Programming with Haiku

Lesson 22

Written by DarkWyrm

Going Deeper: QuickEdit

If only all computer programs were as simple as HelloWorld and other demo programs! Life for developers would be much, much easier. Alas, they are not. For the next few lessons, we will be delving a bit more below the surface than usual. Designing simple programs for any operating system is as trivial as the programs themselves and are often coded very sloppily. Hopefully you will get an idea of what is involved in writing larger Haiku programs as we learn other features of the operating system.

What is our project going to be, you ask? A simple styled text editor called QuickEdit. Although Haiku already has one in StyledEdit and, as such, we will undoubtedly write some code which will be quite similar to it, not all of the features we may add can necessarily be found in StyledEdit.

Program Architecture

Were it not for one particular control, BTextView, our job would be much, much more difficult. Perhaps ironically, word processors are not easy to write. Our program is pretty simple from a "big picture" perspective: wrap a relatively simple GUI around a BTextView to make a simple, but useful, word processor.

Before we dive headlong into some code, let's take a quick glance at the class definition for a BTextView to see what features we can provide without spending a great deal of time. This will likely save us some effort and time later on. Note that this is not the complete definition for the class – the extra bits unrelated to features for QuickEdit have been left out.

```
class BTextView : public BView
{
public:
	void			SetText(const char* inText,
					const text_run_array* inRuns = NULL);
	void			SetText(const char* inText, int32 inLength,
					const text_run_array* inRuns = NULL);
	void			SetText(BFile* inFile, int32 startOffset,
					int32 inLength,
					const text_run_array* inRuns = NULL);

	void			Insert(const char* inText,
					const text_run_array* inRuns = NULL);
	void			Insert(const char* inText, int32 inLength,
					const text_run_array* inRuns = NULL);
	void			Insert(int32 startOffset,
					const char* inText, int32 inLength,
					const text_run_array* inRuns = NULL);

	void			Delete();
	void			Delete(int32 startOffset, int32 endOffset);

	const char*		Text() const;
	int32			TextLength() const;
	void			GetText(int32 offset, int32 length,
					char* buffer) const;
	uint8			ByteAt(int32 offset) const;

	int32			CountLines() const;
	int32			CurrentLine() const;
```

```
void                GoToLine(int32 lineIndex);

void                Cut(BClipboard* clipboard);
void                Copy(BClipboard* clipboard);
void                Paste(BClipboard* clipboard);
void                Clear();

bool                AcceptsPaste(BClipboard* clipboard);
bool                AcceptsDrop(const BMessage* inMessage);

void                Select(int32 startOffset, int32 endOffset);
void                SelectAll();
void                GetSelection(int32* outStart,
                                 int32* outEnd) const;

void                SetFontAndColor(const BFont* inFont,
                                    uint32 inMode = B_FONT_ALL,
                                    const rgb_color* inColor = NULL);
void                SetFontAndColor(int32 startOffset,
                                    int32 endOffset,
                                    const BFont* inFont,
                                    uint32 inMode = B_FONT_ALL,
                                    const rgb_color* inColor = NULL);

void                GetFontAndColor(int32 inOffset,
                                    BFont* outFont,
                                    rgb_color* outColor = NULL) const;
void                GetFontAndColor(BFont* outFont,
                                    uint32* sameProperties,
                                    rgb_color* outColor = NULL,
                                    bool* sameColor = NULL) const;

void                FindWord(int32 inOffset,
                             int32* outFromOffset,
                             int32* outToOffset);

float               LineWidth(int32 lineIndex = 0) const;
float               LineHeight(int32 lineIndex = 0) const;
float               TextHeight(int32 startLine,
                               int32 endLine) const;

void                ScrollToOffset(int32 inOffset);
void                ScrollToSelection();

void                Highlight(int32 startOffset,
                              int32 endOffset);

void                SetTextRect(BRect rect);
BRect               TextRect() const;
void                SetInsets(float left, float top, float right,
                              float bottom);
void                GetInsets(float* _left, float* _top,
                              float* _right, float* _bottom) const;

void                SetStylable(bool stylable);
bool                IsStylable() const;

void                SetTabWidth(float width);
float               TabWidth() const;
```

```
		void			SetWordWrap(bool wrap);
		bool			DoesWordWrap() const;

		void			SetMaxBytes(int32 max);
		int32			MaxBytes() const;

		void			DisallowChar(uint32 aChar);
		void			AllowChar(uint32 aChar);

		void			SetAlignment(alignment flag);
		alignment		Alignment() const;

		void			SetAutoindent(bool state);
		bool			DoesAutoindent() const;

		void			SetColorSpace(color_space colors);
		color_space		ColorSpace() const;

		void			MakeResizable(bool resize,
						BView* resizeView = NULL);
		bool			IsResizable() const;

		void			SetDoesUndo(bool undo);
		bool			DoesUndo() const;

		void			HideTyping(bool enabled);
		bool			IsTypingHidden() const;

		void			Undo(BClipboard* clipboard);
		undo_state		UndoState(bool* isRedo) const;

};
```

Some rather interesting possibilities open up to us just by looking at these methods. We can offer Undo and options to wrap or not wrap text to the window. The width of the space occupied by tab stops can be changed, giving us a little control. It might be possible to use the text rectangle features as something of a page margin setting. Font color and styles can be set, and we should not forget that there are many more effects for fonts than what are listed here, such as underlining, bold and italic styles, outlining, and more. Clipboard operations can be performed. There is also some basic alignment support, although this is not nearly as flexible as in regular word processors.

Despite all of the possibilities, let's start simple. As long as we try to follow the basic workflow conventions of word processors in general, it's much easier to start somewhere and build from there than try to figure out exactly how everything will fit in ahead of time.

For our initial effort on this word processor, we will aim just beyond concepts we already know: a window with a text editor inside it which can save and load text. Create a new project in Paladin using the *Main Window with Menu* template. Change the application signature in App.cpp to `"application/x-vnd.dw-QuickEdit"` – substituting your own initials for that of the author's – and save your changes. You'll also need to go into Change System Libraries in the Project menu and add the libraries libtracker.so and libtranslation.so. App.h and App.cpp should look like this:

App.h

```
#ifndef APP_H
#define APP_H

#include <Application.h>

class App : public BApplication
{
public:
	App(void);
};

#endif
```

App.cpp

```
#include "App.h"
#include "MainWindow.h"

App::App(void)
	:	BApplication("application/x-vnd.dw-QuickEdit")
{
	MainWindow *mainwin = new MainWindow();
	mainwin->Show();
}


int
main(void)
{
	App *app = new App();
	app->Run();
	delete app;
	return 0;
}
```

None of this is particularly exciting or special so far. All of the interesting stuff goes in MainWindow.h and MainWindow.cpp.

MainWindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <Window.h>

#include <Entry.h>
#include <FilePanel.h>
#include <MenuBar.h>
#include <String.h>
#include <TextView.h>

class MainWindow : public BWindow
{
public:
				MainWindow(void);
				~MainWindow(void);
```

```
                    void        MessageReceived(BMessage *msg);
                    bool        QuitRequested(void);

                    void        OpenFile(const entry_ref &ref);
                    void        SaveFile(const char *path);
                    void        FrameResized(float w, float h);
private:
                    void        UpdateTextRect(void);

                    BMenuBar    *fMenuBar;
                    BTextView   *fTextView;
                    BFilePanel  *fOpenPanel,
                                *fSavePanel;

                    BString     fFilePath;
};

#endif
```

New in this header file are the header files for the BFilePanel class, which we will use for loading and saving files, and the BTextView class which will provide all of the text editing services that we will need.

MainWindow.cpp

```
#include "MainWindow.h"

#include <Alert.h>
#include <Application.h>
#include <Directory.h>
#include <File.h>
#include <Menu.h>
#include <MenuItem.h>
#include <Messenger.h>
#include <NodeInfo.h>
#include <Path.h>
#include <ScrollView.h>
#include <String.h>
#include <TranslationUtils.h>

enum
{
      M_FILE_NEW = 'flnw',
      M_SHOW_OPEN = 'shop',
      M_SAVE = 'save',
      M_SAVE_AS = 'svas',
      M_PRINT_SETUP = 'ptcf',
      M_PRINT = 'prin'
};

MainWindow::MainWindow(void)
      :     BWindow(BRect(100,100,500,400), "QuickEdit", B_TITLED_WINDOW,
                  B_ASYNCHRONOUS_CONTROLS)
{
      // We will first start with creating the menu bar and populating the
      // menu with commands that we will be implementing.
      BRect r(Bounds());
      r.bottom = 20;
```

```
fMenuBar = new BMenuBar(r,"menubar");
AddChild(fMenuBar);

BMenu *menu = new BMenu("File");
fMenuBar->AddItem(menu);

// This is a shorthand way of adding new items to a menu. Note that
// the key accelerators are standards for Haiku:
//          Alt + N => New file
//          Alt + O => Open file
//          Alt + S => Save file
//          Alt + Shift + S => Save As

// It is also worth noting that key accelerators are normally paired
// with the Alt key, but it is also possible for Control to be
// the command key just as it is used in Windows and Linux.
menu->AddItem(new BMenuItem("New", new BMessage(M_FILE_NEW), 'N'));
menu->AddItem(new BMenuItem("Open", new BMessage(M_SHOW_OPEN), 'O'));
menu->AddSeparatorItem();
menu->AddItem(new BMenuItem("Save", new BMessage(M_SAVE), 'S'));

// This version, unlike the others above, specifies a key accelerator
// which uses more than just Alt plus a letter.
menu->AddItem(new BMenuItem("Save As" B_UTF8_ELLIPSIS,
            new BMessage(M_SAVE_AS), 'S',
            B_COMMAND_KEY | B_SHIFT_KEY));

// Now we will add the text view and scrollbars. To do this we will
// use a BScrollView for convenience. When working with a
// BScrollView, you create its target first, create the BScrollView
// second, and then call AddChild only for the BScrollView -- it will
// take care of adding its target for you.
r = Bounds();
r.top = fMenuBar->Frame().bottom + 1;

// When calculating the size of the target view, you must compensate
// for the extra width and/or height of the scrollbars provided by
// BScrollView. Haiku gives us the constants B_V_SCROLL_BAR_WIDTH and
// B_H_SCROLL_BAR_HEIGHT for this.
r.right -= B_V_SCROLL_BAR_WIDTH;

// The BTextView constructor we will be using expects a frame size
// but it also requires a rectangle for the area where the text will
// appear. This more or less sets the margins.
BRect textRect = r;
textRect.OffsetTo(0,0);
textRect.InsetBy(5,5);
fTextView = new BTextView(r, "textview", textRect, B_FOLLOW_ALL);

// Without this call, our BTextView will only be a plain text editor
fTextView->SetStylable(true);

BScrollView *scrollView = new BScrollView("scrollview", fTextView,
                          B_FOLLOW_ALL, 0, false, true);
AddChild(scrollView);

// Here we will use a new class: BFilePanel. More on this later.
BMessenger msgr(NULL, this);
fOpenPanel = new BFilePanel(B_OPEN_PANEL, &msgr, NULL, 0, false);
```

```
	fSavePanel = new BFilePanel(B_SAVE_PANEL, &msgr, NULL, 0, false);

	// This will make it possible for the user to start typing as soon as
	// the program is running. Without this call, the user must manually
	// click on the window in order to type. Talk about annoying!
	fTextView->MakeFocus(true);
}


MainWindow::~MainWindow(void)
{
	delete fOpenPanel;
	delete fSavePanel;
}


void
MainWindow::MessageReceived(BMessage *msg)
{
	switch (msg->what)
	{
		case M_FILE_NEW:
		{
			// Erase all text in the BTextView. Reset the file
			// path to empty to signal that the file has not been
			// saved to disk.
			fTextView->SetText("");
			fFilePath = "";
			break;
		}

		// These cases handle messages related to opening and saving
		// files.
		case M_SHOW_OPEN:
		{
			fOpenPanel->Show();
			break;
		}
		case B_REFS_RECEIVED:
		{
			entry_ref ref;
			if (msg->FindRef("refs", &ref) != B_OK)
				break;
			OpenFile(ref);
			break;
		}
		case M_SAVE:
		{
			if (fFilePath.CountChars() < 1)
				fSavePanel->Show();
			else
				SaveFile(fFilePath.String());
			break;
		}
		case M_SAVE_AS:
		{
			fSavePanel->Show();
			break;
		}
```

```
		case B_SAVE_REQUESTED:
		{
			entry_ref dir;
			BString name;
			if (msg->FindRef("directory", &dir) == B_OK &&
				msg->FindString("name", &name) == B_OK)
			{
				BPath path(&dir);
				path.Append(name);
				SaveFile(path.Path());
			}
			break;
		}
		default:
		{
			BWindow::MessageReceived(msg);
			break;
		}
	}
}


bool
MainWindow::QuitRequested(void)
{
	be_app->PostMessage(B_QUIT_REQUESTED);
	return true;
}


void
MainWindow::OpenFile(const entry_ref &ref)
{
	// Convert any symlinks into their targets.
	BEntry entry(&ref, true);
	entry_ref realRef;
	entry.GetRef(&realRef);

	// The Translation Kit provides translation services for text files,
	// too.
	BFile file(&realRef, B_READ_ONLY);
	if (file.InitCheck() != B_OK)
		return;

	// One simple function to read styled text from a file. Nice!
	if (BTranslationUtils::GetStyledText(&file, fTextView) == B_OK)
	{
		// BPath gives us a bridge between the data classes used by the
		// Storage Kit and a regular string-based path. Here we set the
		// BPath instance to the full path of the file we have opened,
		// and we will set the window's title to the file's name.
		BPath path(&realRef);
		fFilePath = path.Path();
		SetTitle(path.Leaf());
	}
}
```

```
void
MainWindow::SaveFile(const char *path)
{
	// This function takes a string path and saves the data in the
	// BTextView to the file, which is either created or overwrites an
	// existing one.
	BFile file;
	if (file.SetTo(path, B_READ_WRITE | B_CREATE_FILE | B_ERASE_FILE)
		!= B_OK)
		return;

	if (BTranslationUtils::PutStyledText(fTextView, &file) == B_OK)
	{
		fFilePath = path;

		BNodeInfo nodeInfo(&file);
		nodeInfo.SetType("text/plain");
	}
}


void
MainWindow::FrameResized(float w, float h)
{
	// Here is one of those little nuggets of wisdom to remember: when a
	// BTextView is resized, it does NOT update its text rectangle. When
	// the window is resized, so is our BTextView. The FrameResized()
	// method ensures that the text rectangle is updated.
	UpdateTextRect();
}


void
MainWindow::UpdateTextRect(void)
{
	BRect r(fTextView->Bounds());
	r.InsetBy(5, 5);
	fTextView->SetTextRect(r);
}
```

There are two additions to our repertoire in this chunk of code. The first is the calls to to `GetStyledText()` and `PutStyledText()`. The Translation Kit even takes care of saving formatted text information for us! The second addition to our toolkit of coding skills is the use of the BFilePanel class.

BFilePanel is a highly customizable file and directory chooser class. Although it is officially part of the Storage Kit, it could also be thought to partly belong in the Interface Kit. Once constructed, it is only necessary to call its `Show()` method to enable the user to choose a file. Earlier we saw these three lines in the MainWindow constructor:

```
BMessenger msgr(NULL, this);
fOpenPanel = new BFilePanel(B_OPEN_PANEL, &msgr, NULL, 0, false);
fSavePanel = new BFilePanel(B_SAVE_PANEL, &msgr, NULL, 0, false);
```

These create the open and save panels, respectively, but the arguments used in this call don't tell us much about how they work. Here is the declaration for the BFilePanel's constructor:

```
BFilePanel(file_panel_mode panelMode = B_OPEN_PANEL,
            BMessenger *target = NULL, entry_ref *panelDirectory = NULL,
            uint32 nodeTypes = 0, bool allowMultiple = true,
            BMessage *msg = NULL, BRefFilter *refFilter = NULL,
            bool modal = false, bool hide_when_done = true);
```

The constructor offers options aplenty and default values for literally each parameter. `panelMode` can either be set to `B_OPEN_PANEL` or `B_SAVE_PANEL`. Once set, it can't be changed. We need a panel for each mode in this instance because the two modes have small, but significant, differences in behavior. `target` is the recipient for the save or open message sent by the panel. The default target is `be_app_messenger`, the global BMessenger object which points to the BApplication instance used by each application. We have changed the panels to point to our window instead. `panelDirectory` can be any directory in the filesystem, but it defaults to the user's home folder. `allowMultiple` determines if multiple entries can be selected. `msg` is the message sent by the panel when a choice has been made. We'll get to the defaults for `msg` in a moment. `refFilter` is a BRefFilter object which can be used, for example, to display only certain file types. `modal` can make the panel's window require that a button be clicked; this isn't necessary except in rare cases. Don't make a file panel modal unless there is a *really* good reason. Finally, the panel stays open even after the user makes a choice if `hide_when_done` is set to false, also a little-used option.

The `nodeTypes` argument needs a little further explanation. Its value is set by one or more of three flags, `B_FILE_NODE`, `B_DIRECTORY_NODE`, and `B_SYMLINK_NODE`. The default, `B_FILE_NODE`, allows selection of files and any symlink which points to a file. Using `B_FILE_NODE` and `B_DIRECTORY_NODE` together would let the user also select directories and symlinks to directories. In either case, double-clicking on a directory in the panel will cause the panel to go into that directory, not select it. `B_SYMLINK_NODE` is almost always used alone and even then only very rarely: if used alone, it only allows symlinks to be selected.

The message sent when the user makes a choice depends on both the choice made and the panel's mode. Unless customized, an open panel will send a `B_REFS_RECEIVED` message and a save panel will send `B_SAVE_REQUESTED`. Customized messages will have the same `what` field as what was set in the constructor or by a call to `SetMessage()`. It will also have any additional data attached to the message when it was set – your original message is copied, the standard open or save fields are added, and the message is sent to the specified target.

Open notifications send a series of `entry_ref` objects held in the `refs` field of a message sent to your target. Note that these entries are the exact ones selected by the user. You must resolve any symlinks yourself, but doing so is trivial, if a little annoying: create a BEntry using the `entry_ref` and set the second argument to true. If you need a new `entry_ref`, call the BEntry's `GetRef()` method and you've got one. See `MainWindow::OpenFile()` in the code above if you need to see an example.

Save notifications contain two fields in addition to any you have added: an `entry_ref` named `directory` and a string named `name`. You can construct the full path to the file using these two fields in combination with a BPath instance. We did this in `MessageReceived()`. Alternatively, you can pass the ref to a BDirectory instance and call `CreateFile()`. Keep in mind that while the file may already exist, the user has already confirmed that overwriting the file is OK. On these occasions you just have to do the actual erasing. We handle this in `SaveFile()` by using the `B_CREATE_FILE` and `B_ERASE_FILE` flags so that the file is created if it doesn't exist and any existing ones are sufficiently clobbered.

Concluding Thoughts

If you have not yet undertaken any programming projects beyond just tinkering or a few simple "toy" applications, the code that we have seen in this lesson may seem a little larger than is comfortable. Large programs, such as a real word processor, are much, much bigger. However, such programs most often start small like what we're working on and gradually get larger. As long as we have a plan and a general design in mind, QuickEdit will not be too complicated for us to understand.

Going Further

- Think of possible features that you might like to implement on your own.
- Try to come up with possible ways that future features discussed in this lesson could be integrated into the application.