UNIX VISION

An object oriented user interface for UNIX

Zoran Constantinescu
"Politehnica" University of Bucharest
<pr92044@ulise.cs.pub.ro>

Constantin Stanciu
"Politehnica" University of Bucharest
<pr92182@ulise.cs.pub.ro>

Abstract

The paper presents a project we started to work on. The main purpose followed is the implementation of a friendly interface under the LINUX operating system. The general objective of this interface is the UNIX and DOS portability, being written in C++ and using the ncurses library.

UNIX VISION is indicated for text based applications that need high design performance, flexibility, and consistent interactive user interface. It consists from a complete object oriented library including multiple, resizeable, overlapping windows, pull-down menus, dialog boxes, buttons, scroll bars, input lines, check boxes, radio buttons etc. ***UNIX VISION*** allow development of complex applications which need object management, easily implemented using the C++ language stream facilities.

Introduction

UNIX VISION is an application framework for windowing programs. It is implemented as an object oriented user interface. Object orientation is becoming increasingly popular in programming language domain, where it helps to face up the growing complexity of software. Also it prompts to design applications as a set of functional units interacting with each other through well defined interfaces. So it makes partitioning of an application into several modules easier, relieves the developer of having to know implementation details of objects to access them. This application save an enormous amount of unnecessary, repetitive work, and provide a proven application framework you can trust.

UNIX VISION is a *hierarchy*, not just a disjoin box full of tools. If you use any of it all, you should use *all* of it. There is a single architectural vision behind every component of ***UNIX VISION***, and they all work together in many subtle, interlocking ways. You shouldn't try to just "pull out" mouse support and use it - the "pulling out" would be more work than writing your own mouse binding from scratch.

Implementation

The idea of ***UNIX VISION*** interface is based on **Turbo Vision**, being *fully compatible* with it. So, any DOS oriented Turbo Vision application, with minor changes, can easily be ported to UNIX, keeping the same aspect. In this mode, even a VT100 'dumb' terminal can display many windows on the screen.

To have fully compatibility for screen operation on different UNIX systems, we used the ncurses library. The ncurses library routines give the user a terminal independent method of updating character screens with reasonable optimization. The ncurses routines emulate the curses(3X) library of System V Release 4 UNIX. From this package we used terminal input, control over terminal, output options, color manipulation, terminfo capabilities and access to low-level ncurses routines.

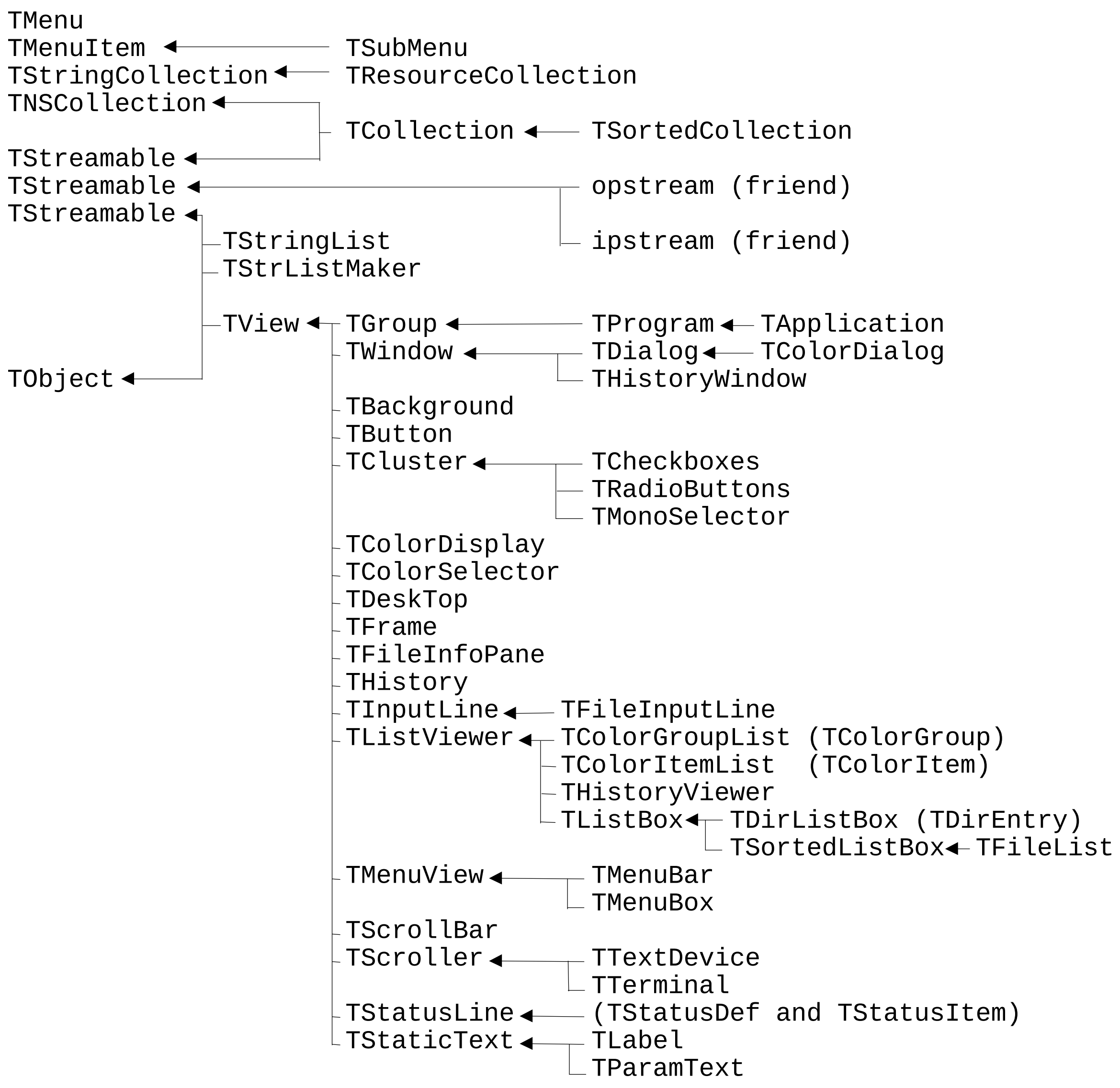
The ***UNIX VISION*** comes with full mouse support. It is activated whenever you are running on an xterm terminal (it even works if you take a telnet or rlogin connection to another machine from the xterm) or if you are running on a Linux console and have the *gpm* mouse server running.

To compile the ***UNIX VISION*** we used the GNU C++ compiler (version 2.6.3), which is designed to generate optimal assembly code and run on almost every UNIX system.

With ***UNIX VISION*** any kind of files can be easily manipulated. To do this we used GNU C++'s libg++ library (version 2.6.2) which includes the iostream classes. The iostream classes implement most of the features of AT&T version 2.0 iostream library classes, and most of the features of the ANSI X3J16 library draft (which is based on the AT&T design).

Internal design

The object hierarchy of ***UNIX VISION*** is:



Any application based on ***UNIX VISION*** is a cooperating society of *views*, *events*, and *mute objects*.

- **Views** - A view is any program element that is visible on the screen - and all such elements are objects. In a ***UNIX VISION*** context, if you can see it, it's a view. Fields, field captions, window borders, scroll bars, menu bars and dialog boxes are all views. Views can be combined to form more complex elements like windows, and dialog boxes. These collective views are called groups, and they operate together as though they were a single view.

- **Events** - An event is some sort of occurrence to which an application must respond. Event come from the keyboard, from the mouse, from other parts of ***UNIX VISION***, or from the kernel . For example. a keystroke is an event, as is a click of a mouse button or a received signal. Event are queued up by ***UNIX VISION***'s application skeleton as they occur, then they are processed in order by an event handler. The TApplication object, which is the body of an application, contains an event handle. Through a mechanism, events that are not serviced by TApplication are passed along to other views owned by the program until either a view is found to handle the event, or an "abandoned event" error occurs.

  There are five types of events:

  - *mouse events* - an up or down click with either button, a change of position, or an auto event when a button is hold down; all mouse events include the position of the mouse, so an object that preceded the event knows where the mouse was when it happened;
  - *keyboard events* - when a key is pressed;
  - *message events* - commands, broadcasts and user messages;
  - *signal events* - when the program receives any of the UNIX signals;
  - *"nothing" event.*

- **Mute objects** - Mute objects are any other objects in the program that are not views. They are "mute" because they do not speak to the screen themselves. They perform calculations, and generally do the work of the application. When a mute object needs to display some output on the screen, it must do so through the cooperation of a view.

At the heart of every view there is a loop that looks something like this:

```
{
  do {
       endState = 0;
       do  {
            TEvent e;
            getEvent( e );
            handleEvent( e );
            if( e.what != evNothing )
                eventError( e );
       } while( endState == 0 );
  } while( !valid(endState) );
    return endState;
}
```

Every view inherits a *handleEvent* method that already knows how to respond to much of the user's input. If we need a view to do something specific for the new application, we need to override its *handleEvent* and teach the new one how to respond to the received events. Events go from one object's event handler to another until some handler is not found which can process it. If the event is processed but not destroyed, it would be transmitted to another handler and so on.

The curent modal view's *getEvent* calls its owner's *getEvent* and so on, all the way back up the view tree to *TApplication.getEvent*. The main part of this method, as we used at the beginning is given next:

```
void TProgram::getEvent(TEvent& event)
{
  event.getKeyEvent();
  if ( event.what == evNothing )
    idle();
}
void TEvent::getKeyEvent()
{
  int x;
  x=getch();
  if (x==ERR)
    what = evNothing;
  else {
    what = evKeyDown;
    keyDown.keyCode = x;
  }
  return;
}
```

We used *getch()* from ncurses, setting first the terminal in nodelay mode, causing *getch()* to be a non-blocking call - if no input is ready, *getch()* will return ERR, and in nocbreak mode - characters typed are immediately available to the program (doesn't wait for newline). This method worked very well, but had a great disadvantage: the application used a lot of CPU time, generating many evNothing events, so executing the *idle()* method too many times.

Next, we wanted to have mouse in the application, so we decided to use LINUX's *gpm* library. To get mouse events, there is a function called Gpm_GetEvent(), which waits the mouse to generate some events, than returns. The problem was how to get both mouse and keyboard events using these functions.

The solution we found was to use the UNIX *select()* system call, that allows device polling. This call causes the process to sleep until one of the selected devices becomes available for reading, or until the time limit, set by timeout, has elapsed. The resulting *getEvent()* was:

```
void TProgram::getEvent(TEvent& event)
{
  ....
  FD_SET(stdin,&select_set);
  if ( mouse )
    FD_SET(mouse,&select_set);
  ::select( FD_SETSIZE,&select_set,NULL,NULL,&timeout );
  if ( FD_ISSET(mouse,&select_set) )
    event.getMouseEvent();
  else if ( FD_ISSET(stdin,&select_set) )
    event.getKeyEvent();
  else { event.what=evNothing; idle(); }
  ....
}
```

Application development facilities

UNIX VISION has built-in tools that help you implement *context-sensitive help* within your application. You can assign a help context number to a view and whenever that view becomes focused, its help context number will become the application's current help context number. The THelpViewer can read and display the proper help text.

Streams provide a simple, yet elegant, means of storing object data outside the program. A stream is a generalized object for handling input and output. TStreamable is the base abstract object providing polymorphic I/O to and from a storage device. What you intend to send to a stream doesn't have to be determined at compile time. The streams know they are dealing with objects, so as long as the object is a descendant of Tobject, the stream can handle it. In fact, different objects can as easily be written to the same stream as a group of identical objects. A *resource* file is a special kind of stream where generic objects ("items") can be indexed via string keys, and later accessed with the appropriate key.

Conclusions and future work

The first version of ***UNIX VISION*** library is operational. It was developed on LINUX and the results are very good. Further work will be oriented toward developing a graphical interface for LINUX, using for example de SVGA library and also porting it to other UNIX systems.

References

[1.] The GNU C++ library documentation
[2.] The ncurses library documentation
[3.] Maurice J. Bach - The Design of the UNIX® Operating System