h` as:

```
typedef struct vLine
  {
    short x, y, xend, yend;
  } vLine;
```

virtual void DrawLines(vPoint* points, int count)

Draws the `count` lines defined by the list of endpoints `points`. This is similar to drawing with a line list. The value of `count` must be 2 or greater. (New in version 1.19)

The current Pen will be used to draw the lines.

virtual void DrawPoint(int x, int y)

Draw a point at x,y using the current Pen.

virtual void DrawPoints(vPoint* pointList, int count)

Draws the `count` points contained in the list `pointList`.

The current Pen will be used to draw the points.

The type `vPoint` is defined in `v_defs.h` as:

```
typedef struct vPoint
  {
    short x, y;
  } vPoint;
```

virtual void DrawPolygon(int n, vPoint points[], int fillMode = vAlternate)

A closed polygon of n points is drawn. Note that the first and last element of the point list must specify the same point. The current Pen will be used to draw the shape, and the current Brush will be used to fill the shape.

The fillMode parameter specifies one of two alternative filling algorithms, `vAlternate` or `vWinding`. These algorithms correspond to the equivalent algorithms on the native platforms.

The type `vPoint` is defined in `v_defs.h` as:

```
typedef struct vPoint       // a point
  {
    short x, y;             // X version
  } vPoint;
```

virtual void DrawRoundedRectangle(int x, int y, int width, int height, int radius = 10)

Draw a rectangle with rounded corners at x,y of size width and height. The radius specifies the radius of the circle used to draw the corners. If a radius of less than 0 is specified, the radius of the corners will be ((width+height)/-2*radius) which gives a more or less reasonable look for various sized rectangles. The current Pen will be used to draw the shape, and the current Brush will be used to fill the shape.

virtual void DrawRectangle(int x, int y, int width, int height)

Draw a rectangle with square corners at x,y of size width and height. The current Pen will be used to draw the shape, and the current Brush will be used to fill the shape.

virtual void DrawRectangles(vRect* rectList, int count)

Draw a list of `count` `vRect` rectangles pointed to by the list `rectList`. The current Pen will be used to draw the rectangles, and the current Brush will be used to fill the rectangles.

The type `vRect` is defined in `v_defs.h` as:

```
typedef struct vRect
  {
    short x, y, w, h;
  } vRect;
```

virtual void DrawRubberLine(int x, int y, int xend, int yend)

Draw a rubber-band line from x, y to xend, yend. This method is most useful for showing lines while the mouse is down. By first drawing a rubber line, and then redrawing over the same line with `DrawRubberLine` causes the line to be erased. Thus, pairs of rubber lines can track mouse movement. The current Pen is used to determine line style.

virtual void DrawRubberEllipse(int x, int y, int width, int height)

Draw a rubber-band Ellipse. See DrawRubberLine.

virtual void DrawRubberPoint(int x, int y)

Draw a rubber-band point. See DrawRubberLine.

virtual void DrawRubberRectangle(int x, int y, int width, int height)

Draw a rubber-band rectangle. See DrawRubberLine.

virtual void DrawText(int x, int y, char* text)

Simple draw text at given x, y using the current font and current pen. Unlike icons and other ***V*** drawing objects, `x` and `y` represent the lower left corner of the first letter of the text. Using a `vSolid` pen results in the text being drawn in with the pen's color using the current background color. Using a `vTransparent` pen results in text in the current color, but just drawing the text over the current canvas colors. (See `vPen::SetStyle`.)

virtual void EndPage()

Supported by printer canvases. Call to specify a page is ending. Bracket pages with `BeginPage` and `EndPage` calls.

virtual void EndPrinting()

Supported by printer canvases. Call to specify a document is ending. Bracket documents with `BeginPrinting` and `EndPrinting` calls. `EndPrinting` includes an implicit call to `EndPage`.

virtual vBrush GetBrush()

Returns a copy of the current brush being used by the canvas.

virtual vFont GetFont()

Returns a copy of the current font of the drawing canvas.

virtual vBrush GetPen()

Returns a copy of the current pen being used by the canvas.

virtual int GetPhysHeight()

Returns the maximum physical y value supported by the drawing canvas. Especially useful for determining scaling for printers.

virtual int GetPhysWidth()

Returns the maximum physical x value supported by the drawing canvas. Especially useful for determining scaling for printers.

virtual void GetScale(int& mult, int& div)

Returns the scaling factors for the canvas. See `SetScale`.

void GetTranslate(int& x, int& y)

int GetTransX()

int GetTransY()

Returns the current x and y translation values.

virtual void SetBackground(vColor& color)

This sets the background of the drawing canvas to the specified color.

virtual void SetBrush(vBrush& brush)

This sets the brush used by the drawing canvas. Brushes are used for the filling methods such as `vDrawPolygon`. It is important to call `SetBrush` whenever you change any attributes of a brush used by a drawing canvas.

virtual void SetFont(vFont& vf)

Change the font associated with this canvas. The default method handles changing the font and calls the FontChanged method for the canvas pane.

virtual void SetPen(vPen& pen)

Sets the current pen of the canvas to pen. Pens are used to draw lines and the outlines of shapes. It is important to call `SetPen` whenever you change any attributes of a pen used by a drawing canvas.

virtual void SetScale(int mult, int div)

Sets the scaling factor. Each coordinate passed to the drawing canvas is first multiplied by mult and then divided by div. Thus, to scale by one third, set mult to 1 and div to 3. Many applications will never have to worry about scaling. Note that scaling applies to output only. The mouse events will provide unscaled coordinates, and it is up to your code to scale mouse coordinates appropriately.

void SetTranslate(int x, int y)

void SetTransX(int x)

void SetTransY(int y)

These methods set the internal translation used by the drawing canvas. Each coordinate sent to the various drawing methods (e.g., `DrawRectangle`) will be translated by these coordinates. This can be most useful when using the scroll bars to change which part of a drawing is visible on the canvas. Your application will have to handle proper mapping of mouse coordinates.

int TextHeight(int& ascent, int& descent)

This function returns the total height of the font `fontId`. The total height of the font is the sum of the `ascent` and `descent` heights of the font `fontId`. Each character ascends `ascent` pixels above the Y coordinate where it is being drawn, and `descent` pixels below the Y coordinate.

int TextWidth(char* str)

Returns the width in pixels or drawing points of the string `str` using the currently set font of the canvas.

7.10 vMemoryDC

A memory drawing canvas.

Synopsis

Header: `<v/vmemdc.h>`

Class name: vMemoryDC

Description

This drawing canvas can be used to draw to memory. Like all drawing canvases, the available methods are described in `vDC`. A very effective technique for using a memory canvas is to draw to both the screen canvas pane and a memory canvas during interactive drawing, and use the memory canvas to update the screen for `Redraw` events. This is especially useful if your application requires extensive computation to draw a screen.

Methods

vMemoryDC(int width, int height)

The constructor is used to construct a memory DC of the specified width and height. This can be anything you need. If you are using the memory DC to update the screen for `Redraw` events, then it should be initialized to be big enough to repaint whatever you will be drawing on the physical screen. The methods `vApp::ScreenWidth()` and `vApp::ScreenHeight()` can be used to obtain the maximum size of the physical screen.

The method `CopyFromMemoryDC` is used to copy the contents of a memory DC to another DC. This can be another memory DC, but will usually be a canvas pane DC.

7.11 vPen

A class to specify the pen used to draw lines and shapes.

Synopsis

Header: `<v/vpen.h>`

Class name: vPen

Description

Pens are used to draw lines and the outlines of shapes. Pens have several attributes, including color, width, and style.

Methods

vPen(unsigned int r = 0, unsigned int g = 0, unsigned int b = 0, int width = 1, int style = vSolid)

The constructor for a pen allows you to specify the pen's color, width, and style. The default will construct a solid black pen of width 1.

int operator ==, !=

You can use the operators `==` and `!=` for comparisons.

vColor GetColor()

This method returns the current color of the pen as a `vColor` object.

int GetStyle()

This method returns the current style of the pen.

void GetWidth()

This gets the width of the line the pen will draw.

void SetColor(vColor& c)

You can use this method to set the pen color by passing in a `vColor` object.

void SetStyle(int style)

This method is used to change the style of a pen. Styles include:

vSolid The pen draws a solid line.

vTransparent The pen is transparent. A transparent pen can be used to avoid drawing borders around shapes. When drawing text, a transparent pen draws the text over the existing background.

vDash The pen draws a dashed line.

vDot The pen draws a dotted line.

vDashDot The pen draws an alternating dash and dotted line.

void SetWidth(int width)

This sets the width of the line the pen will draw.

7.12 vPrintDC

A printer drawing canvas.

Synopsis

Header: `<v/vprintdc.h>`

Class name: vPrintDC

Description

This drawing canvas can be used to draw to a printer. Like all drawing canvases, the available methods are described in `vDC`. A very effective technique for combining a printer DC and a screen DC is to pass a pointer to either a `vCanvasPaneDC` or a `vPrintDC` to the code that draws the screen. The same code can then be used to draw or print.

To successfully use a `vPrintDC`, your code must obtain the physical size of the page in units using `GetPhysWidth` and `GetPhysHeight`. On paper, these represent 1/72 inch points, and correspond very closely, but not exactly, to a pixel on the screen.

You must bracket the printing with calls to `BeginPrinting` and `EndPrinting`. Use `BeginPage` and `EndPage` to control paging. Note that the width of text will not necessarily be the same on a `vCanvasPaneDC` and a `vPrintDC`, even for the same fonts. Also, the size of the paper represents the entire page. Most printers cannot actually print all the way to the edges of the paper, so you will usually use `vDC:SetTranslate` to leave some margins. (Don't forget to account for margins when you calculate what can fit on a page.)

The implementation of `vPrintDC` is somewhat platform dependent. For X, `vPrintDC` represents a PostScript printer, and is derived from the class `vPSPrintDC`. For Windows, `vPrintDC` is derived from the `vWinPrintDC` class. To get platform independent operation for your application, use `vPrintDC`. On Windows, you can also use the PostScript version directly if you want by using the `vPSPrintDC` class, but the program will not conform to standard Windows behavior.

Methods

void SetPrinter(vPrinter& printer)

This method is used to associate a `vPrinter` with a `vPrintDC`. By default, a `vPrintDC` represents standard 8.5x11 inch Letter paper printed in black and white in portrait orientation. You can use `vPrinter::Setup` to allow the user to change the attributes of the printer, then use `SetPrinter` to associate those attributes with the `vPrintDC`. Note: If you change the default printer attributes, you *must* call `SetPrinter` before doing any drawing to the DC.

Example

This is a simple example taken from the `VDraw` demo program. `Print` is called to print the current drawing. `Print` calls `vPrinter::Setup` to set the printer characteristics, and then calls `DrawShapes` with a pointer to the `vPrintDC`. `DrawShapes` is also called to repaint the screen using the `vCanvasPaneDC`. By carefully planning for both screen and printer drawing, your program can often share drawing code in this fashion.

```
//===================>>> myCanvasPane::Print <<<=================
  void myCanvasPane::Print()
  {
    // Print current picture

    vPrintDC pdc;              // create a vPrintDC object
    vPrinter printer;          // and a printer to set attributes

    printer.Setup("test.ps");  // setup the printer
    pdc.SetPrinter(printer);   // change to the printer we setup

    if (!pdc.BeginPrinting())  // call BeginPrinting first
        return;

    pdc.SetTranslate(36,36);   // Add 1/2" (36 * 1/72") margins

    DrawShapes(&pdc);          // Now, call shared drawing method

    pdc.EndPrinting();         // Finish printing
  }

//===================>>> myCanvasPane::DrawShapes <<<=================
  void myCanvasPane::DrawShapes(vDC* cp)
  {
    // Common code for drawing both on Screen and Printer
    ...
  }
```

7.13 vPrinter

A printer object, with a dialog to interactively set printer attributes.

Synopsis

Header: `<v/vprinter.h>`

Class name: vPrinter

Description

The `vPrintDC` class prints to a printer (or a file that will eventually be printed). Printers have such attributes as size of paper, page orientation, color capability, etc. By calling the `vPrinter::Setup` dialog before printing, the user will be given the option of setting various printer attributes.

The exact functionality of the `Setup` dialog will be platform dependent. By using the `vPrinter` class, you will get the behavior appropriate for the platform. If you want to use the `vPSPrintDC` class for PostScript support on Windows, you can use `vPSPrinter` directly.

You can use the various methods associated with a `vPrinter` to get printer attributes as needed to during drawing to the `vPrintDC`.

Methods

int GetCopies()

void SetCopies(int s)

Many printers support printing multiple copies of the same document. This attributes controls the number of copies printed. The `Setup` dialog will provide control of this *if* it is supported.

char* GetDocName()

Printer output may be directed to a file rather than the printer. If it is, this will return the name of the file the output will be sent to.

int GetPaper()

char* GetPaperName()

Printers can print a variety of papers. The user may be able to select which paper from the `Setup` dialog. The printers supported are defined in the `vprinter.h` header file (or the base class used by `vPrinter`).

int GetPortrait()

void SetPortrait(int p)

Many printers can print in either Portrait or Landscape orientation. This returns true if the printer will print in portrait.

int GetToFile()

void SetToFile(int f)

Printer output may be directed to a file rather than the printer. This returns true if the user selected the option to send output to a file.

int GetUseColors()

void SetUseColors(int c)

Printers can be either black and white, or color. This returns true if the printer supports colors. You can make a color printer print black and white by setting this to false.

int Setup(char* fn = 0)

This displays a modal dialog for the user to select desired printer characteristics. If a filename is supplied, that name will be used if the user selects print to file. If `Setup` returns false, you should abandon the print job. After you call `Setup`, you can then call `vPrintDC::SetPrinter` to associate the printer with the `vPrintDC`.

Example

See `vPrintDC` for an example of using `vPrinter::Setup`.

7.14 vTextCanvasPane

A class for drawing text on a canvas.

Synopsis

Header: `<v/vtextcnv.h>`

Class name: vTextCanvasPane

Hierarchy: vPane → vCanvasPane → vTextCanvasPane

Description

This class provides a complete scrolling text window. You can send text line by line to the window, and it will scroll the text up the screen in response to linefeed characters. You can also position the cursor, and selectively clear areas of the text screen or display text at specific locations. This class handles repainting the screen on `Redraw` events. In essence, the `vTextCanvasPane` class provides the functionality of a typical simple-minded text terminal.

New Methods

void ClearRow(const int row, const int col)

This clears to blanks row `row` of the screen from column `col` to the end of the line.

void ClearToEnd(const int row, const int col)

This clears to blanks from row `row` and column `col` to the end of the screen.

int GetCols()

Returns number of columns in current text canvas.

int GetRows()

Returns number of rows in current text canvas.

void GetRC(int& row, int& col)

Returns in `row` and `col` the current row and column of the text cursor.

void GotoRC(const int row ,const int row)

Moves the text cursor to `row,col`.

void DrawAttrText(const char* text, const ChrAttr attr)

Draws `text` starting at the current cursor location using text attribute `attr`. For more details, see `vDC::DrawAttrText`.

void DrawChar(const char chr, const ChrAttr attr)

Draws a single character `chr` at the current cursor location using text attribute `attr`. See `DrawAttrText` for more details.

void DrawText(const char* text)

Draws `text` starting at the current cursor location. The newline character `'\n'` will cause the cursor to move to the beginning of the next line, and the text to scroll if the cursor was on the last line.

void HideTextCursor(void)

This method will hide the text cursor.

void ShowTextCursor(void)

This method will redisplay the text cursor at the current row and column.

void ScrollText(const int count)

This will scroll the text in the text canvas up or down by `count` lines. There will be `count` blank lines created at the bottom or top of the screen.

void ResizeText(const int rows, const int cols)

This method handles resize events. You will want to override this to track the new number of rows and columns.

void TextMouseDown(int row, int col, int button)

This is called when the user clicks the mouse button down. It is called with the text row and column, and the button number.

void TextMouseUp(int row, int col, int button)

This is called when the user releases the mouse button. It is called with the text row and column, and the button number.

void TextMouseMove(int row, int col, int button)

This is called when the mouse moves. It is called with the text row and column, and the button number.

Derived Methods

virtual void Clear()

This clears the text canvas and resets the row and column to 0,0.

void FontChanged(int)

This is called when the font of the canvas changes. `FontChanged` calls `ResizeText`, so you probably won't have to deal with this event.

void Redraw(int x, int y, int width, int height)

Called when the screen needs to be redrawn. Normally, you won't have to override this class since the `vTextCanvasPane` superclass will handle redrawing what is in the window. Instead, you will usually just have to respond to the `FontChanged` and `ResizeText` events when the contents of the canvas will actually change.

Inherited Methods

virtual void HPage(int Shown, int Top)

virtual void HScroll(int step)

virtual void SetFont(int vf)

virtual void SetHScroll(int Shown, int Top)

virtual void SetVScroll(int Shown, int Top)

virtual void VPage(int Shown, int Top)

virtual void VScroll(int step)

See Also

vCanvasPane, vWindow

Chapter 8

Standard V Values

This chapter covers standard predefined values.

The classes and objects covered in this chapter include:

Predefined ItemVals A useful collection of predefined values. Most are useful for defining dialogs, buttons, and menus.

8.1 Predefined ItemVals

A useful collection of predefined values. Most are useful for defining dialogs, buttons, and menus.

Synopsis

Header: `<v/v_defs.h>`

Description

When defining dialogs, menus, and command bars, you are required to provide an id for each item. There are many common operations used in GUI designs, and ***V*** provides various predefined values for building your programs. The natural interpretation of most of these values should be obvious, and the descriptions are kept to a minimum. Most of the definitions describe the accepted practice for menu or button items with the given title. While these `ItemVals` can be used anywhere, some have "standard" usage.

Control Values

M_About Shows an informative message about current application.

M_All Select all.

M_Cancel Cancel. Usually used with a dialog. ***V*** will automatically reset dialog commands to their original state when a `M_Cancel` is selected from a `vDialog` descended object.

M_Clear Used to clear a screen.

M_Close Used to close a file. The user is usually prompted to save or ignore changes if any were made to the file. This is usually not used to close a menu.

M_Copy Copy the highlighted text or item, and save into the clipboard.

M_Cut Cut the highlighted text or item from the file, and usually save into the clipboard.

M_Delete Delete the selected item or text – usually does not copy into the clipboard.

M_Done Done with operation.

M_Edit Typically a menu bar button to pulldown an edit menu.

M_Exit Exit from the program – checking to see if files need to be saved, of course.

M_File Typically a menu bar button to pulldown a file menu.

M_Find Find a pattern.

M_FindAgain Find pattern again.

M_Font Typically a menu bar button to pulldown a font menu.

M_FontSelect Select a font. (This is different from the `M_Font` value in that `M_Font` is intended as a main menu bar item, while this one is for a pulldown menu.

M_Format Typically a menu bar button to pulldown a format menu, which allows the user to select formatting options.

M_Help Show help.

M_Insert Typically a menu bar button to pulldown an insert menu.

M_Line `M_Line` is one of a few of these values that gets special treatment by the system. It is required for defining line separators in menus.

M_New Used to create a new file.

M_No Answer No.

M_None Select none.

M_OK OK, accept operation or information. Causes return from dialog.

M_Open Used to open an existing file.

M_Options Typically a menu bar button to pulldown an options menu.

M_Paste Paste the contents of the clipboard into the insertion point of the current file or item.

M_Preferences Set preferences.

M_Print Print current file.

M_PrintPreview On screen preview how the current file would look if printed.

M_Replace Replace pattern.

M_Save Used to save current file in its current name.

M_SaveAs Save current file under new name.

M_Search Typically a menu bar button to pulldown a search menu.

M_SetDebug Set debug stuff.

M_Test Typically a menu bar button to pulldown a test menu.

M_Tools Typically a menu bar button to pulldown a tools menu.

M_UnDo Undo the last action.

M_View Typically a menu bar button to pulldown a view menu, which allows the user to select different views of the document.

M_Window Typically a menu bar button to pulldown a window menu, which lets the user select different windows.

M_Yes Answer Yes.

8.2 Version Values

A useful collection of predefined values to determine the version of V and the platform.

Synopsis

V defines several values useful for determining the revision of ***V***, and the platform ***V*** is compiled on.

Header: `<v/v_defs.h>`

Version Values

V_VersMajor The major version of ***V***, such as 1.

V_VersMinor The minor release of ***V***, such as 12.

V_Version A text string describing the version of ***V***, such as *V 1.12 - 8/4/96*.

V_VersionX Defined if the is the standard X version of ***V***.

V_VersionMotif Defined if the Motif version of ***V***.

V_VersionWindows Defined if the Windows version of ***V***.

V_VersionWin95 Defined if the Windows 95 version of ***V***.

V_VersionOS2 Defined for the OS2 version of ***V***.

Chapter 9

Utilities

This chapter covers ***V*** utility classes and functions. Whenever possible, any corresponding native utility has been used to implement these classes. For example, you will get the standard file interface dialog for Windows.

Since the Athena implementation has no corresponding native utilities, the ***V*** utility classes have been implemented for Athena using standard ***V*** classes as much as possible. Thus, the source code for the Athena version of these utilities provides an excellent example of ***V*** for study.

The classes and objects covered in this chapter include:

vDebugDialog Utility class to access debugging messages.

vFileSelect A utility class to select or set a file name.

vFontSelect A utility class to select or set a font object.

vNoticeDialog A utility class to display a message.

vReplyDialog A utility class to get a text reply from the user.

vTimer A class for getting timer events.

vYNReplyDialog A utility class to display a message, and get a Yes or No answer.

Utility Functions Several useful functions.

9.1 vDebugDialog

Utility class to access debugging messages.

Synopsis

Header: `<v/vdebug.h>`

Class name: vDebugDialog

Hierarchy: vModalDialog → vDebugDialog

Description

V provides built in debugging features. Most of the ***V*** classes contain debugging messages that are displayed on `stderr` or a special debugging information window. For Unix systems, `stderr` is usually the xterm window used to launch the ***V*** application. For other environments, the debugging window is system dependent.

Several categories of debugging messages have been defined by ***V***, and display of messages from different categories is controlled by the `vDebugDialog` class.

V provides several macros that can be used to insert debugging messages into your code. These are of the form `SysDebugN` for system code, and `UserDebugN` for your code. Display of these messages is controlled by the `vDEBUG` symbol, and the settings of the `vDebugDialog` class.

You define an error message using a `UserDebug` macro. Your message is a format string using the conventions of `sprintf`. You can have none to three values by using the corresponding `UserDebug` through `UserDebug3` macros. Each macro takes a debug type, a message, and any required values for the message format string. For example, `UserDebug(Misc,"myClass: %d\n", val)` will print the message "myClass: xx" when it is executed and the `Misc` debug message type is enabled.

If `vDEBUG` is *not* defined, your debugging messages will be null macros, and not occupy any code space. If `vDEBUG` is defined, then your messages will be conditionally displayed depending on their type.

By default, V starts with the `System` category `BadVals` on, and all three `User` categories on. Unix versions of ***V*** support a command line option that allows you to enable each option using the `-vDebug` command line switch. You include the switch `-vDebug` on the command line, followed by a single argument value made up of letters corresponding to the various debugging categories. If `-vDebug` is specified, all debugging categories except those specified in the value are turned off. The value for each category is listed in its header. For example, using the switch `-vDebug SUCDm` would enable debugging messages for both `System` and `User` constructors and destructors, as well as `System` mouse events. Note that the values are case sensitive.

Debugging Categories

Each of the following debug categories can be set or unset using the `vDebugDialog` class. These category names are to be used as the first argument to the `UserDebug` macro.

System (-vDebug S) These are the messages defined using the `SysDebug` macro. These messages can sometimes be useful to determine if you are using the classes properly. The constructor, destructor, and command events are often the most useful system debug messages. Turning this off will disable all system messages.

User (-vDebug U) These are the messages defined using the `UserDebug` macros. Turning this off will disable all user messages, while turning it on enables those user messages that have been enabled.

CmdEvents (-vDebug c) This category corresponds to command events, which include menu picks and dialog command actions.

MouseEvents (-vDebug m) This category corresponds to mouse events, such as a button click or a move.

WindowEvents (-vDebug w) This category corresponds to window events, such as a resize or redraw.

Build (-vDebug b) This category corresponds to actions taken to build a window, such as adding commands to a dialog.

Misc (-vDebug o) This is a catch all category used for miscellaneous system messages. The `o` vDebug stands for other. You should probably use a UserAppN category for your miscellaneous messages.

Text (-vDebug t) These messages are primarily used by the `vTextCanvasPane` class, and are useful for debugging text display.

BadVals (-vDebug v) These messages are generated when a bad parameter or illegal value is detected. These can be most useful.

Constructor (-vDebug C) These messages are displayed whenever a constructor for an object is called. These messages can be very useful for tracking object creation bugs. You should try to have `UserDebug(Constructor,"X::X constructor")` messages for all of your constructors, and a corresponding Destructor message.

Destructor (-vDebug D) Messages from an object destructor.

UserApp1, UserApp2, UserApp3 (-vDebug 123) These are provided to allow you up to three categories of your own debugging messages.

Example

To use the ***V*** debugging facilities, it is usually easiest to add a Debug command to a menu item – controlled by the `vDEBUG` symbol. Then add calls to `UserDebug` as needed in your code. This example shows how to define a Debug menu item, and then invoke the `vDebugDialog` to control debugging settings.

```
#include <v/vdebug.h>

    vMenu FileMenu[] =
      {
        ...
#ifdef vDEBUG
        {"-", M_Line, notSens,notChk,noKeyLbl,noKey,noSub},
        {"Debug", M_SetDebug,isSens,notChk,noKeyLbl,noKey,noSub},
#endif
        ...
      };

    ...
    case M_SetDebug:
     {
        vDebugDialog debug(this);    // instantiate
        UserDebug(Misc,"About to show Debug dialog.\n");
        debug.SetDebug();            // show the dialog
        break;
     }
    ...
```

9.2 vFileSelect

A utility class to select or set a file name.

Synopsis

Header: `<v/vfilesel.h>`

Class name: vFileSelect

Hierarchy: vModalDialog → vFileSelect

Description

This utility class provides a dialog interface for selecting filenames. It can be used either to select an input file name, or verify or change an output file name. This utility does not open or alter files – it simply constructs a legal file name for use in opening a file.

Methods

vFileSelect(vBaseWindow* win)

vFileSelect(vApp* app)

The `vFileSelect` constructor requires a pointer to a `vBaseWindow`, which includes all ***V*** windows and dialogs, or a pointer to the `vApp` object. You will usually pass the `this` to the constructor.

int FileSelect(const char* prompt, char* filename, const int maxLen, char filterList, int& filterIndex)**

int FileSelectSave(const char* prompt, char* filename, const int maxLen, char filterList, int& filterIndex)**

You provide a `prompt` for the user, such as "Open File." The user then uses the dialog to select or set a file name. `FileSelect` returns `True` if the user picked the OK button, and `False` if they used the Cancel button.

The filename will be filled in to the `filename` buffer of maximum length `maxLen`. The full path of the file will be included with the file name.

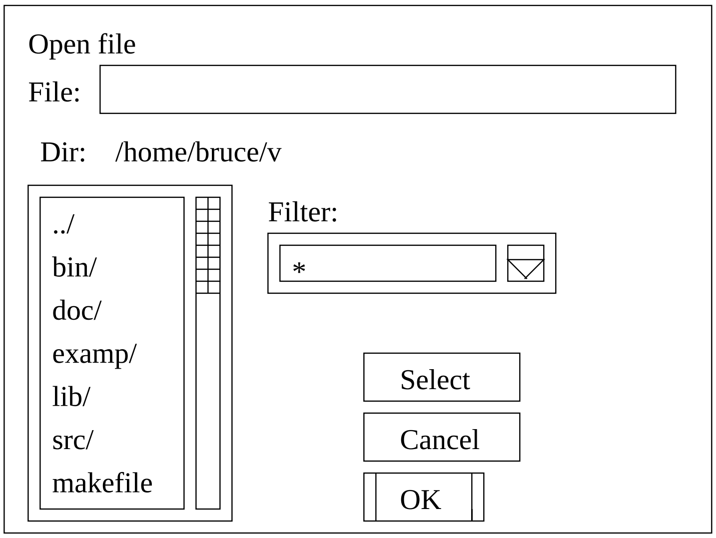
You can also provide a list of filter patterns to filter file extensions. If you don't provide a filter list, the default filter of "*" will be used. Each item in the filter list can include a list of file extensions separated by blanks. You can provide several filtering options. The first filter in the list will be the default. Only leading "*" wild cards are supported.

The `filterIndex` reference parameter is used to track which filter the user selected. After `FileSelect` returns, `filterIndex` will be set to the index of the filter list that the user last selected. For the best interface, you should remember this value for the next time you call `FileSelect` with the same filter list so that the user selected filter will be preserved.

You should use `FileSelect` to open a new or existing file. If the user is being asked to save a file (usually after picking a *Save As* menu choice), use the `FileSelectSave` method. On some platforms, there will be no difference between these two methods (X, for example). On other platforms (Windows, for example), different underlying system provided file dialogs are used. To your program, there will be no difference in functionality.

Example

The following is a simple example of using `vFileSelect`.




```
static char* filter[] =     // define a filter list
  {
    "*",                    // all files
    "*.txt",                // .txt files
    "*.c *.cpp *.h",        // C sources
    0
  };
static int filterIndex = 0;    // to track filter picked
char name[100];

vFileSelect fsel(this);     // instantiate

int oans = fsel.FileSelect("Open file",name,99,filter,filterIndex);

vNoticeDialog fsnote(this); // make an instance

if (oans && *name)
    (void)fsnote.Notice(name);
else
    (void)fsnote.Notice("No file name input.");
```

9.3 vFontSelect

A utility class to select or set a file name.

Synopsis

Header: `<v/vfontsel.h>`

Class name: vFontSelect

Hierarchy: vModalDialog $\rightarrow$ vFontSelect

Description

This class provides the `FontSelect` method to set the font being used. This class provides a platform independent way to change fonts. Depending on the platform, the user will be able to select many or most of the fonts available on the platform. On Windows, for example, the standard Windows font selection dialog is be used. On X, a relatively full set of fonts are available.

Methods

vFontSelect(vBaseWindow* win)

vFontSelect(vApp* app)

The `vFontSelect` constructor requires a pointer to a `vBaseWindow`, which includes all ***V*** windows and dialogs, or a pointer to the `vApp` object. You will usually pass the `this` to the constructor.

int FontSelect(vFont& font, const char* msg = "Select Font")

This method displays a dialog that lets the user select font characteristics. If possible, the native font selection dialog will be used (e.g., Windows). The font dialog will display the current characteristics of the `font` object, and change them upon successful return. A `false` return means the user selected Cancel, while a `true` return means the user finished the selection with an OK.

9.4 vNoticeDialog

A utility class to display a message.

Synopsis

Header: `<v/vnotice.h>`

Class name: vNoticeDialog

Hierarchy: vModalDialog → vNoticeDialog

Description

This simple utility class can be used to display a simple message to the user. The utility displays the message, and then waits for the user to enter to press OK.

New Methods

vNoticeDialog(vBaseWindow* win)

vNoticeDialog(vApp* app)

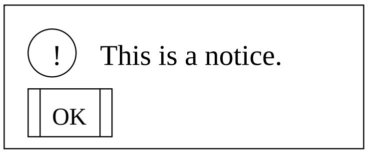
The `vNoticeDialog` constructor requires a pointer to a `vBaseWindow`, which includes all ***V*** windows and dialogs, or a pointer to the `vApp` object. You will usually pass the `this` to the constructor.

void Notice(const char* prompt)

You provide a prompt for the user. If the message contains '*backslash*n' newlines, it will be shown on multiple lines.

Example

The following is a simple example of using vNoticeDialog.




```
#include <v/vnotice.h>
...
vNoticeDialog note(this);   // instantiate a notice

(void)note.Notice("This is a notice.");
```

9.5 vReplyDialog

A utility class to get a text reply from the user.

Synopsis

Header: <v/vreply.h>

Class name: vReplyDialog

Hierarchy: vModalDialog → vReplyDialog

Description

This simple utility class can be used to obtain a text reply from the user. The utility displays a message, and then waits for the user to enter a reply into the reply field. The user completes the operation by pressing OK or Cancel.

New Methods

vReplyDialog(vBaseWindow* win)

vReplyDialog(vApp* app)

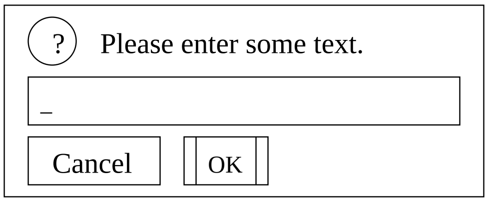
The vReplyDialog constructor requires a pointer to a vBaseWindow, which includes all ***V*** windows and dialogs, or a pointer to the vApp object. You will usually pass the this to the constructor.

int Reply(const char* prompt, char* reply, const int maxLen, char* dflt = "")

You provide a prompt for the user. The text the user enters will be returned to the buffer reply of maximum length maxLen. Reply will return the value M_OK or M_Cancel. Use dflt to provide a default reply.

Example

The following is a simple example of using vReplyDialog.




```
#include <v/vreply.h>
...
vReplyDialog rp(this);      // instantiate
char r[100];                // a buffer for reply

(void)rp.Reply("Please enter some text.",r,99);

vNoticeDialog note(this);   // instantiate a notice

if (*r)
    (void)note.Notice(r);
else
    (void)note.Notice("No text input.");
```

9.6 vTimer

A class for getting timer events.

Synopsis

Header: <v/vtimer.h>

Class name: vTimer

Hierarchy: vTimer

Description

This is a utility class that allows you to get events driven by the system timer. The accuracy and resolution of timers on various systems varies, so this should be used only to get events on a more or less regular basis. Use the C library time routines to get real clock time.

New Methods

vTimer

This constructs a timer object. The timer doesn't run until you start it with TimerSet. To make a timer useful, you can override the constructor to add a pointer to a window, and then use that pointer from within your TimerTick method to do something in that window: myTimer(vWindow* useWindow).

int TimerSet(long interval)

This starts the timer going. The timer will call your overridden TimerTick method approximately every interval milliseconds until you stop the timer. Most systems don't support an unlimited number of timers, and TimerSet will return 0 if it couldn't get a system timer.

void TimerStop()

Calling this stops the timer, but does not destruct it.

void TimerTick()

This method is called by the system every interval milliseconds (more or less). The way to use the timer is to derive your own class, and override the `TimerTick` method. Your method will be called according to the interval set for the timer. Note that you can't count on the accuracy of the timer interval.

9.7 vYNReplyDialog

A utility class to display a message, and get a Yes or No answer.

Synopsis

Header: `<v/vynreply.h>`

Class name: vYNReplyDialog

Hierarchy: vModalDialog → vYNReplyDialog

Description

This simple utility class can be used to display a simple message to the user. The utility displays the message, and then waits for the user to enter to press Yes, No, or Cancel.

New Methods

vYNReplyDialog(vBaseWindow* win)

vYNReplyDialog(vApp* app)

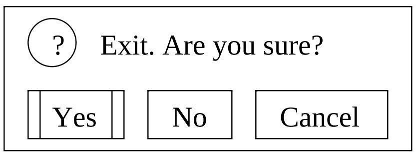
The `vYNReplyDialog` constructor requires a pointer to a `vBaseWindow`, which includes all ***V*** windows and dialogs, or a pointer to the `vApp` object. You will usually pass the `this` to the constructor.

int AskYN(const char* prompt)

You provide a `prompt` for the user. The user will then press the Yes, No, or Cancel buttons. `AskYN` returns a 1 if the user selected Yes, a 0 if they selected No, and a -1 if they selected Cancel.

Example

The following is a simple example of using `vYNReplyDialog`.



```
#include <v/vynreply.h>
...
vYNReplyDialog ynd(this);   // instantiate a notice

int ans = ynd.AskYN("Exit. Are you sure?);
if (ans == 1)
  exit(0);
```

9.8 Utility Functions

Several useful utility functions.

Synopsis

Header: `<v/vutil.h>`

Description

V provides several utility functions that can often help with software portability (and can just be useful). These are free subprograms – not a member of any specific class.

void ByteToStr(unsigned char b, char* str)

This will convert the unsigned char in `b` to a *Hex* character string in `str`. You need to make `str` big enough to hold the string.

void IntToStr(int intg, char* str)

This will convert the integer in `intg` to a character string in `str`. You need to make `str` big enough to hold the string.

void LongToStr(long intg, char* str)

This will convert the long integer in `intg` to a character string in `str`. You need to make `str` big enough to hold the string.

long StrToLong(char* str)

This will convert the character string in `str` into a long integer. You can cast to get ints.

void vBeep()

This utility routine will sound an audible beep.

void vGetcmdIdIndex(ItemVal cmdId, CommandObject *cmdObj)

Sometimes when you work with a CommandObject array to define a dialog, you need to access the elements of a particular item in the array. This is especially true for manipulating lists. This routine will return the index into a CommandObject array of an entry with the supplied `ItemVal cmdId`.

void vGetLocalTime(char* tm)

This will return a string representation of the current local time to the string `tm`. The format will be "HH:MM:SS AM". If you need a different format, you will need to use the C functions `time`, `localtime`, and `strftime` directly.

void vGetLocalDate(char* dt)

This will return a string representation of the current local date to the string `dt`. The format will be "MM/DD/YY". If you need a different format, you will need to use the C functions `time`, `localtime`, and `strftime` directly.

9.9 Utility Programs

bmp2vbm

The utility `bmp2vbm` converts a Window or OS/2 `.bmp` format bitmap file into a `.vbm` ***V*** icon bitmap format file. The `.vbm` file is then used with a `vIcon` object definition. The `bmp2vbm` utility will not convert all `.bmp` files. Specifically, it can't handle old format `.bmp` files, nor can it handle compressed `.bmp` files.

Windows has many tools to generate `.bmp` files. For X, the widely available tool `xv` can generate `.bmp` files from various source formats.

`Bmp2vbm` is a command line tool - run it from a Unix prompt, or from an `MSDOS` box on Windows. The command line format is: `bmp2vbm inputname outputname iconname`. You should specify only the base file names: `bmp2vbm` will automatically supply the `.bmp` and `.vbm` extension. The `iconname` specifies the name used to generate the date (e.g., `iconname_bits`).

9.10 More Goodies

The directory `v/icons` includes over 30 different monochrome icons in `.vbm` format suitable for building command pane tool bars. Most of these icons were derived from various Windows sources, and I would encourage their use for the standard functions they define. Some of these include creating a new file (new.vbm), opening an existing file (open.vbm), cut, copy, and paste (*.vbm), printing (print.vbm), and so on.

There is a demo program in the `v/icons` directory that can be compiled and used to see what all the icons look like. All the icons are 16 by 16 bits, and will match standard buttons in height on Windows. The height of standard buttons on X depends on the default system font.

As usual, contributions of other ***V*** icons is more than welcome. I hope to build up the icons directory to several hundred icons.

Appendix A

Tutorial C++ Source

The following is a complete tutorial ***V*** application. The source for this tutorial is available int the directory ~/v/tutor.

A.1 The Application

tutapp.cpp

```
//=======================================================================
//  tutapp.cpp:     Source file for tutorial V application
//
//      Copyright 1995, Bruce E. Wampler. All rights reserved.
//=======================================================================
//
// This file is used to define the main application object. There
// will be exactly one instance of the application object. You will
// usually derive you own app class from the vApp class. This file
// defines a sample tutApp class. The usual purpose of tutApp is to
// start the initial window, and act as a central controller for
// your application. Rather than reading this file sequentially,
// you should skip to the end and read the comments surrounding the
// AppMain function.


//  Files required for tutorial minimal application:
//      tutapp.h:       Header for the min app
//      tutapp.cpp:     Source code for min app
//      tcmdwin.h:      Header code for sample command window
//      tcmdwin.cpp:    Source code for sample command window
//      tdialog.h:      Header for sample modeless dialog
//      tdialog.cpp:    Source for sample modeless dialog
//      tmodal.h:       Header for sample modal dialog
//      tmodal.cpp:     Source for sample modal dialog
//

// First #include header files we need to use.

#include "tutapp.h"     // our header file

//========================>>> tutApp::NewAppWin <<<======================
  vWindow* tutApp::NewAppWin(vWindow* win, char* name, int w, int h,
    vAppWinInfo* winInfo)
  {
    // This version of NewAppWin is provided with the information
    // required to name and size a window.
```

```
    //
    // Typically, this method would get a file name or other information
    // needed to setup the AppWinInfo class  specifically for the
    // application. Thus, each open window usually represents a view of
    // a file or data object.

    vWindow* thisWin = win;             // local copy to use
    vAppWinInfo* awinfo = winInfo;
    char *myname = name;                // local copy

    if (!*name)
        myname = "Example";            // make up a name

    // The UserDebug macros are useful for tracking what is going on.
    // This shows we're building a window.

    UserDebug1(Build,"tutApp::NewAppWin(%s)\n",myname);

    // You may instantiate an instance of the window outside of
    // NewAppWin, or allow NewAppWin to create the instance.

    if (!thisWin)       // Didn't provide a window, so create one.
        thisWin = new tCmdWindow(myname, w, h);

    // The vAppWinInfo class is meant to serve as a database used by the
    // tutApp controller. If you use this feature, you will probably
    // derive your own myAppWinInfo class from vAppWinInfo. The instance
    // of vAppWinInfo created here will be automatically deleted when
    // this window instance is closed through CloseAppWin.

    if (!awinfo)        // Did caller provide an appinfo?
        awinfo = new vAppWinInfo(myname);

    // After you have created an instance of the window and an instance
    // of the AppWinInfo, you MUST call the base vApp::NewAppWin method.
    // You won't need to explicitly keep track of the pointer to
    // each new window -- unless it has to interact with other windows.
    // If that is the case, then you can use your derived vAppWinInfo
    // to coordinate the interaction.

    return vApp::NewAppWin(thisWin,name,w,h,awinfo);
  }

//===========================>>> tutApp::Exit <<<========================
  void tutApp::Exit(void)
  {
    // This can be called to close all windows. If the app needs to do
    // something special, it can. Otherwise, it can call the general
    // vApp::Exit method, which will perform appropriate calls the the
    // specialized tutApp::CloseAppWin.

    UserDebug(Build,"tutApp::Exit()\n");

    vApp::Exit();       // easy default behavior
  }

//======================>>> tutApp::CloseAppWin <<<======================
  void tutApp::CloseAppWin(vWindow* win)
  {
    // This will be called BEFORE a window has been unregistered or
    // closed. The app can do whatever it needs to to close down the
    // data associated with this window. (It is invoked explicitly by
    // you in response to a Close menu pick, for example, or when the
    // user clicks the close button. It can also be called by vApp::Exit().
    // After this method cleans up, it can then call the superclass
```

```
    // vApp::CloseAppWin to unregister and close this window. Note that
    // the win gives a handle that can be used with vApp::getAppWinInfo
    // to retrieve the AppWinInfo class.

    UserDebug(Build,"tutApp::CloseAppWin()\n");

    // Code to handle close of window (such as saving/closing
    // a file) would go here...

    vApp::CloseAppWin(win);   // Unregister and close the window.
  }

//=====================>>> tutApp::AppCommand <<<=========================
  void tutApp::AppCommand(vWindow* win, ItemVal id, ItemVal val,
    CmdType cType)
  {
    // Any commands not processed by the window WindowCommand
    // method will be passed to here for default treatment.

    UserDebug1(Build,"tutApp::AppCmd(ID: %d)\n",id);
    vApp::AppCommand(win, id, val, cType);
  }

//=======================>>> tutApp::KeyIn <<<===========================
  void tutApp::KeyIn(vWindow* win, vKey key, unsigned int shift)
  {
    // Any key strokes not processed by the window will be passed
    // along to here for default treatment.

    vApp::KeyIn(win, key, shift);
  }

//=========================================================================
// Remember that any static references to an object are constructed by
// the C++ startup code before main or any other functions are called.
// Thus, the constructor for tutApp (and thus vApp) is invoked before
// anything else happens. This enables V to perform whatever
// initializations are required by the host GUI system - and frees you
// from having to worry about the typical gory details. All this means
// that EVERY V application needs a static instance of the tutApp to
// get things rolling. Note that the global variable theApp is set to
// point to this instance, and is the easiest way to access the vApp
// and tutApp methods (e.g., theApp->Exit()).
//=========================================================================

  static tutApp tut_App("TutorApp");  // The single instance of the app

//==========================>>> AppMain <<<==============================
  int AppMain(int argc, char** argv)
  {
    // The V framework defines the instance of main. After some
    // processing of command line arguments, AppMain is called with
    // cleaned up command line arguments. Note that at this time, no
    // windows have been defined. Normally, AppMain is the place to
    // start up the first window. You can perform any initialization you
    // need to do here.

    (void) theApp->NewAppWin(0, "Tutorial V Example", 350, 100, 0);

    // At this point, the window is up, and all events are being
    // routed through its methods.

    // We MUST return 0 if the status is OK at this point.
    return 0;
  }
```

tutapp.h

```
//=======================================================================
//      tutapp.h:  Header file for tutorial V application
//
//      Copyright 1995, Bruce E. Wampler. All rights reserved.
//=======================================================================

#ifndef TUTAPP_H                // Standard technique for avoiding
#define TUTAPP_H                // problems with multiple #includes

#ifdef vDEBUG
#include <v/vdebug.h>
#endif

#include <v/vapp.h>     // We are derived from vApp
#include <v/vawinfo.h>  // Need for app info

#include "tcmdwin.h"            // we use our tCmdWindow class

    class tutApp : public vApp
      {
        friend int AppMain(int, char**); // allow AppMain access

      public:           //---------------------------------------- public

        tutApp(char* name) : vApp(name) {}      // just call vApp
        virtual ~tutApp() {}

        // Routines from vApp that are normally overridden

        virtual vWindow* NewAppWin(vWindow* win, char* name, int w, int h,
           vAppWinInfo* winInfo);
        virtual void Exit(void);
        virtual void CloseAppWin(vWindow* win);
        virtual void AppCommand(vWindow* win, ItemVal id,
               ItemVal val, CmdType cType);
        virtual void KeyIn(vWindow* win, vKey key,
            unsigned int shift);

        // New routines for this particular app go here

      protected:        //------------------------------------- protected

      private:          //--------------------------------------- private

      };
#endif
```

A.2 The Command Window

tcmdwin.cpp

```
//=======================================================================
//  tcmdwin.cpp:     Source file for tutorial cmdwin class
//
//      Copyright 1995, Bruce E. Wampler. All rights reserved.
//=======================================================================

// This file contains the source code for a typical command window
// derived from the vCmdWindow class. It will contain the definitions
// of the menu bar and command and status bars. It represents the main
// interaction point with the user.
```

```
//
// We start out with the #includes needed to define this class plus
// any V utility dialogs such as vNotice we use.

#include <v/vnotice.h>  // so we can use notice
#include <v/vkeys.h>    // to map keys
#include <v/vutil.h>    // for utilities
#include <v/vfilesel.h> // for file select

#include "tcmdwin.h"            // our header file

// Now, we define static arrays for the menus, command bars, and
// status bars used by this window. This consists of defining the
// constants needed for IDs, followed by the static declarations of
// the menu and command arrays. Note that V predefines quite a few
// standard IDs which you can use instead of defining your own.

  // Start ID defines for the main window at 100

  const ItemVal m_CheckMe = 100;  // for CheckMe command
  const ItemVal m_CopySens = 101; // for Set Copy Sensitive
  const ItemVal m_Dialog = 102;   // to pop up the dialog
  const ItemVal m_ModalDialog = 103; // for modal dialog
  const ItemVal m_Clear = 104;    // Clear screen

// Now, the static declarations of the menu arrays. You first define
// the pulldown menus, one for each main menu bar label.

    static vMenu FileMenu[] =    // Items for File menu
      {
        {"New",M_New,isSens,notChk,noKeyLbl,noKey,noSub},
        {"Open",M_Open,isSens,notChk,noKeyLbl,noKey,noSub},
        {"Save",M_Save,notSens,notChk,noKeyLbl,noKey,noSub},
        {"Save As",M_SaveAs,notSens,notChk,noKeyLbl,noKey,noSub},
#ifdef vDEBUG                   // Defines V debug code
        {"-",M_Line,notSens,notChk,noKeyLbl,noKey,noSub},
        {"Debug",M_SetDebug,isSens,notChk,noKeyLbl,noKey,noSub},
#endif
        {"-",M_Line,notSens,notChk,noKeyLbl,noKey,noSub},
        {"Exit",M_Exit,isSens,notChk,noKeyLbl,noKey,noSub},
        {NULL}
      };

    static vMenu EditMenu[] =    // Items for Edit menu
      {
        {"Cut",M_Cut,notSens,notChk,noKeyLbl,noKey,noSub},
        {"Copy",M_Copy,notSens,notChk,noKeyLbl,noKey,noSub},
        {"Paste",M_Paste,notSens,notChk,noKeyLbl,noKey,noSub},
        {NULL}
      };

    static vMenu TestMenu[] =   // Items for Test menu
      {
        {"CheckMe",m_CheckMe,isSens,notChk,noKeyLbl,
            noKey,noSub},
        {"Copy Sensitive",m_CopySens,isSens,notChk,noKeyLbl,
            noKey,noSub},
        {"Dialog",m_Dialog,isSens,notChk,noKeyLbl,
            noKey,noSub},
        {"Modal Dialog",m_ModalDialog,isSens,notChk,noKeyLbl,
            noKey,noSub},
        {NULL}
      };

    // Now, define the items on the menu bar
```

```
    vMenu StandardMenu[] =     // The menu bar with three items
      {
        {"File",M_File,isSens,notUsed,notUsed,noKey,&FileMenu[0]},
        {"Edit",M_Edit,isSens,notUsed,notUsed,noKey,&EditMenu[0]},
        {"Test",M_Test,isSens,notUsed,notUsed,noKey,&TestMenu[0]},
        {NULL}
      };

// We now define a command bar. Command bars are optional, and there
// may be more than one. You can place any CommandObject you want on a
// command bar.

    static CommandObject CommandBar[] =  // A simple command bar
      {
        {C_Label,999,0 ,"Command Bar",NoList,CA_None,
            isSens,NoFrame,0,0},
        {C_Button,M_Copy,M_Copy,"Copy",NoList,CA_None,
            notSens,NoFrame,0,0},
        {C_Button,m_Dialog,m_Dialog,"Dialog",NoList,CA_None,
            isSens,NoFrame,0,0},
        {C_Button,m_Clear,m_Clear,"Clear",NoList,CA_None,
            isSens,NoFrame,0,0},
        {C_Button,M_Exit,M_Exit,"Exit",NoList,CA_None,
            isSens,NoFrame,0,0},
        {C_EndOfList,0,0,0,0,CA_None,0,0,0}  // This ends list
      };

// Sometimes it is easier to define IDs near the definition of
// the dialog or status bar using them.

  const ItemVal m_cmdMsg = 110;
  const ItemVal m_cmdCount = 111;
  const ItemVal m_keyMsg = 112;
  const ItemVal m_keyVal = 113;

    static vStatus StatBar[] =    // Define a simple status bar
      {
        {"Commands issued: ",m_cmdMsg,CA_NoBorder,isSens,0},
        {" ",m_cmdCount,CA_None,isSens,0},
        {"Last keypress: ",m_keyMsg,CA_NoBorder,isSens,0},
        {"   ",m_keyVal,CA_None,isSens,0},
        {0,0,0,0,0}           // This ends list
      };

    static int copy_sens = 0;   // local for tracking sensitive

//======================>>> tCmdWindow::tCmdWindow <<<====================
  tCmdWindow::tCmdWindow(char* name, int width, int height) :
    vCmdWindow(name, width)
  {
    // This is the constructor for tCmdWindow.

    UserDebug1(Constructor,"tCmdWindow::tCmdWindow(%s) Constructor\n",name)

    // The "Standard" window will consist of a menubar, a canvas, an
    // optional button bar, and an optional status bar.
    //
    // First, create and add the proper panes to the CmdWindow
    // Note: there must be a corresponding delete in the destructor

    // Create and add the standard Menu Bar to this window
    myMenu = new vMenuPane(StandardMenu);
    AddPane(myMenu);
```

```
    // Create and add our Canvas pane to this window
    myCanvas = new tCanvasPane;
    AddPane(myCanvas);

    // Create and add the command pane to this window
    myCmdPane = new vCommandPane(CommandBar);
    AddPane(myCmdPane);

    // Create and add the Status Bar to this window
    myStatus = new vStatusPane(StatBar);
    AddPane(myStatus);

    // In the V model, a window may have dialogs. Each dialog used
    // by a window must have an instance pointer. The easiest way
    // to create dialogs is to construct each one using a new here
    // which only defines the dialog - you need to use its
    // ShowDialog method at the appropriate time to display it).
    // You delete dialogs in the destructor for this window.
    //
    // Now, create whatever dialogs this app defines:
    // instances of tDialog and tModalDialog

    sampleDialog = new tDialog(this);
    sampleModalDialog = new tModalDialog(this);

    // FINALLY, after all the panes have been constructed and
    // added, we must show the window!

    ShowWindow();
  }

//====================>>> tCmdWindow::~tCmdWindow <<<====================
  tCmdWindow::~tCmdWindow()
  {
    UserDebug(Destructor,"tCmdWindow::~tCmdWindow() destructor\n")

    // Now put a delete for each new in the constructor.

    delete myMenu;
    delete myCanvas;
    delete myStatus;
    delete myCmdPane;
    delete sampleDialog;
    delete sampleModalDialog;
  }

//=======================>>> tCmdWindow::KeyIn <<<=======================
  void tCmdWindow::KeyIn(vKey keysym, unsigned int shift)
  {
    // Keystrokes are routed to this window. This example code shows very
    // simple processing of keystrokes, and how to update the m_keyVal
    // field of the status bar.

    static char ctrl[] = "^X ";
    static char chr[] = " X ";

    if (VK_IsModifier(keysym))
        SetString(m_keyVal, "mod");     // change status bar
    else if (keysym < ' ')              // ctrl char
      {
        ctrl[1] = keysym + '@';
        SetString(m_keyVal, ctrl);      // change status bar
      }
    else if (keysym < 128)              // normal printable char
      {
```

```
        chr[1] = keysym;
        SetString(m_keyVal, chr);       // change status bar
      }
    else
        SetString(m_keyVal, "+++");     // change status bar
  }

//===================>>> tCmdWindow::WindowCommand <<<===================
  void tCmdWindow::WindowCommand(ItemVal id, ItemVal val, CmdType cType)
  {
    // All commands generated from this window's menus and dialog bars
    // are routed through here.  The easiest way to handle commands is to
    // use a single, sometimes large switch. Each time you add a command
    // to a menu or command bar, add a case to the switch here. In this
    // example, we use the V Notice dialog to display entered commands.

    static int cmdCount = 0;    // Used for sample status update
    vNoticeDialog note(this);   // Used for default actions
    char buff[20];              // buffer for status bar

    ++cmdCount;                 // count commands that have been issued
    IntToStr(cmdCount,buff);    // Use V utility routine to get string
    SetString(m_cmdCount, buff);        // change status bar

    UserDebug1(CmdEvents,"tCmdWindow:WindowCommand(%d)\n",id)

    switch (id)                 // The main switch to handle commands
      {
        // File Menu commands ------------------------------------

        case M_New:             // For this example, we will open a
          {                     // new window using our NewAppWin.
            (void) theApp->NewAppWin(0,"",250,100);
            break;
          }

        case M_Open:            // This demos vFileSelect dialog
          {
            char name[100] = "";        // start out with null name
            vFileSelect fsel(this);     // an instance of vFileSelect
            int fI;                     // Filter index
            static char* filter[] = {   // Filter for file select
                "*", "*.txt", "*.c *.cpp *.h", 0 };

            // Show the file select dialog
            int ans = fsel.FileSelect("Open file",name,99,filter,fI);

            if (ans && *name)   // User picked a file name
              {
                SetTitle(name); // Set title of window to name
                note.Notice(name);  // Show the name
              }
            else                // Notify no name selected
                note.Notice("No file name selected.");
          }

        case M_Save:            // This would usually save a file
          {
            note.Notice("Save");
            break;
          }

        case M_SaveAs:          // Save to a specified name
          {
            note.Notice("Save As");
```

```
          break;
        }

#ifdef vDEBUG                   // Include debugging like this
        case M_SetDebug:
          {
            vDebugDialog debug(this);   // an instance of debug
            debug.SetDebug();           // dialog - let user set
            break;
          }
#endif

        case M_Exit:            // Standard exit command
          {                     // Invoke the standard app Exit
            theApp->Exit();     // to close all windows
            break;              // will never get here
          }

        // Edit Menu commands ------------------------------------
        case M_Cut:             // Standard items for Edit menu
          {
            note.Notice("Cut");
            break;
          }

        case M_Copy:
          {
            note.Notice("Copy");
            break;
          }

        case M_Paste:
          {
            note.Notice("Paste");
            break;
          }

        // Test Menu commands ------------------------------------
        case m_CheckMe:         // Demonstrate using a checked menu
          {
            ItemVal curval = GetValue(id); // Get current status
            SetValue(m_CheckMe,!curval,Checked); // Toggle check

            if (curval)                 // Change menu label
                SetString(m_CheckMe,"Check Me");
            else
                SetString(m_CheckMe,"UnChk Me");
            break;
          }

        case m_CopySens:        // Demo changing sensitivity
          {
            copy_sens = !copy_sens;     // toggle
            // This will change both menu and command button
            SetValue(M_Copy,copy_sens,Sensitive);
            break;
          }

        case m_Dialog:          // Invoke our dialog
          {
            if (!sampleDialog->IsDisplayed())   // not twice!
                sampleDialog->ShowDialog("Sample Modeless Dialog");
            break;
          }
```

```
        case m_ModalDialog:     // Invoke our modal dialog
          {
            ItemVal val, id;
            id = sampleModalDialog->ShowModalDialog("Sample Modal",val);
            // Now do something useful with id and val ...
            break;
          }

        case m_Clear:           // Clear the canvas
          {
            myCanvas->Clear();  // Invoke the canvas Clear
            break;
          }

        default:                // route unhandled commands up
          {                     // to superclass
            vCmdWindow::WindowCommand(id, val, cType);
            break;
          }
      }
  }
```

tcmdwin.h

```
//========================================================================
//      tcmdwin.h:  Header file for tutorial V command window
//
//      Copyright 1995, Bruce E. Wampler. All rights reserved.
//========================================================================
//
// Derive a window from the vCmdWindow class

#ifndef TCMDWIN_H
#define TCMDWIN_H

#include <v/vcmdwin.h>  // So we can use vCmdWindow
#include <v/vmenu.h>    // For the menu pane
#include <v/vstatusp.h> // For the status pane
#include <v/vcmdpane.h> // command pane

#ifdef vDEBUG
#include <v/vdebug.h>
#endif

#include "tdialog.h"    // user defined: tDialog
#include "tmodal.h"     // user defined: tModalDialog
#include "tcanvas.h"    // user defined: tCanvasPane

    class tCmdWindow : public vCmdWindow
      {
        friend int AppMain(int, char**);        // allow AppMain access

      public:           //---------------------------------------- public
        tCmdWindow(char*, int, int);    // Constructor with size
        virtual ~tCmdWindow();          // Destructor
        virtual void WindowCommand(ItemVal id,ItemVal val,CmdType cType);
        virtual void KeyIn(vKey keysym, unsigned int shift);

      protected:        //------------------------------------- protected

      private:          //--------------------------------------- private

        // Each user CmdWindow should conform to a "Standard" window,
        // which includes a menu bar, a canvas, an optional command bar,
```

```
        // and an optional status bar.

        vMenuPane* myMenu;              // For the menu bar
        tCanvasPane* myCanvas;          // For the canvas
        vStatusPane* myStatus;          // For the status bar
        vCommandPane* myCmdPane;        // for the command pane

        // Each user CmdWindow will probably have some dialogs and
        // subwindows. Declare pointers to each instance here.

        tDialog* sampleDialog;
        tModalDialog* sampleModalDialog;
      };
#endif
```

A.3 The Canvas

tcanvas.cpp

```
//=======================================================================
//      tcanvas.cpp - source for tutorial canvas
//
//      Copyright 1995, Bruce E. Wampler, All Rights Reserved.
//=======================================================================
//
// Each V application usually needs a canvas. In order to handle
// various events: mouse, redraw, resize, and scroll, you will need to
// derive your own canvas class. The base V vCanvasPane class can only
// draw -- it does not have any memory of what has been drawn on the
// screen (the vTextCanvasPane does handle redrawing, but is still
// limited). Thus, your class will usually be responsible for handling
// redrawing. This example is very simple. It lets the user draw
// lines - up to 200 - and will redraw the screen when it has been
// exposed.

// The example does not handle scrolling.

#include "tcanvas.h"            // include our header file

//====================>>> tCanvasPane::tCanvasPane <<<====================
  tCanvasPane::tCanvasPane()
  {
    // The constructor initializes our simple data structure.

    _mouseDown = 0; _nextpt = 0;
    _begx = -1; _begy = -1; _curx = -1; _cury = -1;
    _pt = new point[200];       // allocate only 200 lines
  }

//-===================>>> tCanvasPane::tCanvasPane <<<====================
  tCanvasPane::~tCanvasPane()
  {
    delete [] _pt;              // free the point array
  }

//======================>>> tCanvasPane::Clear <<<=======================
  void tCanvasPane::Clear()
  {
    vCanvasPane::Clear();       // clear the canvas
    _nextpt = 0;                // start over at 0
  }

// This example does not handle scrolling, but a derived canvas would
```

```
// be likely to. Thus, we've included the derived scrolling methods,
// but simply call the superclass method for default handling, which
// is essentially a no op.

//=====================>>> tCanvasPane::HPage <<<=======================
  void tCanvasPane::HPage(int shown, int top)
  {
    vCanvasPane::HPage(shown, top);
  }

//=====================>>> tCanvasPane::VPage <<<=======================
  void tCanvasPane::VPage(int shown, int top)
  {
    vCanvasPane::VPage(shown, top);
  }

//=====================>>> tCanvasPane::HScroll <<<=====================
  void tCanvasPane::HScroll(int step)
  {
    vCanvasPane::HScroll(step);
  }

//=====================>>> tCanvasPane::VScroll <<<=====================
  void tCanvasPane::VScroll(int step)
  {
    vCanvasPane::VScroll(step);
  }

//====================>>> tCanvasPane::MouseDown <<<====================
  void tCanvasPane::MouseDown(int X, int Y, int button)
  {
   // Mouse down means the user is starting a line. We don't care which
   // button was pressed. There is nothing to draw until the mouse moves.

    _mouseDown = 1;                    // track mouse button
    _pt[_nextpt].x = _begx = _curx = X; // starting point
    _pt[_nextpt].y = _begy = _cury = Y;
    if (++_nextpt >= 200)              // set next point and do a simple
        _nextpt = 0;                   // minded storage allocation
  }

//=====================>>> tCanvasPane::MouseMove <<<===================
  void tCanvasPane::MouseMove(int x, int y, int button)
  {
    // Mouse move means the user is drawing a line, so we have to draw
    // it on the screen. By drawing a Rubber Line, we can easily track
    // the user motions, and undraw the previous line.

    if (_begx != _curx || _begy != _cury) // Was there a previous line?
        DrawRubberLine(_begx, _begy, _curx, _cury);  // Undraw old line

    if (_begx != x || _begy != y)       // If we moved, draw new line
        DrawRubberLine(_begx, _begy, x, y);

    _curx = x; _cury = y;               // update positions
  }

//=======================>>> tCanvasPane::MouseUp <<<===================
  void tCanvasPane::MouseUp(int X, int Y, int button)
  {
    // Mouse up means the user has ended a line, so we need to draw
    // a permanent line and update the data base.

    _mouseDown = 0;                     // Mouse down now
    if (_begx != X || _begy != Y)       // We drew a line
```

```
        DrawLine(_begx, _begy, X, Y);   // So draw permanent version

    _pt[_nextpt].x = X; _pt[_nextpt].y = Y;  // End point

    if (++_nextpt >= 200)               // set next point and do a simple
        _nextpt = 0;                    // minded storage allocation

    _begx = -1; _begy = -1; _curx = -1; _cury = -1;  // for next line
  }

//========================>>> tCanvasPane::Redraw <<<=====================
  void tCanvasPane::Redraw(int x, int y, int w, int h)
  {
    // This is a simple Redraw that just redraws everything.
    // Often, that will be more than fast enough, but the input
    // parameters can be used to make a more intelligent redraw.

    for (int i = 0 ; i < _nextpt ; i += 2)
        DrawLine(_pt[i].x, _pt[i].y, _pt[i+1].x, _pt[i+1].y);
  }

//======================>>> tCanvasPane::Resize <<<=======================
  void tCanvasPane::Resize(int w, int h)
  {
    // We also don't handle resizing in this example.
    vCanvasPane::Resize(w,h);
  }
```

tcanvas.h

```
//========================================================================
//  tcanvas.h -- header file for tutorial canvas class
//
//      Copyright 1995, Bruce E. Wampler, All Rights Reserved.
//========================================================================

#ifndef TCANVAS_H
#define TCANVAS_H

#include <v/vcanvas.h>  // derive from vCanvasPane

    typedef struct point        // simple structure for points
      {
        int x; int y;
      } point;

    class tCanvasPane : public vCanvasPane
      {
      public:           //--------------------------------------- public
        tCanvasPane();
        virtual ~tCanvasPane();

        // Windows
        virtual void Clear();

        // Scrolling
        virtual void HPage(int, int);
        virtual void VPage(int, int);
        virtual void HScroll(int);
        virtual void VScroll(int);

        // Events
        virtual void MouseDown(int, int, int);
        virtual void MouseUp(int, int, int);
```

```
        virtual void MouseMove(int, int, int);
        virtual void Redraw(int, int, int, int); // Expose/redraw event
        virtual void Resize(int, int);          // Resize event

      protected:        //------------------------------------ protected

      private:          //-------------------------------------- private
        // Note that we try to use a leading underscore to indicate
        // private members. We aren't always consistent!
        int _mouseDown;         // track if mouse down
        int _begx; int _begy;   // starting point
        int _curx; int _cury;   // current point
        point *_pt;             // the array of points
        int _nextpt;            // where next point goes
      };
#endif
```

A.4 A Modeless Dialog

tdialog.cpp

```
//========================================================================
//  tdialog.cpp - Source file for tutorial tDialog class
//
//  Copyright 1995, Bruce E. Wampler, All Rights Reserved
//========================================================================

// #include the headers we need
#include <v/vnotice.h>
#include "tdialog.h"

// The structure of a derived dialog class is very similar to the
// structure of a command window class. First we define IDs for the
// various command objects used in the dialog. Then we declare the
// static array that defines the dialog.

const ItemVal mdLbl1 = 200;

const ItemVal mdFrm1 = 201;  const ItemVal mdLbl2 = 202;
const ItemVal mdCB1 = 203;   const ItemVal mdCB2 = 204;
const ItemVal mdCB3 = 205;

const ItemVal mdFrmV1 = 206; const ItemVal mdLbl3 = 207;
const ItemVal mdRB1 = 208;   const ItemVal mdRB2 = 209;

const ItemVal mdFrmV2 = 210; const ItemVal mdLbl4 = 211;
const ItemVal mdBtn1 = 212;  const ItemVal mdBtn2 = 213;

const ItemVal mdBtnChange = 214;

    static char change_me[] = "Change Me A";    // a label to change

// This defines the dialog

    static DialogCmd DefaultCmds[] =
      {
        {C_Label, mdLbl1, 0,"X",NoList,CA_MainMsg,isSens,NoFrame, 0, 0},

        {C_Frame,mdFrmV2,0,"",NoList,CA_None,isSens,NoFrame,0,mdLbl1},
        {C_Label,mdLbl4,0,"Buttons",NoList,CA_None,isSens,mdFrmV2,0,0},
        {C_Button,mdBtn1,mdBtn1,"Button 1",NoList,CA_None,
                isSens,mdFrmV2,0,mdLbl4},
        {C_Button,mdBtn2,mdBtn2,"Button 2",NoList,CA_None,
```

```
                isSens,mdFrmV2,0,mdBtn1},

        {C_Frame,mdFrm1,0,"",NoList,CA_None,isSens,NoFrame,mdFrmV2,mdLbl1},
        {C_Label,mdLbl2,0,"CheckBox",NoList,CA_None,isSens,mdFrm1,0,0},
        {C_CheckBox,mdCB1,0,"Test A",NoList,CA_None,
                isSens,mdFrm1,0,mdLbl2},
        {C_CheckBox,mdCB2,0,"Test B",NoList,CA_None,
                isSens,mdFrm1,mdCB1,mdLbl2},
        {C_CheckBox,mdCB3,1,"Test C",NoList,CA_None,isSens,mdFrm1,0,mdCB1},

        {C_Frame,mdFrmV1,0,"",NoList,CA_None,isSens,NoFrame,mdFrm1,mdLbl1},
        {C_Label,mdLbl3,0,"Radios",NoList,CA_None,isSens,mdFrmV1,0,0},
        {C_RadioButton,mdRB1,1,"KOB",NoList,CA_None,
                isSens,mdFrmV1,0,mdLbl3},
        {C_RadioButton,mdRB2,0,"KOAT",NoList,CA_None,
                isSens,mdFrmV1,0,mdRB1},

        {C_Button,mdBtnChange,0,change_me,NoList,CA_None,
                isSens,NoFrame,0,mdFrmV1},
        {C_Button,M_Cancel,M_Cancel," Cancel ",NoList,CA_None,
            isSens,NoFrame,mdBtnChange,mdFrmV1},
        {C_Button,M_OK,M_OK," OK ",NoList,CA_DefaultButton,
            isSens,NoFrame,M_Cancel,mdFrmV1},

        {C_EndOfList,0,0,0,0,CA_None,0,0,0}
      };


//==========================>>> tDialog::tDialog <<<======================
  tDialog::tDialog(vBaseWindow* bw) :
    vDialog(bw)
  {
    // The constructor for a derived dialog calls the superclass
    // constructor, and then adds the command objects to the dialog
    // by calling AddDialogCmds.

    UserDebug(Constructor,"tDialog::tDialog()\n")
    AddDialogCmds(DefaultCmds);         // add the command objects
  }

//=========================>>> tDialog::~tDialog <<<======================
  tDialog::~tDialog()
  {
    // Destructor often doesn't need to do anything

    UserDebug(Destructor,"tDialog::~tDialog() destructor\n")
  }

//====================>>> tDialog::DialogCommand <<<======================
  void tDialog::DialogCommand(ItemVal id, ItemVal retval, CmdType ctype)
  {
    // After the user has selected a command from the dialog,
    // this routine is called with the value.

    vNoticeDialog note(this);   // an instance we can use

    UserDebug1(CmdEvents,"tDialog::DialogCommand(id:%d)\n",id)

    switch (id)         // We will do some things depending on value
      {
        case mdCB1:             // CheckBox
            note.Notice("Test A");
            break;

        case mdCB2:             // CheckBox
```

```
          note.Notice("Test B");
          break;

      case mdCB3:             // CheckBox
          note.Notice("Test C");
          break;

      case mdRB1:             // Radio Button
          note.Notice("KOB");
          break;

      case mdRB2:             // Radio Button
          note.Notice("KOAT");
          break;

      case mdBtn1:            // Button
          note.Notice("Button 1");
          break;

      case mdBtn2:            // Button
          note.Notice("Button 2");
          break;

      case mdBtnChange:       // Example: change my own label
          // We will change the label on this button
          change_me[10]++;            // change the "A"
          SetString(mdBtnChange, change_me);
          break;
     }
    // All commands should also route through the parent handler
    // which has useful default behaviors for Cancel and OK
    vDialog::DialogCommand(id,retval,ctype);
  }
```

tdialog.h

```
//=======================================================================
//
//  tdialog.h - Header file for tutorial tDialog class
//
//  Copyright 1995, Bruce E. Wampler, All Rights Reserved
//=======================================================================

#ifndef TDIALOG_H
#define TDIALOG_H

#include <v/vdialog.h>  // we derive from vDialog

    class tDialog : public vDialog
      {
      public:           //---------------------------------------- public
        tDialog(vBaseWindow*);
        virtual ~tDialog();            // Destructor
        virtual void DialogCommand(ItemVal id, ItemVal retval,
                CmdType ctype);

      protected:        //------------------------------------- protected

      private:          //--------------------------------------- private
        int _toggleId;
      };
#endif
```

A.5 A Modal Dialog

tmodal.cpp

```
//=======================================================================
//  tmodal.cpp - Source file for tutorial tModalDialog class
//
//  Copyright 1995, Bruce E. Wampler, All Rights Reserved
//=======================================================================
//

#include "tmodal.h"             // our header file
#include <v/vnotice.h>

const ItemVal mmLbl1 = 300;
const ItemVal mmBtn1 = 301;
const ItemVal mmBtn2 = 302;

    static CommandObject DefaultCmds[] =
      {
        {C_Label, mmLbl1, 0,"X",NoList,CA_MainMsg,isSens,NoFrame, 0, 0},

        {C_Button,mmBtn1,mmBtn1," Test 1 ",NoList,CA_None,
                isSens,NoFrame,0,mmLbl1},
        {C_Button,mmBtn2,mmBtn2," Test 2 ",NoList,CA_None,
                isSens,NoFrame, mmBtn1,mmLbl1},

        {C_Button,M_Cancel,M_Cancel," Cancel ",NoList,CA_None,
                isSens,NoFrame, 0,mmBtn1},
        {C_Button,M_OK,M_OK,"   OK   ",NoList,CA_DefaultButton,
                isSens,NoFrame,M_Cancel,mmBtn1},

        {C_EndOfList,0,0,0,0,CA_None,0,0,0}
      };


//======================>>> tModalDialog::tModalDialog <<<================
  tModalDialog::tModalDialog(vBaseWindow* bw) :
    vModalDialog(bw)
  {
    UserDebug(Constructor,"tModalDialog::tModalDialog()\n")
    AddDialogCmds(DefaultCmds);         // add the predefined commands
  }

//=================>>> tModalDialog::~tModalDialog <<<====================
  tModalDialog::~tModalDialog()
  {
    UserDebug(Destructor,"tModalDialog::~tModalDialog() destructor\n")
  }

//===================>>> tModalDialog::DialogCommand <<<==================
  void tModalDialog::DialogCommand(ItemVal id,ItemVal retval,CmdType ctype)
  {
    // After the user has selected a command from the dialog,
    // this routine is called with the id and retval.

    vNoticeDialog note(this);

    UserDebug1(CmdEvents,"tModalDialog::DialogCommand(id:%d)\n",id)

    switch (id)         // We will do some things depending on value
      {
        case mmBtn1:            // Button 1
            note.Notice(" Test 1 ");
            break;
```

```
        case mmBtn2:            // Button 2
            note.Notice(" Test 2 ");
            break;
      }

    // let default behavior handle Cancel and OK
    vModalDialog::DialogCommand(id,retval,ctype);
  }
```

tmodal.h

```
//========================================================
//  tmodal.h - Header file for tModalDialog class
//
//  Copyright 1995, Bruce E. Wampler, All Rights Reserved
//========================================================

#ifndef TMODAL_H
#define TMODAL_H

#include <v/vmodald.h>  // derived from vModalDialog

    class tModalDialog : public vModalDialog
      {
      public:           //--------------------------------------- public
        tModalDialog(vBaseWindow*);
        virtual ~tModalDialog();                // Destructor
        virtual void DialogCommand(ItemVal id, ItemVal retval,
                CmdType ctype);

      protected:        //--------------------------------------- protected

      private:          //--------------------------------------- private

      };
#endif
```

A.6 The Makefile

makefile

```
# Sample GNU make makefile for V tutorial application
CC      =       g++

# Note: Platform dependent for a Linux system

HOME    =       /home/bruce
X11INC  =       /usr/X11/include
X11LIB  =       /usr/X11R6/lib
Arch    =       intel
LIBS    =       -lV -lXaw -lXmu -lXt -lXext -lX11

VPATH   =       ../include

# Architecture dependent

VLibDir =       $(HOME)/v/lib/$(Arch)

oDir    =       ../obj/$(Arch)

LibDir  =       ../lib/$(Arch)
```

```
Bin     =       ../bin/$(Arch)

#--------------------------------------------------------------

# Typical flags for includes and libraries

CFLAGS  =       -O -I$(X11INC) -I$(HOME)

LFLAGS  =       -O -L$(X11LIB) -L$(VLibDir)

EXOBJS  =       $(oDir)/tutapp.o \
                $(oDir)/tdialog.o \
                $(oDir)/tmodal.o \
                $(oDir)/tcanvas.o \
                $(oDir)/tcmdwin.o

all:    $(Bin)/tutapp

$(Bin)/tutapp:  $(EXOBJS) $(VLibDir)/libV.a
        $(CC) -o $@ $(LFLAGS) $(EXOBJS) $(LIBS)

$(oDir)/tcanvas.o:      tcanvas.cpp v_defs.h tcanvas.h
        $(CC) -c $(CFLAGS) -o $@ $<

$(oDir)/tdialog.o:      tdialog.cpp v_defs.h tdialog.h
        $(CC) -c $(CFLAGS) -o $@ $<

$(oDir)/tmodal.o:       tmodal.cpp v_defs.h tmodal.h
        $(CC) -c $(CFLAGS) -o $@ $<

$(oDir)/tcmdwin.o:      tcmdwin.cpp v_defs.h tcmdwin.h
        $(CC) -c $(CFLAGS) -o $@ $<

$(oDir)/tutapp.o:       tutapp.cpp v_defs.h tdialog.h tmodal.h \
        tutapp.h tcmdwin.h
        $(CC) -c $(CFLAGS) -o $@ $<
```

Appendix B

C++ Coding Style Guidelines

I have developed the following guidelines for writing C++ code over my long career as a programmer. All of ***V*** has been written using these guidelines, and I believe that using them is a big first step leading to readable, portable, and reliable code. Of course, just following these guidelines won't automatically give you that, but I think they are still necessary.

B.1 Readability

The ultimate goal of style guidelines is to help you to write code that is readable. While this means code that is readable by you, it mostly means code that is readable by others. Remember, *code has a life of its own!* No matter how small the project may seem, or how temporary, most code ends up being used and reused much longer than you might think. The real cost of software is often in the long term maintenance. While you may end up maintaining your own code, often it will be someone else. Even if it is you, after a few months, or even weeks, you will have likely forgotten just exactly what you were doing when you wrote the code to begin with.

The point of this is to emphasize the importance of producing readable code. Generally, readable code is inviting to look at. It is visually pleasing, just as a well designed book is well laid out and visually pleasing to look at. Your code should have plenty of visual attributes that make it easy to read. This means lots of whitespace, consistent indentation, abundant, well formatted comments, and visual separation of important sections of code. Much of the structure of your code should be visually obvious without having to read it. Many of the following guidelines are intended to help you produce readable code.

B.2 Naming

It is critical to choose meaningful names for your variables and functions. Avoid short, two or three letter names unless those names are really meaningful. While you may want to use short, abbreviated names to avoid typing, this habit will make your code more difficult to read later. While you should avoid short names, consistently using names that are too long can present problems, too. This can lead to code that must be split across multiple lines because the names are too long. Even so, it is probably better to trend to overly long names than short, abbreviated names.

Names should use both upper and lower case letters, using a case change to indicate word breaks. For example, a name like `maxLength` is more readable than `MAXLENGTH`, `maxlength`, or even `max_length`. In general, using mixed case is better than using underscores. Underscores are better used to indicate special classes of variables (see Class Definitions below).

B.3 Files

Each C++ module should be split into two files – the `.h` header file which contains `class` definitions and variable declarations, and a `.c` or `.cpp`[1] file that contains source code for the functions.

Generally, each class will have its own `.h` and `.cpp` files. Utility helper functions that go with a class can be included in the same file as the class. Other functions that do not go with a class should be collected into logical groups and kept in a separate file. In general, files should not be much larger than twenty to forty thousand characters long.

B.4 Include Files

Include files or header files (`.h`) files must each have a `#define` statement that prevents problems caused by multiple inclusion. The standard way to do this is:

```
//
//  myclass.h - header file for myclass class definition
//
#ifndef MYCLASS_H              // Check to prevent
#define MYCLASS_H              // multiple inclusion

    ... definitions go here

#endif                         // last line of file
```

Files are included from the source file by placing the #include statement near the beginning of the source file, starting in column one.

```
#include "myclass.h"      // includes start in column 1
```

B.5 Function Definitions

All functions should have a prototype definition for use by others. For class methods, this will be part of the class declaration. For other functions, this should also be in a `.h` file. The parameters of all prototypes should include both the *type* and the *name* of each parameter since the name often conveys extra useful information.

The body of each method or function should use this convention:

```
//=================>>> myClass::myMethod <<<==================
  void myClass::myMethod(const int size)
  {
    // An introductory block of comments explaining the purpose
    // and interface to this function. You can also include an
    // author and modification history here if appropriate.

    ... declare variables used throughout the function

    ... body of function
  }
```

[1] The naming conventions for C++ source files has not really been standardized yet. Common alternatives for .cpp include .C and .cxx.

Each function should include the separator line to visually separate the body of the function from others in the same file. The preferred indentation for the function name is two spaces, with the enclosing { and } braces on separate lines, also indented two spaces. An acceptable alternative style is to have these lines start in the first column.

Following the opening { should come an introduction to the function. Variables required by the entire function follow the opening comments. Following that is the body of the function. Make liberal use of whitespace for visual separation.

B.6 Indentation

The preferred indentation scheme is based on groups of four spaces, with braces indented two additional spaces. It is acceptable to keep braces lined up with the outer statement rather than indenting two extra, but all braces *must* be on a line by themselves. This spacing works well with standard eight character tab stops – your code will either be indented on even tab stops, or on tab stops plus four.

Except for the most trivial cases of short, related assignment statements, each statement should be on a separate line. The body of loops and conditional statements should always use braces – never use a simple statement. There are two reasons for this. First, using braces on separate lines adds whitespace, which adds to the readability. Second, code is inevitably modified, and by always using braces, you will be more likely to add a statement in the proper place. As a special case for initializer loops with no code body, it is acceptable to use just a semicolon rather than braces.

The old K&R style of placing the opening brace at the end of the line is not acceptable. Most importantly, you lose the visual impact of lined up braces when you do this. It also tends to compress the code, and extra whitespace really helps make code more readable.

When calling functions that require long, complex argument lists, it is often advisable to place each argument on a separate line accompanied by an explanatory comment.

Use a blank between keywords and the associated left paren: `if (test)`. Don't put a space for function calls: `function(param);`. Don't use parens for the returned value of a `return` statement. This helps to visually distinguish a `return` from a function call.

The following code demonstrates indentation for various C++ statements:

```
//=====================>>> sample <<<=====================
  int sample(int action)
  {
    // This meaningless sample demonstrates indentation.
    // The code should not considered to do anything useful
    // other than demonstrating indentation.

    char* name;                 // explain each variable
    int set;                    // with useful comment

    if (action)                 // indent 4, space after if
      {                         // the { in 2
        set = doSomething(action);
      }                         // always use {}'s
    else
      {
        set = SomethingElse(action);
      }

    switch (set)                // example for switch
      {
        case 1:                 // case in +4 from switch
          {                     // always use braces for cases!
            getName1(name);
            break;
```

```
        }

      case 2:                // Try to comment each case
        {
          int temp = len(name);  // try to declare as needed
          fixName(temp,name);
          break;
        }

      default:               // Good idea to have default
          break;
    }

  // Prefix some blocks with comments like this
  // to describe what a section of code does
  // Note that 'char* cp' is preferred to 'char *cp'.
  // Take advantage of C++ scope rules, and declare
  // variables (e.g., len) as close to use as possible.
  for (int len = 0, char* cp = name ; *cp ; ++cp)
    {
      ++len;                 // all loops use {}'s
      tryThat(set,cp);       // and meaningful comments!
    }

  while (IsStillOK(name))    // indent like for
    {
      Complex(name,          // a complex function call
          set,               // can explain each parameter
          len);              // for easier maintenance
    }

  int status = (checkName(name)) ? len    // sample ?:
                                 : len / 2;

  return status;             // no parens on return
}
```

B.7 Comments

It is difficult to over comment your code. Comments are one of the most helpful things you can do to make your code easier to maintain. A 1 to 1 ratio of comments to code should be considered a bare minimum, with a ratio of more comments than code probably a better thing.

I claim it is almost impossible to have too many comments. A few expert programmers may disagree with this philosophy, and say that well written code can be self-commenting. The problem is that this is not really true. Assume, for example, that you are using a standard software library, such as `Xt` or ***V***. You may know the library backwards and forwards, and it may seem perfectly clear to you what some code is doing. But assume that someone else will be maintaining the code later. They may not know the library as well, and what is obvious and self-commenting to you will be gibberish to them. A few well placed comments explaining what you are doing will be very helpful.

In order to write really effective comments, you must comment *as you write the code*! Do not go back and add comments after the code is written. You can go back and improve and expand your comments, but you should comment as you go. A few seconds taken to add a comment as you write the code can save many minutes or even hours later.

Comments should be meaningful and correct. If you change code, be sure you change the comments to correspond! If you are in the habit of commenting as you write, this will not be so hard.

Make the layout of your comments visually pleasing. Use whitespace to separate sections of code. Line up block comments near the left, and try to keep short per line comments lined up on the right without going too far right. Line them up on a tab stop if possible.

Above all, remember that what seems obvious to you at the moment is likely to be forgotten even a week or two later. And keep in mind that someone else is likely to modify or study your code later. Don't keep secrets. If you had to look something up, or have other information that might make the code more understandable, put that in a comment. If you are doing something tricky or obscure (which you should avoid, but sometimes can't avoid), explain what is going on! You might be teaching a valuable trick to whoever is working with your code later!

My own code has more comments than almost any other code I've seen. Time after time, when someone else has had to use or maintain my code, I've gotten feedback that it is very easy to understand and modify. I attribute much of this positive feedback to the abundant comments found in my code.

B.8 Class Definitions

The standards for class definitions are based on keeping braces on separate lines, and on not using implicit assumptions. Thus, a class will have braces on separate lines, either indented two (the preferred style), or lined up with the class statement.

There should always be all three `public`, `protected`, and `private` sections in that order, even if a section is empty. This order assumes it is more useful to have the public stuff at the top for easier readability. And even if a section is empty, that conveys information about the definition of the class. The prototypes for member functions should include both the type and name (e.g., `int OnOrOff`).

There should almost never be public access to class variables. Provide methods to access and set variables of the class. You may find it helpful to prefix class variables (especially private class variables) with an underscore (_variableName) to indicate the variable is private to the class.

The following example shows indentation and layout of a class definition. Note the visual separator for public, protected, and private, and the alignment with the braces.

```
class myClass : public superClass    // name here
  {
    friend int FriendFunction(int ival);  // friends at top

  public:    //------------------------- public

    myClass();          // constructor and destructor
    virtual ~myClass(); // first

    // simple methods can be inline
    int getVal() { return _val; }
    virtual void service(int iParam);    // prototype

  protected:  //------------------------ protected

    // even an empty section conveys information

  private:    //------------------------ private

    int _val;           // _ for class variables
  };
```

B.9 C++ Language Features

With C++, it is preferred to use `const` definitions of symbolic values rather than `#defines`.

Use `const` parameters whenever possible.

Declare variables as you need them, preferably inside a code block, rather than all at the top of a function. This makes your code much more maintainable, and helps avoid errors introduced by bad reuse of a variable, especially in loops.

For each `new` operation, there *must* be a corresponding `delete` operation. These `new` and `delete` pairs will often be found in the constructor and destructor for your objects.

Always define copy constructors and an assignment operator for each class if they use pointers and dynamic memory allocation using `new`. Some of the biggest problems in C++ code involves objects with pointers to dynamically allocated space. You should use either deep copy semantics or reference counts to avoid creating objects with dangling pointers.

When using ***V*** use the debug macros as much as possible. It is especially helpful to use `UserDebug` statements in constructors and destructors.

B.10 Software Portability

Always remember that your code might someday be ported to a different system. Keep this in mind when writing your code. These guidelines will help to make your code more portable.

Don't use nonstandard or nonportable language features. For example, templates are not yet universally portable. Avoid using them.

Use restrictive names when naming files. The most conservative approach is to use single case names limited to 8 characters for the name part, and 3 for the extension. This should get better as time goes by, but for now this is still a pretty good idea.

If you must use system calls, abstract them and isolate them in a single place.

Don't go behind the back of ***V*** to access X directly.

Avoid conversions that are Big and Little Endian dependent. If you need them, isolate them.

Appendix C

Quick Reference

C.1 CommandObject

```
//  CommandObject: "#include <v/v_defs.h>"

typedef struct CommandObject
  {
    CmdType cmdType;    // what kind of item is this
    ItemVal cmdId;      // unique id for the item
    ItemVal retVal;     // value returned when picked
    char* title;        // string
    void* itemList;     // used when cmd needs a list
    CmdAttribute attrs; // list of attributes
    int Sensitive;      // if item is sensitive or not
    ItemVal cFrame;     // Frame used for an item
    ItemVal cRightOf;   // Item placed left of this id
    ItemVal cBelow;     // Item placed below this one
    int size;           // Optional - for size information
  } CommandObject;
```

CmdType

```
                  (with suggested prefix for id names)
    C_EndOfList:   Used to denote end of command list
    C_Blank:       filler to help RightOfs, Belows work (blk)
    C_BoxedLabel:  a label with a box (bxl)
    C_Button:      Button (btn)
    C_CheckBox:    Checked Item (chk)
    C_ColorButton: Colored button (cbt)
    C_ColorLabel:  Colored label (clb)
    C_ComboBox:    Popup combo list (cbx)
    C_Frame:       General purpose frame (frm)
    C_Icon:        a display only Icon (ico)
    C_IconButton:  a command button Icon (icb)
    C_Label:       Regular text label (lbl)
    C_List:        List of items (lst)
    C_ProgressBar: Bar to show progress (pbr)
    C_RadioButton: Radio button (rdb)
    C_Slider:      Slider to enter value (sld)
    C_Spinner:     Spinner value entry (spn)
    C_TextIn:      Text input field (txi)
    C_Text:        wrapping text out (txt)
    C_ToggleButton: a toggle button (tbt)
    C_ToggleFrame: a toggle frame (tfr)
    C_ToggleIconButton:  a toggle Icon button (tib)
```

CmdAttribute

```
CA_DefaultButton: Special Default Button
   [Used by: C_Button]
CA_Hidden:        Starts out hidden
   [All controls]
CA_Horizontal:    Horizontal orientation
   [C_ProgressBar, C_Slider]
CA_Large:         Command larger than normal
   [C_List C_ProgressBar, C_Slider, C_Spinner, C_TextIn]
CA_MainMsg:       Gets replacement message
   [C_Label]
CA_NoBorder:      No border (frames, status bar)
   [C_Frame,C_ToggleFrame,C_Text,status bar]
CA_NoLabel:       No label on progress bar
   [C_ProgressBar]
CA_NoNotify:      Don't notify on all events
   [C_ComboBox,C_List,C_Spinner]
CA_NoSpace:       No space between widgets
   [C_Frame,C_ToggleFrame]
CA_None:          No special attributes
   [All controls]
CA_Percent:       Use % on progress bar
   [C_ProgressBar]
CA_Small:         Command smaller than normal
   [C_ProgressBar,C_TextIn]
CA_Size:          Specify size
   [C_Button,C_List,C_Spinner]
CA_Text:          A Text value box
   [C_Spinner]
CA_Vertical:      Vertical orientation
   [C_ProgressBar, C_Slider]
```

Useful symbolic values

```
NoList:           No list used
NoFrame:          Not a member of a frame
isSens:           Is sensitive
notSens:          Not sensitive
isChk:            Is checked
notChk:           Not checked
notUsed:          Not used
noIcon:           No icon
```

C.2 vApp

```
//  vApp:  "#include <v/vapp.h>"

    class vApp : public vBaseItem
      {
      public:  //---------------------------------------- public
        vApp(char* appName);
        virtual ~vApp();

        // Methods to override

        virtual void AppCommand(vWindow* win, ItemVal id,
                                ItemVal retval, CmdType ctype);
        virtual void CloseAppWin(vWindow* win);
        virtual void Exit(void);
        virtual void KeyIn(vWindow* win, vKey key, unsigned int shift);
        virtual vWindow* NewAppWin(vWindow* win, char* name, int w, int h,
                               vAppWinInfo* winInfo = 0);
```

```
    // Utility methods

    int DefaultHeight();
    int DefaultWidth();
    int IsRunning();
    void SendWindowCommandAll(ItemVal id, int val, CmdType ctype);
    void SetValueAll(ItemVal id, int val, ItemSetType setType);
    void SetStringAll(ItemVal id, char* str);
    const char* name();

    vAppWinInfo *getAppWinInfo(vWindow* Win);

    // platform dependent

    Display* display();           // To get the X display
    XtAppContext appContext();    // To get the context
  };

extern vApp *theApp;     // Pointer to single global instance
extern int AppMain(int argc, char** argv);  // Pseudo main program
```

C.3 vBrush

```
//  vBrush:  "#include <v/vbrush.h>"
    class vBrush
      {
      public:       //-------------------------------- public
        vBrush(unsigned int r = 0, unsigned int g = 0, unsigned int b = 0,
               int style = vSolid, int fillMode = vAlternate);
        ~vBrush();

        int operator ==(vBrush b2);
        int operator !=(vBrush b2);

        vColor GetColor();
        int GetFillMode();
        int GetStyle();
        void SetColor(vColor c);
        void SetFillMode(int fillMode);
        void SetStyle(int style);
      };
```

C.4 vCanvasPane

```
//  vCanvasPane:  "#include <v/vcanvas.h>"

    class vCanvasPane
      {
      public:   //---------------------------------------- public

        vCanvasPane(PaneType pt = P_Canvas);
        virtual ~vCanvasPane();

        virtual void ShowPane(int OnOrOff);

        // Cursor
        void SetCursor(VCursor id);
        VCursor GetCursor();
        void UnSetCursor(void);

        // Scrolling
```

```
virtual void HPage(int Shown, int Top);
virtual void VPage(int Shown, int Top);
virtual void HScroll(int step);
virtual void VScroll(int step);
virtual void SetHScroll(int Shown, int Top);
virtual void SetVScroll(int Shown, int Top);
virtual int GetHScroll(int& Shown, int& Top);
virtual int GetVScroll(int& Shown, int& Top);
virtual void ShowHScroll(int OnOff);
virtual void ShowVScroll(int OnOff);

// Change messages
virtual void FontChanged(vFont vf);

// Events
virtual void MouseDown(int x, int y, int button);
virtual void MouseUp(int x, int y, int button);
virtual void MouseMove(int x, int y, int button);
virtual void EnterFocus();
virtual void LeaveFocus();

// Expose/redraw events
virtual void Redraw(int x, int y, int width , int height);
virtual void Resize(int newW, int newH);

// Information
virtual int GetWidth();
virtual int GetHeight();

// Drawing
void Clear(void);
virtual void ClearRect(int left, int top, int width, int height);
virtual void DrawAttrText(int x, int y, char* text, const ChrAttr attr);
virtual void DrawColorPoints(int x, int y, int nPoints, vColor* pts);
virtual void DrawText(int x, int y, char* text);
virtual void DrawEllipse(int x, int y, int width, int height);
virtual void DrawIcon(int x, int y, vIcon& icon);
virtual void DrawLine(int x, int y, int xend , int yend);
virtual void DrawPoint(int x, int y);
virtual void DrawPolygon(int n, vPoint points[], int fillStyle = vAlternate);
virtual void DrawRectangle(int x, int y, int width, int height);
virtual void DrawRoundedRectangle(int x, int y, int width, int height, int radius);
virtual void DrawRubberLine(int x, int y, int xend, int yend);
virtual void DrawRubberEllipse(int x, int y, int width, int height);
virtual void DrawRubberRectangle(int x, int y, int width, int height);
virtual vBrush GetBrush(void);
virtual void SetBrush(vBrush brush);
virtual vFont GetFont(void);
virtual void SetFont(vFont fnt);
virtual int TextWidth(char* str);
virtual int TextHeight(int& ascent, int& descent);
vDC* GetDC();

// Appearance
virtual void SetScale(int mult, int div);
virtual void GetScale(int& mult, int& div);
virtual void SetBackground(vColor color);
virtual void SetPen(vPen pen);
vPen GetPen();
void SetTranslate(int x, int y);
void SetTransX(int x);
void SetTransY(int y);
void GetTranslate(int& x, int& y);
int GetTransX();
int GetTransY();
```

```
    // Platform dependent
    Widget DrawingWindow();
  };
```

C.5 vCmdWindow

```
//  vCmdWindow:  "#include <v/vcmdwin.h>"

    class vCmdWindow : public vWindow
      {
      public:           //----------------------------------- public
        vCmdWindow(char* name = "+", int width = 0, int height = 0);
        virtual ~vCmdWindow();                         // Destructor

        virtual void CloseWin(void);    // callback for close button
      };
```

C.6 vColor

```
//  vColor:  "#include <v/vcolor.h>"

// Message constants for use in Color buttons (for color buttons, etc.)

        M_Black, M_Red, M_DimRed, M_Green, M_DimGreen, M_Blue, M_DimBlue,
        M_Yellow, M_DimYellow, M_Magenta, M_DimMagenta, M_Cyan, M_DimCyan,
        M_DarkGray, M_MedGray, M_White,
        M_ColorFrame

// Index constants into V "standard" color arrays: vStdColors, vColorNames.

        vC_Black, vC_Red, vC_DimRed, vC_Green, vC_DimGreen, vC_Blue, vC_DimBlue,
        vC_Yellow, vC_DimYellow, vC_Magenta, vC_DimMagenta, vC_Cyan, vC_DimCyan,
        vC_DarkGray, vC_MedGray, vC_White

    class vColor
      {
      public:    //--------------------------------------- public
        vColor(unsigned int rd = 0, unsigned int gr = 0, unsigned int bl = 0);
        ~vColor();

        int operator ==(vColor c2);
        int operator !=(vColor c2);
        int BitsOfColor();
        ResetColor(unsigned int rd = 0, unsigned int gr = 0, unsigned int bl = 0);
        Set(unsigned int rd = 0, unsigned int gr = 0, unsigned int bl = 0);
        Set(vColor& c);
        void SetR(unsigned int rd = 0);
        void SetG(unsigned int gr = 0);
        void SetB(unsigned int bl = 0);

        unsigned int r();
        unsigned int g();
        unsigned int b();
      };

    extern vColor vStdColors[16];       // 16 "standard" colors
    extern char* vColorNames[16];       // and their names
```

C.7 vDebugDialog

```
// vDebugDialog: "#include <v/vdebug.h>"

    class vDebugDialog : public vModalDialog
      {
      public:    //--------------------------------------- public
        vDebugDialog(vBaseWindow* bw,char* title = "Debugging Options");
        vDebugDialog(vApp* aw,char* title = "Debugging Options");
        ~vDebugDialog();
        void SetDebug();
      };
```

Command Line Switches

-vDebug SU<list> Turn on System or User (or both) debug messages in list.

c Command events (menu, dialog commands).

m Mouse events.

w Window events (resize, redraw).

b Build events.

o Other misc. events.

v Bad values.

C Constructors.

D Destructors.

123 User items 1, 2, or 3.

C.8 vDialog

```
// vDialog: "#include <v/vdialog.h>"

    class vDialog
      {
      public:             //----------------------------------- public

        vDialog(vBaseWindow* creator, int modal = 0, char* title = "");
        vDialog(vApp* creator, int modal = 0, char* title = "");
        ~vDialog();

        WindowType wType();

        virtual void AddDialogCmds(CommandObject* cList);
        virtual void CancelDialog(void);
        virtual void CloseDialog(void);
        virtual void SetDialogTitle(char * title);
        virtual void DialogCommand(ItemVal id, ItemVal retval, CmdType ctype);
        virtual int GetTextIn(ItemVal id, char* strout, int maxlen);
        virtual int GetValue(ItemVal id);
        virtual void SetValue(ItemVal id, ItemVal val, ItemSetType setType);
        virtual void SetString(ItemVal id, char* str);
        int IsDisplayed(void);
        virtual void ShowDialog(const char* msg);
      };
```

C.9 vFileSelect

```
// vFileSelect: "#include <v/vfilesel.h>"

    class vFileSelect : protected vModalDialog
      {
      public:     //--------------------------------------- public
        vFileSelect(vBaseWindow* bw, char* title = "File Select");
        vFileSelect(vApp* aw, char* title = "File Select");
        ~vFileSelect();

        int FileSelect(const char* msg, char* filename,
            const int maxlen, char** filter);
        int FileSelectSave(const char* msg, char* filename,
            const int maxlen, char** filter);
      };
```

C.10 vFont

```
// vFont: "#include <v/vfont.h>"
    enum vFontID                // various font related ids
      {
        vfDefaultSystem,        // the default system font
        vfDefaultFixed,         // the system default fixed font
        vfDefaultVariable,      // the system default variable font
        vfSerif,                // serifed font - TimesRoman
        vfSansSerif,            // SansSerif - Swiss or Helvetica
        vfFixed,                // fixed font - Courier
        vfDecorative,           // decorative - dingbat
        vfOtherFont,            // for all other fonts
        vfNormal,               // normal style, weight
        vfBold,                 // boldface
        vfItalic,               // italic style
        vfEndOfList
      };

    class vFont         // make the font stuff a class to make it portable
      {
      public:           //--------------------------------------- public
        vFont(vFontID fam = vfDefaultFixed, int size = 10,
           vFontID sty = vfNormal, vFontID wt = vfNormal, int und = 0,
           int angle = 0);
        ~vFont();

        vFontID GetFamily() { return _family; }
        int GetPointSize() { return _pointSize; }
        vFontID GetStyle() { return _style; }
        vFontID GetWeight() { return _weight; }
        int GetUnderlined() { return _underlined; }
        void SetFontValues(vFontID fam = vfDefaultFixed, int size = 10,
           vFontID sty = vfNormal, vFontID wt = vfNormal, int und = 0);
      };
#endif
```

C.11 vFontSelect

```
// vFontSelect: "#include <v/vfontsel.h>"

    class vFontSelect : protected vModalDialog
      {
      public:      //--------------------------------------- public
        vFontSelect(vBaseWindow* bw, char* title = "Select Font");
```

```
        vFontSelect(vApp* aw, char* title = "Select Font");
       ~vFontSelect();

       int FontSelect(vFont& font, const char* msg = "Select Font" );
      };
#endif
```

C.12 vIcon

```
    // <v/vicon.h>

    class vIcon     // an icon
      {
      public:             //---------------------------------------- public
        vIcon(unsigned char* ic, int h, int w, int d = 1);
        ~vIcon();
        int height;             // height in pixels
        int width;              // width in pixels
        int depth;              // bits per pixel
        unsigned char* icon;    // ptr to icon array

      protected:        //--------------------------------------- protected
      private:          //--------------------------------------- private
      };
```

C.13 vMenu

```
//  vMenu:  "#include <v/v_menu.h>"

    typedef struct vMenu
      {
        char* label;       // The label on the menu
        ItemVal menuId;    // A User assigned unique id
        unsigned
          sensitive : 1,   // If item is sensitive or not
          checked : 1;     // If item is checked or not
        char* keyLabel;    // Label for an accelerator key
        vKey accel;        // Value of accelerator key
        vMenu* SubMenu;    // Ptr to a submenu
      } MenuItem;
```

Useful symbolic values

```
    isSens:           Is sensitive
    notSens:          Not sensitive
    noSub:            No submenu
    isChk:            Is checked
    notChk:           Not checked
    noKey:            No accelerator specified
    noKeyLbl:         No accelerator label
```

C.14 vModalDialog

```
//  vModalDialog:  "#include <v/vmodald.h>"

    class vModalDialog : public vDialog
      {
      public:   //---------------------------------------- public

        vModalDialog(vBaseWindow* creator, char* title = "");
```

```
    vModalDialog(vApp* creator, char* title = "");
    virtual ~vModalDialog();

    virtual ItemVal ShowModalDialog(const char* msg, ItemVal& retval);

    // rest are inherited from vDialog
  };
```

C.15 vNoticeDialog

```
//  vNoticeDialog:  "#include <v/vnotice.h>"

    class vNoticeDialog : protected vModalDialog
      {
      public:    //--------------------------------------- public
        vNoticeDialog(vBaseWindow* bw, char* title = "Notice");
        vNoticeDialog(vApp* aw, char* title = "Notice");
        ~vNoticeDialog();

        void Notice(char* msg);
      };
```

C.16 vPen

```
//  vPen:  "#include <v/vpen.h>"

    class vPen
      {
      public:    //--------------------------------------- public
        vPen(unsigned int r = 0, unsigned int g = 0,
             unsigned int b = 0, int width = 1, int style = vSolid);
        ~vPen();

        int operator ==(vPen p2);
        int operator !=(vPen p2);

        void SetStyle(int style);
        int GetStyle(void);
        void SetWidth(int width);
        int GetWidth();
        void SetColor(vColor c);
        vColor GetColor();
      };
```

C.17 vReplyDialog

```
//  vReplyDialog:  "#include <v/vreply.h>"

    class vReplyDialog : protected vModalDialog
      {
      public:   //---------------------------------------- public
        vReplyDialog(vBaseWindow* bw, char* title = "Reply");
        vReplyDialog(vApp *aw, char* title = "Reply");

        int Reply(const char* msg, char* reply, const int maxlen);
      };
```

C.18 vStatus

```
//  vStatus:  "#include <v/v_defs.h>"

    typedef struct vStatus      // for status bars
      {
        char* label;            // text label
        ItemVal statId;         // id
        CmdAttribute attrs;     // attributes - CA_NoBorder, CA_None
        unsigned sensitive : 1; // if button is sensitive or not
        int width;              // to specify width (0 for default)
      } vButton;
```

Useful symbolic values

```
    CA_NoBorder:      No border (frames, status bar)
    CA_None:          No special attributes
    isSens:           Is sensitive
    notSens:          Not sensitive
    noIcon:           No icon
```

C.19 vTextCanvasPane

```
//  vTextCanvasPane:  "#include <v/vtextcnv.h>"

    class vTextCanvasPane : public vCanvasPane
      {
      public:  //---------------------------------------- public
        vTextCanvasPane();
        virtual ~vTextCanvasPane();

        // Window management/drawing

        virtual void Clear(void);
        virtual void ClearRow(const int r, const int c);
        virtual void ClearToEnd(const int r, const int c);
        int GetCols();
        int GetRows();
        void GetRC(int& r, int& c);
        void GotoRC(const int r, const int c);
        virtual void DrawAttrText(const char* text, const ChrAttr attr);
        virtual void DrawChar(const char chr, const ChrAttr attr);
        virtual void DrawText(const char* text);
        void HideTextCursor(void);
        void ShowTextCursor(void);

        // Scrolling
        void ScrollText(const int lineCount);

        // Events
        virtual void ResizeText(const int rows, const int cols);
        virtual void TextMouseDown(int row, int col, int button);
        virtual void TextMouseUp(int row, int col, int button);
        virtual void TextMouseMove(int row, int col, int button);
      };
```

C.20 vTimer

```
//  vTimer:  "#include <v/vtimer.h>"

    class vTimer
```

```
{
public:   //--------------------------------------- public
  vTimer();
  virtual ~vTimer();

  virtual int TimerSet(long interval);
  virtual void TimerStop(void);
  virtual void TimerTick(void);
};
```

C.21 V Utilities

```
//  V Utilities:  "#include <v/vutil.h>"

    extern void LongToStr(long intg, char* str);
    extern void IntToStr(int intg, char* str);
    extern void vGetLocalTime(char* tm);
    extern void vGetLocalDate(char* tm);
```

C.22 vWindow

```
//  vWindow:  "#include <v/vwindow.h>"

    enum WindowType
      { WINDOW, CMDWINDOW, DIALOG };  // Types of windows

    class vWindow
      {
      public:           //-------------------------------- public
        vWindow(char *name = "+", int width = 0, int height = 0,
           WindowType wintype = WINDOW);    // Constructor

        virtual ~vWindow();                 // Destructor

        // Methods to Override

        virtual void KeyIn(vKey keysym, unsigned int shift);
        virtual void MenuCommand(ItemVal id);
        virtual void WindowCommand(ItemVal id, ItemVal retval, CmdType ctype);

        // Utility Methods

        const Widget vHandle();         // X only - native handle
        const char* name();             // the name set
        virtual void ShowWindow(void);
        WindowType wType();          // what kind of window we are

        virtual void AddPane(vPane* add_pane);
        virtual void CloseWin(void);
        virtual int GetValue(ItemVal id);
        virtual void RaiseWindow(void);
        virtual void SetString(ItemVal id, char* str);
        virtual void SetStringAll(ItemVal id, char* str);
        virtual void SetTitle(char* title);
        virtual void SetValue(ItemVal id, int val, ItemSetType setType);
        virtual void SetValueAll(ItemVal id, int val, ItemSetType setType);
        void ShowPane(vPane* wpane, int OnOrOff);
      };
```

C.23 vYNReplyDialog

```
//  vYNReplyDialog:  "#include <v/vynreply.h>"

    class vYNReplyDialog : protected vModalDialog
      {
      public:     //--------------------------------------- public
        vYNReplyDialog(vBaseWindow* bw, char* title = "Yes or No");
        vYNReplyDialog(vApp* aw, char* title = "Yes or No");
        ~vYNReplyDialog();

        int AskYN(const char* msg);
      };
```

Appendix D

V Programming Tools

D.1 The V Application Generator

The ***V*** Application Generator will automatically generate C++ code needed to build a simple ***V*** application. It has several options that let you specify the name of the application, the name of your derived classes, and what ***V*** interface elements to include in the application. The ***V*** Application Generator, `vgen`, does not generate code that does any real work, it just provides a very good starting skeleton program to build your application.

In the past, the approach for beginning a new ***V*** application has been to copy one of the example programs, and modify it. `Vgen` has the advantage of allowing a certain amount of customization with names and interface elements included in the basic skeleton program.

On Linux or other Unix platforms, `vgen` will generate the skeleton code and a makefile compatible with GNU make. On the Windows version, just the files will be generated, and it is up to you to include them in the project file of your compiler. This is usually a very trivial operation.

Once you have generated and compiled the skeleton application, you can modify the code to build your own application. It is highly recommend that you start every new ***V*** application this way to get a consistency in the structure of the code.

D.1.1 Overview

`Vgen` is a very simple program to use. You run it, and then select if you are generating a standard ***V*** application skeleton, or if you are generating an extra dialog skeleton to add to an existing application. You can also generate a skeleton for a menuless and canvasless ***V*** application. You can also specify the directory where the generated code should be placed. Once you have generated the skelton, you should get it to compile, either by using the generated makefile, or by building an appropriate project with a Windows C++ compiler.

The most common use of `vgen` is to generate a standard ***V*** application skeleton. This consists of a derived `vCommandWindow` class with a simple menu bar, a command pane with a sample button, a derived `vCanvasPane`, and a status pane. The standard ***V*** skeleton also allows you to generate a modeless and a modal dialog. You can specify the name of the classes you want to use, as well as the file names to use for each of the generated files. The standard files generated include a file for the derived `vApp` class, a file for the derived `vCmdWindow` class, a file for the derived `vCanvasPane` class, and files for the dialogs. The X version will also generate a GNU compatible makefile.

`Vgen` also will generate extra copies of dialogs. You can specify the class name of the dialog, and then generate a skeleton file. These dialogs must be added manually to the basic skeleton application.

The remainder of the reference manual will explain each menu item and each dialog.

D.1.2 File Menu

The file menu only has an Exit command, which closes `vgen`.

D.1.3 Generate Menu

The `Generate` menu selects which type of code you want to generate. The `vgen` tool bar includes shortcuts for the most common operations.

Generate:Standard Application

This option brings up a dialog that controls the generation of a standard ***V*** application. This section will explain each option contained on the Standard V App dialog.

When `vgen` generates a skeleton application, it uses some fixed conventions for naming derived classes and file names. The *Application Base Name* input lets you specify the base name of each class. The default base name is `my`. Thus, `vgen` will generate the derived class names `myApp`, `myCmdWindow`, `myCanvasPane` or `myTextCanvasPane`, `myDialog`, and `myModalDialog`.

The *File Base Name* input lets you control the base name of the generated code files. If you intend to do development for Windows, it is recommended that you specify a name that follows the 8 character limit on file base names. Using the default `my` file base name, `vgen` will generate the files `myapp.cpp` and `myapp.h`, `mycmdw.cpp` and `mycmdw.h`, `mycnv.cpp` and `mycnv.h`, `mydlg.cpp` and `mydlg.h`, and `mymdlg.cpp` and `mymdlg.h`. If you generate a makefile, it will be called `makefile.my`.

The generate dialog allows you to control which interface elements are included in the generated code. You can elect to include a tool bar and a status bar. You can also include code that shows the date and time on the status bar if you wish. You can control if the code generates Windows MDI or SDI model code (this has no effect on the X code). The command window class includes a short, standard menu bar that you can later modify to add your own menu items. You can also specify a title that will appear in the app's title bar.

You can generate a canvas pane derived from a `vCanvasPane` or a `vTextCanvasPane`. You can elect to show the vertical or horizontal scroll bars by default.

You can also control generation of a modal and a modeless dialog. If you include these, code to activate the dialogs will be included in the menu bar. You will usually modify that code to activate the dialogs in a manner needed by your application.

You also have the option of generating a GUN make compatible makefile. The make file needs to know where the `v/include` and ***V*** library files are found on your system. The default `vgen` assumes that these will be located in `/usr/local/v`. There is a variable, `HOMEV` in the make file that sets this path. If ***V*** is found in a different place, you can change it in the generate dialog, or you can change an `ifdef` in the source code and recompile `vgen`.

Finally, you can control where the generated files are written. The *Set Save Path* brings up the standard file selection dialog for where to save the `myapp.cpp` file. That file and the others will be saved in whatever directory you specify. If you don't specify a save path, the files will be saved in the startup directory.

When you have made all your selections, the *Generate* button will generate the skeleton application.

Generate:Extra Dialog

Many applications need more than one modeless or one modal dialog. `Vgen`'s solution to this is not super sophisticated, but it is easier than modifying an existing dialog from scratch. The *Extra Dialog* generate command allows you to generate extra dialog classes that you can then manually add to your main application. The dialogs generated are just like the dialogs that the generate standard app builds, but with a different base name. The options in this dialog include set the class and file base names, the title, modal or modeless, and the save path.

Generate:Menuless/Canvasless App

V supports a menuless, canvasless application. This menu option allows you to generate a simple application of this type, which includes only a window with a tool bar. This option does not generate a menu bar, a canvas, a status bar, or any dialogs.

D.1.4 Future Plans

I hope that `vgen` is only the beginning. It seems pretty useful as it stands, but a full dialog layout editor is planned to complete this tool.

D.2 The V Icon Editor

The ***V*** Icon Editor is a tool used to create and modify icons. It is intended chiefly to create icons for the various ***V*** controls that use icons. It has many editing features found in other icon or bitmap editors, but because it is intended chiefly as an icon editor, it is limited to icons with a maximum size of 150 pixels square and will easily manipulate up to 64 colors, although it will display icons with up to 256 colors. Typically, however, icons tend to be less than 64 pixels square, and use a limited number of colors.

The current version of the ***V*** Icon Editor will handle the native ***V*** `VBM` icon format, as well as `XBM` and `XPM` X Windows format files, and Windows `BMP` format files, so files generated by the X and Windows host platforms can be easily edited and converted to `VBM` format.

The ***V*** Icon Editor was originally developed as a team project for the Spring, 1996 Software Engineering class at the University of New Mexico by Philip Eckenroth, Mike Tipping, Marilee Padilla, and John Fredric Jr. Masciantoni. It has been heavily enhanced by Bruce Wampler. Although this program makes use of many ***V*** features, as a largely student project, the quality of the code is somewhat variable, especially in its use of objects.

D.2.1 Overview

The icon editor functions very much like other similar programs, and should be easy to use. This guide is not intended as a complete tutorial, but more as a brief, but complete, reference.

The ***V*** Icon Editor will usually be called `viconed`. It may be started with the name of a file to edit on the command line.

The interface to the ***V*** Icon Editor consists of a standard menu bar, two tool bars, a drawing canvas, and a status bar. The most common operations are supported by the tool bars (which, for the most part, duplicate menu commands). The drawing canvas shows an enlarged view of the icon as well as an actual size view of the icon. The enlarged view may be zoomed to several sizes, and displayed with or without a grid. The remainder of this guide will describe each menu command, and other features that can be invoked from the tool bars.

You draw an icon using one of three types of brushes: the normal brush, the text brush, and the copy/paste brush. The brush will draw in one of several shapes. Not all shapes work with all three brushes, but you can get some interesting effects using the text or copy/paste brush to draw a line or rectangle, for example. The normal brush also has a choice of several sizes and styles. Drawing with the left mouse button uses the foreground color, while drawing with the right mouse button selects the background color. Colors are selected with the color selection dialog.

D.2.2 File Menu

The *File* menu generally includes commands needed to open, close, and manipulate icon files.

File:New

This will create a new icon using the current canvas. If the current icon has been changed, you will be asked if you want to save it. Then you will be prompted for the size of the new icon. A blank icon of the specified size will be created, and the color palette initialized to a set of default colors.

File:Open

This command is used to open an existing icon in one of the supported formats. If the current drawing canvas has been edited or had an icon loaded, a new canvas window will be opened. The color palette for the canvas window will be initialized to the colors used in the opened icon.

The format of the icon is determined by the file extension. `VBM` is the native ***V*** bitmap format, and is the format required by the various ***V*** icon controls. The current version only supports the 1 and 8 bit `VBM` formats. `Viconed` also supports the X Windows `XBM` monochrome bitmap files, and `XPM` color pixmap files (up to about 90 colors). The Windows `BMP` bitmap format is supported for 8-bit bitmaps. All icons are limited to a maximum of 150 by 150 pixels.

File:Save

This will save the current icon. If the icon was new, you will be prompted for a file name.

File:Save as

You will be prompted for a name to save the current icon. The format of the saved icon is determined by the file extension. `VBM` specifies the standard ***V*** icon format. `Viconed` will automatically save either the monochrome 1-bit `VBM` format, or the 256 color mapped 8-bit `VBM` format. The 24-bit `VBM` format is not supported. The other formats supported include X `XBM` monochrome bitmaps, and X `XPM` color pixmaps up to about 90 colors. The Windows `BMP` bitmap format is supported for 8-bit bitmaps.

When `viconed` saves an icon, it will minimize the size of the color map used in the file.

File:Close

This will close the current icon, asking you if you want to save it if it has changed, and then clear the drawing canvas, ready to create a new icon, or open another icon.

File:About

This displays a dialog with information about `viconed`.

File:Exit

All open icons will be closed, with save prompts as needed, and `viconed` will exit.

D.2.3 Edit Menu

The current version of `viconed` does not yet support standard cut, copy, and paste operations. (It does have copy/paste brush support described later.)

Edit:Undo

This will undo the last operation that changed the icon. Only one level of undo is supported.

Edit:Clear

This will clear the current icon to the background color. A clear is *not* undoable!

Edit:Resize

You can resize the existing icon to a new size. The upper left corner of the current icon will remain constant. If you specify a smaller icon, you will lose the lower and right portions. If you specify a bigger size, then the current icon will become the upper left corner of the new icon. You may find the copy/paste brush useful when resizing an icon.

D.2.4 Draw Menu

The Draw menu is used to select the shape of the brush. The normal brush will draw the selected shape using the current normal brush style. The text brush will draw the shape using the current text. The copy/paste brush will draw the shape using the copied shape.

Using the left button selects the foreground color, while the right button selects the background color.

The shape selections are duplicated on the tool bar for easy interaction.

Draw:Point

This draws a single point, or instance of the text or copy/paste brush. Holding the mouse button down and moving will draw a series of points.

Draw:Line

The button press selects the starting point of the line, and the release selects the end point.

Draw:Rectangle

The button press selects the first corner of the rectangle, and the release the opposite corner. If snap is on, then this will draw a square.

Draw:Rounded Rectangle

This is a rounded rectangle instead of a square cornered rectangle.

Draw:Ellipse

This draws an ellipse, or a circle with snap on.

Draw:Pick Color

This lets you pick a color from the current icon. Pressing the left button will make the color under the cursor the current foreground color, while the right button will pick the background color. Using the pick color (a dropper icon on the tool bar) is often easier than using the color selection dialog.

Draw:Fill

This will fill the closed area with the foreground or background color depending on the mouse button pressed.

Draw:Refresh Image

Normally, this command should not be needed, but it will cause the icon to be redrawn.

Draw:Show Grid

This will turn on or off the display of the drawing grid.

D.2.5 Brush Menu

These select the type of brush to use. Brush selection is duplicated on the tool bar.

All three brushes actually use the same mechanism - a general brush that can hold a pattern to draw onto the icon. A regular brush is usually a pattern of a single pixel, but can be any of the patterns supported by the brush style dialog. The text brush uses text to make the pattern. The copy/paste brush set the pattern based on a selection from the current icon.

You can get some interesting effects by using different brush shapes (point, rectangle, etc.) when drawing with any of the brushes. Using the point shape and then dragging with the mouse held pressed can yield shadow effects, for example.

Brush:Regular Brush

The regular brush draws the currently selected shape using the current regular brush style. The styles include a single pixel point, as well as square, line, and circular shapes of various sizes. The regular brush style is selected from the brush dialog, which is toggled on and off from the tool bar.

Brush:Text Brush

When you select the text brush, you will be prompted for some text, which will then become the brush. You can then position the text, and press the right or left mouse to draw the text in the icon.

Currently, only upper and lower case alphanumerics are supported, and some of the letters don't quite look right.

Brush:Copy/Paste Brush

Right after you select the copy/paste brush, you will need to select an area of the current icon to "copy". This then becomes the brush, and you then draw the brush into the icon by pressing the right or left mouse buttons.

D.2.6 Zoom Menu

`Vicone` will display the icon zoomed from two up to 32 times the size of the actual icon. Use the zoom menu to select the zoom factor.

D.2.7 Tool Bars

There are two tool bars in the codeviconed interface. The first tool bar shows the current foreground and background colors on the left side. The next icon on this tool bar is the "snap" toggle. When pressed, drawing with the rectangle brush shape will draw squares, and the ellipse shape will draw circles. The next icon is the brush toggle, and will display the brush style dialog. The next icon is the color selection toggle, and will display the color selection dialog. These three toggles do not have corresponding entries on the menus. The right end of the first tool bar show three toggles to select the regular, text, and copy/paste brushes.

The second tool bar contains buttons for clear and undo, as well as toggles for selecting the brush shape. All these are duplicates of menu commands.

Color Selection Dialog

The color selection dialog is used to pick and select foreground and background colors. Internally, `viconed` uses 256 colors for each icon pixel. Depending on the color resolution of your display, all 256 colors may or may not be available. Typically, icons do not use very many colors, so this shouldn't matter.

The color selection dialog shows a large color square showing the current selection color. Two buttons next to the current color square are used to apply the current selection color to the foreground or background color.

Below that is a palette of 64 small color buttons arranged in 4 rows of 16. Selecting one of these buttons make it the current selection color. `Viconed` supplies 64 default colors for new icons. Again, depending on the color resolution of you color display, these may or may not show as 64 colors. When a new icon is loaded, its colors are used to load the color selection color palette. Most icons won't use 64 colors, and unused colors are filled with black.

Below the palette are three sliders that can be used to change the color. Select a palette button to change, then use the sliders to adjust the red, green, and blue. You can also press the small red, green, or blue button next to the sliders to enter a specific value for that color component. The reset button will reset the palette entry back to its original color.

Note that even though the color selection dialog only has 64 entries, the colors shown do not necessarily have any relation to the colors used in the icon. The colors in the icon are set by the foreground and background colors. You can use the sliders to specify any color, and then apply it to the foreground or background color. The color selection dialog allows you to easily pick any one of the 64 palette colors.

Use only standard, basic colors in icons (black, white, red, green, blue, etc.) to minimize the impact on the color maps used on systems with color resolutions of 256 or less.

Appendix E

Release Notes

V is usually found in one of several archives: `v-1.00.tar.gz`, the full distribution of all current versions of V; `vx-1.00.tar.gz`, the X only distribution; and `vwin100.zip`, the MS-Windows only distribution. Note the version (1.00) will change as V is updated.

E.1 X Window System

The current X implementation of ***V*** uses the Athena widget set with some modified versions of some widgets from the Xaw3d widget set. The code has about 95% of what is needed to work on Motif, but this effort has been largely abandoned. If someone out there *really* wants Motif, please work on it.

Directories

The ***V*** directory structure has been designed to allow you to either install ***V*** in a personal directory, or at a higher system level. By defining an appropriate search path in the `Makefile`, your applications can find the required ***V*** files.

You should download the standard distribution file `vx-1.00.tar.gz`[1], which is a `gzip` GNU Zipped tar file (or the full combined version, `v-1.00.tar.gz`). After you unzip the file (with `gunzip`, or the -z switch on some versions of `tar`), you can extract the file to wherever you want it. It will build a `/v` subdirectory from the directory containing the `vx-1.00.tar` file. The file hierarchy is:

/v The main ***V*** directory.

/v/appgen V application generator program.

/v/bccide MS-Windows build files for Borland C++. (Not included in X only distribution.)

/bmp2vbm Source for a simple MS-Windows and OS/2 `.bmp` bitmap format to `.vbm` V bit map format converter.

/v/bin The `/bin` directory is used to hold the binaries of ***V*** sample programs. No binaries are included on the distribution, but at least the subdirectories `/intel` for Linux and `/sun4` for Suns are included. There may be other subdirectories for other architectures.

[1] Or whatever the latest version is...

/v/doc The sources for ***V*** documentation. The documentation is in LaTeXformat. The standard distribution includes only the source LaTeXfiles. Other versions, including dvi and Postscript, are available from the ***V*** ftp site.

/v/draw Source for the VDraw example program. Examples that are identical across platforms use a `.cpp` file extension.

/v/drawex Very simple V draw example from C/C++ Users Journal article.

/v/examp Source for a simple ***V*** example.

/v/includex/v Source for the X `*.h` ***V*** header files.

/v/includew/v Source for the MS-Windows `*.h` ***V*** header files. (Not included on X only version.)

/v/lib Compiled version of the ***V*** library will be placed under appropriate subdirectories here. No precompiled libraries are included in the standard distribution.

/v/msvc Project and make files for Microsoft VC++.

/v/obj Compile object code is saved under here.

/v/srcx The full C++ source for the X ***V*** library. The files use a `.cxx` extension.

/v/srcwin The full C++ source for the Ms-Windows ***V*** library. The files use a `.cxx` extension. (Not on X only version.)

/v/test The test program used to test ***V*** functionality.

/v/texted Source for a simple editor based on vTextEdit class.

/v/tutor The source code for the tutorial example.

/v/vxgl Source for V canvas class that interfaces to X version of OpenGL.

/v/watcom Project and make files for Watcom C++ compiler.

Compilers

The `makefile` provided with ***V*** uses the GNU C++ compiler, `g++`. ***V*** does not use templates or other C++ features that can cause portability problems. The current version has been built and tested using `g++` Version 2.7.2.3, although it did work with Version 2.6.3, but not earlier versions. There is no inherent reason that ***V*** should not compile with other C++ compilers.

The X Makefile

The `Makefile` is the main way to build X versions of ***V***. It has comments that should help you to build the X version of ***V***. See the file README.TXT for more instructions for installing ***V***on a *nix platform. All of the customizations for a given platform have been isolated into the file [Config.mk].

To date, ***V*** has successfully been compiled on several platforms, including Linux, SunOS, Solaris, AIX, SGIs, and DEC Alphas. The standard distribution includes a `Makefile` that can be easily configured for several platforms. The makefile requires GNU make! The secret is to examine `Config.mk` and add and modify the definitions at the beginning as needed for your platform. (For Linux, this will usually be a no op, since Linux is the default configuration.) Examine the definitions already there, and then add a section with the locations defined as needed for your platform. Then use an `ARCH=` definition on the `make` line (or make your platform the default.)

X Resources

V makes limited used of X resources. The main use is to define the basic color schemes for controls and dialogs. The following resources are used:

vDialogBG The color used for the background of dialogs and command bars.

vStatusBarBG The color used for the background of the status bar.

vMenuBarBG The background color of the menu bar and menu drop downs.

vControlBG The background color for some controls, such as sliders and scroll bars.

vControlFace The color used for the faces of various controls such as buttons.

vLightControlShadow The color used for the light shadow on 3D controls.

vDarkControlShadow The color used for the dark shadow on 3D controls.

By varying just the above X resources, you can really change the visual look of your ***V*** app. The `/v/srcx` directory contains several files of the form `vRes*` that contain various color schemes. The default color scheme is contained in `vResDefault` (but you don't need to load it – it is the default). The file `vResBlueMtf` contains the color scheme similar to Motif. This is the contents of `vResDefault`:

```
*vDialogBG: gray75
*vStatusBarBG: gray80
*vMenuBarBG: gray70
*vControlBG: gray80
*vControlFace: gray70
*vLightControlShadow: gray87
*vDarkControlShadow: gray50
```

To use one of these, or your own, resource files, you can use the command `xrdb -merge vResColorscheme`. You can also add the lines to your `.Xresources` file.

The X program name is the name you supply to the `vApp` constructor.

X Bugs

The PostScript print driver does not draw shapes with hatched brushes.

The PostScript drawing canvas does not support `CopyFromMemoryDC`.

Source code uses two naming conventions - `.cxx` and `.cpp`. Gnu g++ version 2.6 and later support both file extensions. G++ version 2.5 doesn't like `.cpp`, so you might have to rename those files to `.cxx`,

There seems to be problems with colors on X Pseudocolor systems.

E.2 Microsoft Windows

The current implementation of ***V*** for MS-Windows is for Windows 3.1 and WIN32 (Windows 95 and NT). We will refer to this version as `Vwin` in this description. Eventually some native Windows 95 controls will be used, but for now the WIN32 version works fine for Win95. The Windows version of ***V*** is available at as [vwin100.zip] (a MS-Windows Zip format file). on the ***V*** ftp site. You will need a version of `ZIP` or `gunzip` and `tar` to extract ***V***. The MS-Windows version does not include documentation. Documentation is available as both LaTeXfiles in the X distribution, or split into a pair of Postscript or DVI files on the ***V*** ftp directory.

Directories

The directory structure of ***V*** under MS-Windows is similar to the X version. On the distribution, the MS-Windows hierarchy is found under the `/v` directory. (We will use Unix / notation for files instead of the usual MS-Windows backslash notation. Most MS-Windows compilers handle the / correctly, and / is used throughout the ***V*** source files.) When you unzip the archive, a subdirectory `/v` will be built.

Under `/v` are `/bin/win` for the example ***V*** MS-Windows binaries, `/draw` for the VDraw example program, `/examp` for a simple example program, `/includew/v` for the ***V*** `.h` header files, `/lib/win` for the MS-Windows compiled library, `/obj/win*` for the object files, `/srcwin` for the MS-Windows version of the source code, `/test` for the test driver program, and `/tutor` for the source code to the tutorial included in this reference manual.

For MS-Windows, the ***V*** library source files use a `.cpp` extension. The example programs also use `.cpp`. The source for most of the example programs is identical for the MS-Windows and X versions! However, the source for the library `.cpp` and `.h` files are different for each platform, so you must be careful not to mix the X and MS-Windows versions of source code and header files.

Compilers

V has been successfully been compiled using Borland C++ 4.5 for Win3.1 and WIN32; Borland C++ 5.02 for WIN32; Watcom 10.6 for Win3.1 and WIN32; the GNU-WIN32 gnu g++ compiler; and Microsoft VisualC++ under several versions.

Several Borland `.ide` files are included on the directory `/vwin/bccide`. The `.ide` files assume ***V*** is built on drive C:, so you may have to modify it if you want to build ***V*** on your own system. If you are using another compiler, then you need to compile *every* `.cpp` file found on the `/srcwin` directory.

Project files for compiling with Watcom C++ are included in the directory `v/watcom`. Unlike the Borland versions, the object code and libraries are built directly on these `watcom` directories.

The required changes and makefiles required for the `mingw32` compiler will be made available on the ***V*** web site.

MDI/SDI Models

V for MS-Windows supports both the MS-Windows MDI and SDI models. By default, ***V*** uses MDI, and will bring up the main MDI window, and open the first MDI child window. There currently is no way to have a main MDI window with no active MDI child windows – when you exit the last window, the application closes. The menu, command bar, and status bars will change to the ones defined by each child window as each child window is activated.

V will automatically append a `Window` menu item to the main menu. The built in `Window` menu supports the standard cascade and tile MDI operations, as well as showing a list of MDI children.

You can also get MS-Windows applications to look like the standard SDI model. If you want an SDI app, you control this in the static declaration of the `vApp` object:

```
static testApp* tApp = new testApp("Vtest",1);
```

The second parameter controls MDI or SDI. A default parameter is defined by ***V*** as 0 to indicate the MDI model. If you specify a 1, then ***V*** will take an SDI look. It actually does this by using the MDI code, but maximizing the canvas window, removing the extra buttons from the menu bar, and not adding the `Window` menu. It is impossible for the user to tell that this is really an MDI application, but ***V*** does not strictly enforce this. If you create more than a single `vCmdWindow` object, unpredictable things will happen under the SDI simulation. It is up to you to not do that.

Since X doesn't have an MDI/SDI equivalent, it is harmless to specify SDI to an X version of your app.

Icons

As stated in the main part of this manual, ***V*** does not use resource files. This is true for the MS-Windows versions. However, there is one reason you might want to include a `.RC` file with a ***V*** MS-Windows application, and that is to allow you to define the icons used with the application. (These are MS-Windows icons, and are *not* the same things as `vIcons`.)

Typical MS-Windows MDI applications use two icons - one for when the whole application is iconized, and one when each child window is iconized. If you don't supply a `.RC` file, you will get the default MS-Windows icons. The ***V*** distribution supplies two default icons of its own, called `vapp.ico` and `vwindow.ico`. By including the definitions `vAppIcon ICON vapp.ico` and `vWindowIcon ICON vwindow.ico` in the `.RC` file, ***V*** will load and use those icons for the application and each child window respectively. You can substitute whatever two icons you want for your application by specifying different `.ico` files for the `vAppIcon` and `vWindowIcon` names in the `.RC` file.

DEF File

MS-Windows applications are typically compiled using a `.DEF` file. You can modify any of the `.DEF` files included with ***V*** sample programs.

Bugs on MS-Windows

E.3 V Versions

Version 1.00 This version was local to the University of New Mexico on January 10, 1996. Versions 1.01, 1.02, and 1.03 were local maintenance releases.

Version 1.04 This was the first major public release of ***V***, and was announced to the world on February 14, 1996.

Version 1.05 This version had several bug fixes obtained from feedback of the public release.

Version 1.06 This was an X only release, and added 3D controls.

Version 1.07 This release was never formally announced, and included some of the changes listed for version 1.07.

Version 1.08 The 4/15/96 release added several significant features to ***V***:

- The `vMemoryDC` drawing canvas, including new methods `CopyFromMemoryDC` and `DrawColorPoints`.
- Internal revisions for handling of color, including adding `vColor::ResetColor` to allow reuse of color maps, and `vColor::BitsOfColor` to get color capability. These revisions allow ***V*** apps to make more effective use of default color maps.
- `vPen::SetColor(r,g,b)` and `vBrush::SetColor(r,g,b)` are being dropped in favor of the `vColor` forms. These calls break the hidden management of color maps, and while still included in the code, should *not* be used. Support for the `(r,g,b)` form will be dropped entirely in future versions of ***V***.
- `C_ToggleIconButton` was added to allow a pressed in button interface look in place of check boxes and radio buttons.
- Documentation for `ChangeColor` and `C_ColorLabel` was added, although the functionality has been there for a while.
- The WIN32 port was finished, and the X and MS-Windows versions are now in sync.

Version 1.09 Added `C_ToggleButton` and `C_ToggleFrame` controls. It also includes a large number of ***V*** icons suitable for building command pane tool bars.

Version 1.10 The 5/29/96 release of ***V*** includes the following enhancements and changes:

- The ***V*** Icon Editor - an icon editor to create icons for various ***V*** controls.
- Inclusion of accelerator key support in menus.
- Addition of the `ChangeListPtr` set type to allow completely dynamic lists, combo boxes, and spinners.
- Addition of `DrawLines`, `DrawPoints`, and `DrawRectangles` to `vCanvas`.
- Several bug fixes for both MS-Windows and X, some relatively major.
- The canvas page scroll messages were changed on the X version to correspond to the behavior of the MS-Windows version. A page scroll message is sent only at the completion of a scroll, not continuously as before. It is usually rather difficult to implement nice continuous scrolling, so this approach seems more useful to more people. This is the only known change that might affect compatibility with previous ***V*** applications.
- Addition of a directory for outside contributions.

Version 1.11 The 7/4/96 release of ***V*** has several minor bug fixes for the MS-Windows and X versions. It also adds the `WorkSlice` methods to support applications that require computations to continue even if the user is not entering commands to the application.

Version 1.12 This was a bug fix release for MS-Windows. The X version was unchanged, but renumbered for consistency.

Version 1.13 This 8/24/1996 release of ***V*** is a major release with several new features, and some significant bug fixes that can change the behavior of existing ***V*** applications. The following includes a list of changes:

- The ***V*** Application Generator, `vgen` is now included with the standard distribution. It will generate a simple ***V*** application as a starting skeleton for new apps.
- The values being passed by ***V*** to `vCanvasPane::VPage` and `HPage` were incorrect on the X version. The documentation states that the values for `Top` should be in the range 0 to 100. The MS-Windows version worked correctly, but the X version was passing a range of 0 to (100-Shown). This bug actually has been in the X version since the switch to 3D Controls. With version 1.13, both MS-Windows and X work the same.
- The MS-Windows version of `vDC::DrawText` was fixed to work according to the documentation. It had been drawing text with the x,y as the upper left corner of the text. Beginning with 1.13, it now draws at the lower left corner as specified in the documentation.
- Two functions, `GetHScroll` and `GetVScroll`, were added to `vCanvasPane` to make dealing with scroll bars easier.
- A new standard using `enums` for generating IDs for controls has been adopted beginning with 1.13.
- `C_TextIn` controls now allow you to specify the width of the control in characters using the `size` field. This is described in Chapter 6.
- In `vCanvasPane`, new parameters (with default values for backward compatibility) were added to `CopyFromMemoryDC` to allow subregions to be copied.
- Using a `vTransparent` pen when drawing text now results in leaving the existing background when drawing, and a `vSolid` pen overwrites with the current background color.
- There was a conflict on MS-Windows with using `VK_` for key names. The MS-Windows version was changed long ago, and now the X version also uses lower case letters (e.g., `vk_Tab`).

Version 1.14 The major addition to 10/6/96 ***V*** Release 1.14 is the addition of the `vTextEditor` class, which is a very good first pass at a complete editing canvas. The editor is complete, can be extended to support custom command sets or file management. It is missing cut, copy, and paste, which will be implemented as general support for these is added to ***V***. The code for `vTextEditor` is based on `vTextCanvasPane`, and is identical for the X and MS-Windows versions.

Also, for the X Version, support for OpenGL has been added. This support is found in the distribution directory `v/vxgl`. While the ***V*** OpenGL canvas pane seems very robust, it is still somewhat experimental. I would like any feedback on its use and design.

Other changes, mostly bug fixes, include:

- X version: The little close button on the left of the menu bar has been dropped by popular request. It seems most people didn't like it. If you do, you can still get it by defining the symbol USE_CLOSE_BUTTON. Instead, ***V*** now supports the X WM_DELETE_WINDOW protocol. This protocol is supported slightly differently by different window managers, but accomplishes the same thing as the old close button.
- X version: There was a minor bug in how the scroll bars worked when `top == 0`.

- X version: The method used to get the size of a window was changed, and should now give correct values.
- X version: There was a bug in drawing radio buttons that only showed up on some systems.
- X version: There was a bug in changing the current selection in combo boxes.
- X version: There was a bug in setting colors for the PostScript DC.
- X and MS-Windows: There were several bugs in `vTextCanvasPane` exposed by the implementation of `vTextEditor`.
- X version: There was a bug in the key mapping that would cause a program to terminate if an unrecognized key was pressed.
- MS-Windows: The method to determine the size of the MDI frame and client windows was improved (I hope!).
- MS-Windows: A bug with the work timer was fixed. The interaction between the work timer, check events, and the MS-Windows message loop was changed to work better.
- MS-Windows: The argument order of ClearRect was fixed to correspond to X and the documentation.
- MS-Windows: There was a bug that didn't allow SetValue to work correctly for some controls.
- MS-Windows: A bug in handling the MS-Windows caret in text canvases was fixed. This one was a bit subtle, but nasty in possible side effects. Also, EnterFocus and LeaveFocus did not work correctly.
- MS-Windows: A bug in setting text colors on NT and Windows 3.1 was fixed. The bug did not manifest itself on Windows 95.

Version 1.15 Release of V Version 1.15 has some non-backward compatible changes. In previous versions of V, there were inconsistencies in the order of width and height parameters. These have all been now changed to consistently use a width/height order. (Except for vIcon, which still use height/width.) The decision to fix this order came from a general consensus of the V mail list.

You will need to change your code to reflect the new changes. The following things must be changed:

1. Any calls to the constructor of a base or derived vCmdWindow will need the width and height order swapped.

2. Calls or overrides of vApp::NewAppWin need the order of width and height swapped.

3. Calls to vCanvasPane::SetHeightWidth(h,w) need to be changed to vCanvasPane::SetWidthHeight(w,h).

4. Calls or overrides of all versions of Redraw(x,y,h,w) need to be changed to Redraw(x,y,w,h).

5. Calls or overrides of all versions of Resize(h,w) need to be changed to Resize(w,h). (The vTextCanvas row/column versions retain their row/column order.)

Also, the makefiles have been revised for more flexible building on different *nix platforms.

A new method, vDialog::DialogDisplayed has been added to allow dynamic setting of dialog control values.

Version 1.16 Version 1.16 has no significant changes in V functionality. It mostly has some bug fixes. The only major change is the release of a completely new set of Makefiles for the Unix version. These new makefiles were contributed by a V user, and are much cleaner than the old versions.

A summary of the changes:

- A small change to the code generated by vAppGen.
- A fix to scrolling in the V Icon Editor.
- Some changes to the v_defs.h file for MS-Windows, including compatibility changes needed for Microsoft VC++. Project files were added for MSVC++.
- The == and != operators for brushes, fonts, pens, and colors were changed to use reference parameters consistently.
- Various minor changes to enhance compiler compatibility, both on MS-Windows and X.
- VReply was fixed to work over multiple shows.
- A void* was added to vAppWinfo.
- Vmemdc had height and width switched.
- Sizing of buttons on MS-Windows was fixed for Windows 95.
- A resource leak was fixed for MS-Windows.
- A major bug that showed up only under Microsoft VC++ was fixed.
- Initialization of text in strings was fixed for MS-Windows.
- Changing the values of radio buttons on MS-Windows now works.
- Spinners now honor the size specification.
- A tab keystroke now works correctly on MS-Windows.
- A bug in
- Various new tests were added to the test program.
- A couple of bugs were fixed in the X OpenGL V interface.

Version 1.17 Version 1.16 has proven to be remarkably stable. A few minor bugs have been reported and corrected for Version 1.17. Some enhancements have been added, the most significant allows you to specify the number of rows displayed in a list box.

A summary of the changes:

- Lists by default display 8 rows, buy you can now specify and size between 1 and 32 rows.
- A bug in Windows when closing multiple windows was fixed.
- The vReply dialog has been changed to allow a default string in the input field.
- Direct printing to `lpr` has been added to the X version.
- The PostScript driver was modified to print better lines.

Version 1.18 Release 1.18 has a major enhancements to ***V***. The main addition is support for Tool Tips - little boxes with text info that are automatically displayed when the user holds the mouse over a control. Tool Tip support is *very* trivial to add to your programs, and greatly enhances the user interface.

There are some minor enhancements, and a few bug fixes. Beginning with the 1.18 release, there will be a separate document that summarizes the changes for that version. This will simplify the upgrade path for past users.

Version 1.19 Release 1.19 has some minor enhancements, and a few bug fixes. The main enhancement is the addition of password support for text in controls. There have also been some improvements to `vgen`, especially for support for the `mingw32` MS-Windows compiler. The support for the `mingw32` compiler has been improved.

E.4 Future Plans

- An OS/2 Port is released. Watch the `www.objectcentral.com` home page for more news. It is not up to version 1.19.
- I plan to add multi-selection list support for 1.20.
- I plan to release a ***V*** based IDE (Integrated Development Environment). The ***V*** IDE will support gcc on both Linux and MS-Windows.
- I plan to add MS-Windows support for OpenGL.
- I hope to port ***V*** to `gtk` for version 1.21. This version will not have major enhancements to ***V***. However, lifting the Athena restriction will allow many new features to be added.
- ***V*** still needs a class to portably interact with the standard system clipboards.

Appendix F

The Latest Version: What's New?

This Appendix will cover the latest version of ***V***. The current release is Version 1.18.

F.1 New Features

New Features for C_List

The number of rows displayed can now be controlled by using the `CommandObject` element `size`. By specifying the attribute `CA_Size` and providing a value for the `size` element, you can specify how many rows to show. If you don't specify a size, 8 rows will be displayed. ***V*** will support between 1 and 32 rows. Note the that the `size` element is the last one of a `CommandObject`, and can left out of a declaration, which results in the compiler generating a zero value, giving the default 8 rows.

The width in pixels (approximately) of the list can be controlled by specifying the `CA_ListWidth` attribute and providing a value to the `retVal` parameter, which is otherwise unused for a list object. This implementation isn't perfect - you may have to play with the interaction between the width you specify, and the font used in a list control.

Tool Tips

The major new feature of ***V*** 1.18 is support for Tool Tips. You can easily add Tool Tips by adding the appropriate text to your existing `CommandObject` definitions of tool bars and dialogs. The new definition of a `CommandObject` follows:

```
typedef struct CommandObject
  {
    CmdType cmdType;    // what kind of item is this
    ItemVal cmdId;      // unique id for the item
    ItemVal retVal;     // initial value of object
    char* title;        // string
    void* itemList;     // used when cmd needs a list
    CmdAttribute attrs; // list of attributes
    int Sensitive;      // if item is sensitive or not
    ItemVal cFrame;     // Frame used for an item
    ItemVal cRightOf;   // Item placed left of this id
    ItemVal cBelow;     // Item placed below this one
    int size;           // Used for size information
    char* tip;          // ToolTip string
  } CommandObject;
```

char* tip The tip parameter is used to specify an optional ToolTip string for use with a command object. If you provide a string here, that string will be automatically displayed after the user holds the mouse over that control. The exact delay before the tip is shown, and the format of the tip box is somewhat platform dependent, and all platforms might not support tool tips. (Currently, only OS/2 does not support tips.) Note that if you use a tip, you must be sure to include a value (usually 0) for the size parameter!

void vBeep()

This utility routine will sound an audible beep.

void vGetcmdIdIndex(ItemVal cmdId, CommandObject *cmdObj)

Sometimes when you work with a CommandObject array to define a dialog, you need to access the elements of a particular item in the array. This is especially true for manipulating lists. This routine will return the index into a CommandObject array of an entry with the supplied `ItemVal cmdId`.

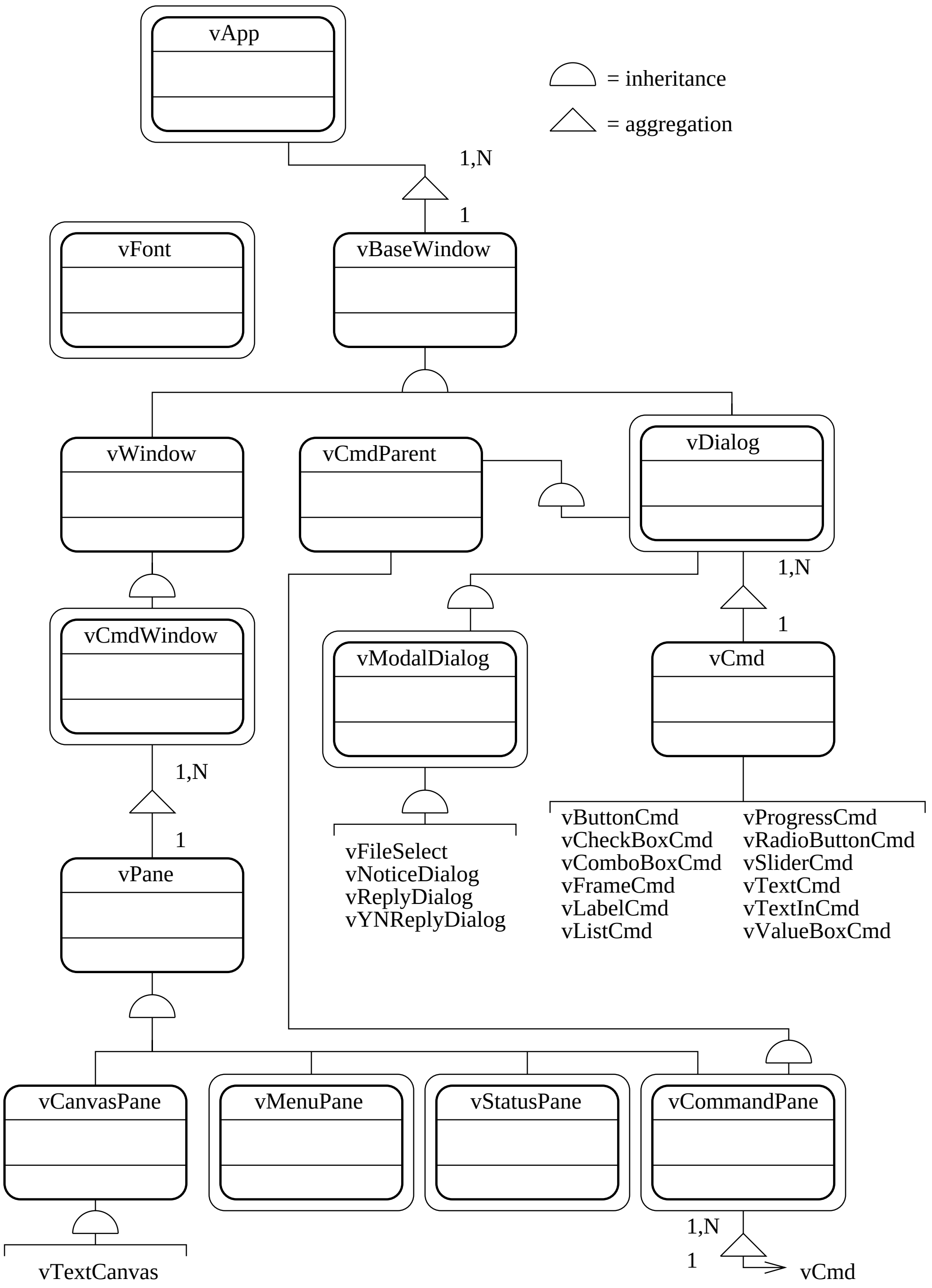
Appendix G

V Class Hierarchy

The following figure is shows the internal organization of the ***V*** class hierarchy. Note that boxes with a double line edge denote classes that have object instances, while boxes with single line edges are abstract classes used to build subclasses. Most of the time, you won't care about these abstract classes.

Also note that the classes derived from `vCmd` represent the classes used to implement command objects. Normally, you won't need to used these classes directly. Instances of these objects are generated by ***V*** from your `CommandObject` declarations when you call `AddDialogCmds`.

The V GUI Class Hierarchy



V - A Portable C++ GUI Framework

Index