Learning to Program with Haiku

Lesson 17

Written by DarkWyrm

With a minor diversion into operator overloading and copy constructors, we've been slowly learning how to put together a graphical application for Haiku. So far we have examined the boilerplate code used to start any Haiku program, event-based programming, and sending messages. Today we'll be putting what we know about messages to use to create a menu for a program.

Using Menus

We're going to learn about several classes this lesson: BMenu, BMenuBar, BMenuItem, and BView, but we will be learning about them as we put together another relatively simple application that we'll call MenuColors, which shows a window with a colored box that we can change by choosing a color from a menu. Let's get our project set up.

1. Create a new, empty project in Paladin. You can call the project and target MenuColors or any other name you like.
2. Under the Project menu, click Add New File.
3. Type in the file name App.cpp, check the 'Create a header and a source file' box, and click OK or press Enter. This will create App.cpp and App.h.
4. Do the same for MainWindow.cpp.

Now that we have our files set up, let's bang out some code. First, here is the code for App.h and App.cpp. It would be a good idea to type everything out instead of copying and pasting the text so that you get familiarized with the boilerplate code.

```
//	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.test-MenuColors")
{
	MainWindow *mainwin = new MainWindow();
	mainwin->Show();
}


int
main(void)
{
```

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

None of the code in these files is anything new. Note that because we haven't entered in the class definition for MainWindow in its header file, if you try to build your project right now, you get some errors. Don't worry – we'll have everything working in just a moment. Here is the code for MainWindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <Window.h>

class MainWindow : public BWindow
{
public:
				MainWindow(void);
		void		MessageReceived(BMessage *msg);
};

#endif
```

This, too, is familiar territory. All of the new code will be in MainWindow.cpp, but let's start with the boilerplate code for MainWindow and add to it bit by bit. The best way to write code is to add a little at a time, compile, and then test what you've written to help you find bugs.

```
#include "MainWindow.h"

#include <MenuBar.h>
#include <Menu.h>
#include <MenuItem.h>
#include <View.h>


MainWindow::MainWindow(void)
	:	BWindow(BRect(100,100,500,400),"MenuColors",B_TITLED_WINDOW,
			B_ASYNCHRONOUS_CONTROLS | B_QUIT_ON_WINDOW_CLOSE)
{
}


void
MainWindow::MessageReceived(BMessage *msg)
{
	switch (msg->what)
	{
		default:
		{
			BWindow::MessageReceived(msg);
			break;
		}
	}
}
```

Now that we have all of our baseline code in place, build it just to make sure that you haven't mistyped anything. If you get errors, double check with the sources above and fix the typos. Once everything builds properly, run it and assuming everything works right, close it and move on to the interesting stuff.

Adding Views

Window controls are all objects which are descended from the BView class. If you take a quick glance at /boot/develop/headers/os/interface/View.h, you'll see that it's a large, complicated class with a lot of methods. Luckily, we don't need to know very many at this point. The BView below will be our colored box. Add this code to the MainWindow constructor:

```
BView *view = new BView(BRect(100,100,300,200),"colorview",B_FOLLOW_ALL,
                        B_WILL_DRAW);
AddChild(view);
view->SetViewColor(0,0,160);
```

This is pretty simple code. The first line creates a new BView which has its top left corner at (100,100) and its bottom right corner at (300,200). It has the name "colorview" and resizes itself whenever the window is resized. The `B_WILL_DRAW` flag at the end tells the window that it does its own drawing. Without this flag, this BView will just be a blank white box. The second line attaches the view to the window. The last one sets the color of the BView to a dark blue color. Build your project and see how it all looks.

Adding controls to windows in Haiku isn't much different. Most of them have a constructor which requires the same kinds of information and possibly a message, but not much else. Other controls are added to a window just the same way. Pay close attention to the methods that we use and you will notice many similarities among the classes.

Adding a Menu

With the exception of pop-up menus, menus in Haiku are normally kept in a menu container of some sort. The Haiku API provides two menu containers: BMenuField and BMenuBar. We'll ignore BMenuField for now and focus on BMenuBar. Change the constructor code to the following:

```
// This will define the height of the menu bar. Bounds() returns the size of the
// window. In this case, the rectangle will be (0,0)-(200,100).
BRect r(Bounds());
r.bottom = 20;

// The only part of r that matters is the height. When we add items to the menu
// bar, it will expand to fill the width of the window at the height we specify.
BMenuBar *menuBar = new BMenuBar(r,"menubar");
AddChild(menuBar);

BView *view = new BView(BRect(100,100,300,200),"colorview",B_FOLLOW_ALL,
                        B_WILL_DRAW);
AddChild(view);
view->SetViewColor(0,0,160);
```

Now that we have a container for our color menu, we can create the menu itself. This will require three components: the menu, the items in the menu, and message identifiers for each menu item. First, let's handle the message identifiers. Add this code just after the `#include` statements at the top of the file.

```
enum
{
	M_SET_COLOR_RED = 'sred',
	M_SET_COLOR_GREEN = 'sgrn',
	M_SET_COLOR_BLUE = 'sblu',
	M_SET_COLOR_BLACK = 'sblk'
};
```

The only part of this code that might look strange are the single-quoted values. This is another one of those fancy-schmancy coder tricks. Message constants are 32-bit integers. Each one of those letters translates into an 8-bit value, so `'sred'` actually translates into a 32-bit integer. Truth be told, the values themselves don't really matter much as long as they are unique, but by convention they are set to 4-letter constants like this.

The constants themselves can either be placed at the top of a class' source file or in its header. You will usually want to avoid putting the identifiers in the header so that adding a message identifier doesn't force a recompile of several files. The exception to this rule of thumb would be for messages that are used by more than one class.

Now that the message identifiers have been defined, let's go on to creating and populating the menu. The code looks like this:

```
MainWindow::MainWindow(void)
	:	BWindow(BRect(100,100,500,400),"MenuColors",B_TITLED_WINDOW,
			B_ASYNCHRONOUS_CONTROLS | B_QUIT_ON_WINDOW_CLOSE)
{
	BRect r(Bounds());
	r.bottom = 20;

	BMenuBar *menuBar = new BMenuBar(r,"menubar");
	AddChild(menuBar);

	// This is the code that creates and populates the menu.
	BMenu *menu = new BMenu("Colors");
	menu->AddItem(new BMenuItem("Red",new BMessage(M_SET_COLOR_RED),'R'));
	menu->AddItem(new BMenuItem("Green",new BMessage(M_SET_COLOR_GREEN),'G'));
	menu->AddItem(new BMenuItem("Blue",new BMessage(M_SET_COLOR_BLUE),'B'));
	menu->AddItem(new BMenuItem("Black",new BMessage(M_SET_COLOR_BLACK),'K'));

	// The menu bar adds menus the same way that a menu adds items. In fact, the
	// menu bar is more or less a menu whose items are arranged horizontally
	// instead of vertically.
	menuBar->AddItem(menu);

	BView *view = new BView(BRect(100,100,300,200),"colorview",B_FOLLOW_ALL,
				B_WILL_DRAW);
	AddChild(view);
	view->SetViewColor(0,0,160);
}
```

We're almost done! If you try running your project and clicking in the menu, you'll find that nothing happens. Everything works the way it should. When a menu item is clicked, it sends a message to the window, like clicking on the 'Red' menu item sends a `M_SET_COLOR_RED` message to the window. The window receives it, but doesn't do anything with it, so all that is needed now is to write the code for when the window receives each of the menu items' messages.

```
void
MainWindow::MessageReceived(BMessage *msg)
{
	// FindView() is a BWindow method which searches for a BView by name and
	// returns a pointer to it.
	BView *view = FindView("colorview");

	switch (msg->what)
	{
		case M_SET_COLOR_RED:
		{
			// When the window receives this message, we'll set the
			// background color to dark red
			view->SetViewColor(160,0,0);

			// Calling Invalidate() forces the view to redraw itself.
			view->Invalidate();
			break;
		}
		case M_SET_COLOR_GREEN:
		{
			view->SetViewColor(0,160,0);
			view->Invalidate();
			break;
		}
		case M_SET_COLOR_BLUE:
		{
			view->SetViewColor(0,0,160);
			view->Invalidate();
			break;
		}
		case M_SET_COLOR_BLACK:
		{
			view->SetViewColor(0,0,0);
			view->Invalidate();
			break;
		}
		default:
		{
			// As always, the default is to make the BWindow version of
			// this method handle messages we don't care about
			BWindow::MessageReceived(msg);
			break;
		}
	}
}
```

We're done now! Run your project and you will see that our box changes color when you click on one of the menu's items. It really doesn't take much effort to put together a menu.

Going Further

Here are some ideas you can try if you'd like to tinker with it some more.

1) Try changing the `B_FOLLOW_ALL` resizing mode flag in the BView constructor to something else. Change it to one of these and see what it does:

- `B_FOLLOW_LEFT | B_FOLLOW_TOP`
- `B_FOLLOW_LEFT_RIGHT | B_FOLLOW_TOP`
- `B_FOLLOW_RIGHT | B_FOLLOW_TOP`
- `B_FOLLOW_RIGHT | B_FOLLOW_BOTTOM`
- `B_FOLLOW_LEFT_RIGHT | B_FOLLOW_BOTTOM`
- `B_FOLLOW_RIGHT | B_FOLLOW_TOP_BOTTOM`

2) Try adding more colors to your menu

3) Add a Quit item to the menu which sends the window a `B_QUIT_REQUESTED` message.

Classes and Methods to Remember

BWindow

- `BWindow(BRect frame, const char *title, window_type type, uint32 flags, uint32 workspace = B_CURRENT_WORKSPACE)` – Create a new window. While there are other types, right now the window types to remember are `B_TITLED_WINDOW` and `B_DOCUMENT_WINDOW`.
- `void AddChild(BView *child)` – attaches a BView (or BView subclass) to the window.
- `BView * FindView(const char *name)` – Returns a pointer to a BView named `name` or `NULL` if not found.
- `BRect Bounds(void)` – Returns the size of the window's client area, i.e. the white area inside the window's frame.
- `void Show(void)` – Shows the window.

BView

- `BView(BRect frame, const char *name, int32 resizeMode, int32 flags)` – Create a new view. Check the BView section of the BeBook for all of the available resizing modes. Don't worry about the `flags` parameter just yet except to remember `B_WILL_DRAW`.
- `void SetViewColor(uint8 red, uint8 green, uint8 blue)` – Sets the background color of the view.
- `void Invalidate(void)` – Forces the BView to redraw itself.
- `void AddChild(BView *child)` – attaches a BView (or BView child class) to the view.