Learning to Program with Haiku

Lesson 22

Written by DarkWyrm

Implementing the fortune functions in the last lesson shouldn't have been too difficult. Rest assured that you will be able to finish this project even if it was harder than expected. We will be designing the graphical interface for our fortune program today.

Usability and Interface Design

One thing that developers need to be aware of is usability, which is quality of a program's suitability for a task while matching the technical level of its users. Haiku shows the roots that BeOS had in the Macintosh operating system. Although others have made notable strides in this area, Apple products have, for the most part, been shining examples of how to design products to be both stylish and easy to use. BeOS inherited these qualities without explicitly copying the Macintosh way of doing things. This also means that Haiku developers are expected to write programs that not only avoid being unnecessarily difficult but actually help the user do their work.

Usability is its own discipline and a good study of it would take far more than the scope of these lessons. Nonetheless, we should examine some basics of design to avoid major gaffes in writing our programs. A few minutes of thinking ahead can save a lot of headaches down the road for both you and your program's users.

Determine your program's main users.

The answer to this question could be developers or general Haiku users, for example. Knowing who will be using your program will guide you in figuring out your users needs and skills. Keep in mind that if your program has a general audience, like our fortune program, you will want to design your program for nontechnical users. This will enable as many people as possible to enjoy your work. When designing for nontechnical users, it is best to keep in mind someone you know who “doesn't know much about computers,” such as a grandparent, colleague, or spouse. The more effort you put into designing your app to match the expertise of your target audience, the less support you'll be asked to do later on and the greater popularity it will enjoy.

Decide the primary and secondary tasks that your program is meant to do.

Unless you're writing a game of some kind – which is just as admirable as programs to do "work" – your program will be intended to be a tool to perform one or more tasks. The influence of a feature on the overall design of the interface should be directly proportional to the frequency of its use. A program which rips MP3s from CDs should make getting album information as easy as possible. Editing the artist name, album release date, and other information which belongs in an MP3's tags should also be pretty simple. Features which may be used rarely, if ever, should be carefully considered before including them in your program. By adding features, you add complexity. This creates more code to support and maintain and also makes using your program more difficult for less skilled users in your target audience – people who are far too often ignored or marginalized by developers and power users. Your users are best thought of as reasonably intelligent, but very busy. By prioritizing the work your program will do, you will get the most return on your investment of time and also write a higher quality application.

Think of some of the ways that your program can be extra helpful.

In some ways, your program should be thought of as a butler: you tell him to go answer the door and he does a job which some consider drudgery without taking away your being in charge. Automating mindless tasks is something which computers do really well. Most people hate this kind of stuff. Even little thoughtful touches here and there add up to an overall great user experience.

Handle problems gracefully.

Few people like to be asked pointless questions. Fewer still like to have questions asked of them every few minutes. It's a lot like a four year old child who runs up to his mom – who is otherwise engaged in conversation – and proceeds to say "Mommy" every three or four seconds in an attempt to get her attention. Overusing BAlerts creates this kind of annoyance for the user. It's OK to ask the user questions, but do so in a way that is not condescending and try to keep the frequency to a minimum whenever possible. Knowing full well that Murphy's Law certainly applies to programming, try to think of things that could go wrong and handle them gracefully. Crashes are not an option, nor is losing a user's data unless there is no other way out.

Designing Our GUI

Our fortune program isn't terribly important in the grand scheme of things, but it will be something that the user sees fairly often if it is set up to run whenever the computer is booted. This is a program for anyone, so it should be as nontechnical as possible. The main thing it will do is show a fortune when it is started, but we also should make it simple to get another fortune without having to restart the program.

We will only do one thing a couple of different ways, so we should be able to get away with using buttons instead of having to create a menu. One button can be for getting another fortune. Although there is always a close button on the window's tab, we should also use a regular button so the user can close the window by pressing the Enter key. Because we want people to have a way to know its version and who wrote this fortune program, we'll make a button which shows an About window. This should be easy to write.

Writing the Code

The first thing we should do is find out where the fortunes are being kept. Under BeOS and Zeta, they were kept in /boot/beos/etc/fortunes. Haiku keeps them in a different place: /boot/system/data/fortunes. Rather than try to guess what platform we might be using, we'll use `find_directory()`.

```
status_t find_directory(directory_which which, BPath *path,
                        bool and_create_it = false, BVolume *vol = NULL);
```

This call takes a constant – a list of which can be found in FindDirectory.h – and places the location that corresponds to it into `path`. Use this call whenever you need to get a common location in the filesystem, such as the home folder, the user's settings folder, the common add-ons folder, and so on. If the actual location of such a folder should change, your program won't suddenly stop working correctly.

We are going to make a global variable to hold the path for the fortunes. Using lots of global variables is a bad idea, but one in this program won't hurt anything. Add this line to FortuneFunctions.h:

```
extern BString gFortunePath;
```

Place this somewhere between the includes and your class code in FortuneFunctions.cpp.

```
BString gFortunePath = "/boot/beos/etc/fortunes";
```

The `extern` keyword declares `gFortunePath` without defining it. If FortuneFunctions.h is used by more than one file, this will prevent linker problems. We actually define and initialize it in the main sources.

Now that we've made the necessary tweaks to our fortune access class, let's turn our attention to App.cpp. We will need to do a little additional setup before we show our fortune window.

```
#include "App.h"

#include <FindDirectory.h>
#include <OS.h>
#include <Path.h>
#include <stdlib.h>

#include "FortuneFunctions.h"
#include "MainWindow.h"

App::App(void)
  :	BApplication("application/x-vnd.test-HaikuFortune")
{
	BPath path;
	
	// We have to use an #ifdef here because the fortune files under R5
	// and Zeta are in the system/etc/ directory, but in Haiku they're
	// kept in the system/data directory. We detect the platform by using
	// a compiler definition. __HAIKU__ is defined under Haiku, but not BeOS
	// R5 or Zeta. Zeta has its own __ZETA__ definition. All three have the
	// __BEOS__ definition which, in this case, isn't at all useful.
	#ifdef __HAIKU__
	find_directory(B_SYSTEM_DATA_DIRECTORY,&path);
	#else
	find_directory(B_BEOS_ETC_DIRECTORY,&path);
	#endif
	
	path.Append("fortunes");
	gFortunePath = path.Path();
	
	// If we want the rand() function to actually be pretty close to random
	// we will need to seed the random number generator with the time. If we
	// don't, we will get the same "random" numbers each time the program is
	// run. rand() isn't really random, but this makes it close enough for us.
	srand(system_time());
	
	MainWindow *win = new MainWindow();
	win->Show();
}


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

Now let's move on to the MainWindow class.

```
#include "MainWindow.h"

#include <Alert.h>
#include <Application.h>
#include <Button.h>
#include <Screen.h>
#include <ScrollView.h>
#include <View.h>


// We only need a couple of message constants for this app
enum
{
      M_GET_ANOTHER_FORTUNE = 'gafn',
      M_ABOUT_REQUESTED = 'abrq'
};


MainWindow::MainWindow(void)
  :   BWindow(BRect(0,0,300,300), "HaikuFortune", B_DOCUMENT_WINDOW,
                  B_ASYNCHRONOUS_CONTROLS | B_QUIT_ON_WINDOW_CLOSE),
      fFortune(gFortunePath.String())
{
      // Create all of the controls for our window.
      BView *back = new BView(Bounds(), "background", B_FOLLOW_ALL, B_WILL_DRAW);
      back->SetViewColor(ui_color(B_PANEL_BACKGROUND_COLOR));
      AddChild(back);

      BButton *close = new BButton(BRect(0,0,1,1), "closebutton", "Close",
                              new BMessage(B_QUIT_REQUESTED),
                              B_FOLLOW_RIGHT | B_FOLLOW_BOTTOM);
      close->ResizeToPreferred();
      close->MoveTo(Bounds().right - 15 - close->Frame().Width(),
                        Bounds().bottom - 15 - close->Frame().Height());
      back->AddChild(close);
      close->MakeDefault(true);

      BButton *next = new BButton(BRect(0,0,1,1), "nextbutton", "Get another",
                              new BMessage(M_GET_ANOTHER_FORTUNE),
                              B_FOLLOW_RIGHT | B_FOLLOW_BOTTOM);
      next->ResizeToPreferred();
      next->MoveTo(close->Frame().left - 15 - next->Frame().Width(),
                        Bounds().bottom - 15 - next->Frame().Height());
      back->AddChild(next);

      BButton *about = new BButton(BRect(0,0,1,1), "aboutbutton",
                              "About" B_UTF8_ELLIPSIS,
                              new BMessage(M_ABOUT_REQUESTED),
                              B_FOLLOW_LEFT | B_FOLLOW_BOTTOM);
      about->ResizeToPreferred();
      about->MoveTo(next->Frame().left - 15 - about->Frame().Width(),
                        Bounds().bottom - 15 - about->Frame().Height());
      back->AddChild(about);
```

```
	BRect r(15,15,Bounds().right - B_V_SCROLL_BAR_WIDTH – 15,
		next->Frame().top - 15);


	fTextView = new BTextView(r, "textview",
				r.OffsetToCopy(0,0).InsetByCopy(10,10),
				B_FOLLOW_ALL, B_WILL_DRAW | B_PULSE_NEEDED |
							B_FRAME_EVENTS);
	fTextView->MakeEditable(false);

	// BScrollViews are a little weird. You can't create one without having
	// created its target. It also automatically calls AddChild() to attach its
	// target to itself, so all that is needed is to instantiate it and attach
	// it to the window. It's not even necessary (or possible) to specify the
	// size of the BScrollView because it calculates its size based on that of
	// its target.
	BScrollView *sv = new BScrollView("scrollview", fTextView, B_FOLLOW_ALL, 0,
							false, true);
	back->AddChild(sv);

	BString fortune;
	status_t status = fFortune.GetFortune(fortune);
	if (status == B_OK)
	{
		BString title;
		title.Prepend("Fortune: ");
		SetTitle(title.String());

		fTextView->SetText(fortune.String());
	}
	else
	{
		fTextView->SetText("HaikuFortune had a problem getting a fortune.\n\n"
			"Please make sure that you have installed fortune files to "
			"the folder ");
		fTextView->Insert(gFortunePath.String());
	}

	// This code is for working around a problem in Zeta. BButton::MakeDefault
	// doesn't do anything except change the focus. The idea is to be able to
	// press Enter to close HaikuFortune and the space bar to get another
	// fortune. Zeta doesn't let us do this, so we'll leave focus on the Close
	// button.
	if (B_BEOS_VERSION <= B_BEOS_VERSION_5)
		next->MakeFocus(true);

	// Calculate a good width for the window and center it
	SetSizeLimits(45 + close->Bounds().Width() + next->Bounds().Width(), 30000,
			200, 30000) ;
	r = BScreen().Frame();
	MoveTo((r.Width()-Bounds().Width()) / 2.0, r.Height() / 4.0);
}


void
MainWindow::MessageReceived(BMessage *msg)
{
	switch (msg->what)
	{
		case M_GET_ANOTHER_FORTUNE:
		{
			BString fortune;
			status_t status = fFortune.GetFortune(fortune);

			if (status == B_OK)
			{
				BString title;
				fFortune.LastFilename(title);
				title.Prepend("Fortune: ");
				SetTitle(title.String());

				fTextView->SetText(fortune.String());
			}
			else
			{
				fTextView->SetText("HaikuFortune had a problem getting a "
							"fortune.\n\nPlease make sure that you "
							"have installed fortune files to the "
							"folder ");
				fTextView->Insert(gFortunePath.String());
			}
			break;
		}
		case M_ABOUT_REQUESTED:
		{
			// Using a BAlert for the About window is a common occurrence.
			// They take care of all of the layout, so all that needs done
			// is write the text that goes in the alert.
			BAlert *alert = new BAlert("HaikuFortune",
						"A graphical fortune program for "
						"Haiku.\n\n", "OK");
			alert->Go();
			break;
		}
		default:
		{
			BWindow::MessageReceived(msg);
			break;
		}
	}
}


void
MainWindow::FrameResized(float w, float h)
{
	// The BWindow::FrameResized() method is called whenever the window is
	// resized. In this case, we change the size of the text rectangle in the
	// text view used for the fortune. We have to do this because it won't do it
	// itself. Lazy. :(
	BRect textrect = fTextView->TextRect();

	textrect.right = textrect.left + (w - B_V_SCROLL_BAR_WIDTH - 40);
	fTextView->SetTextRect(textrect);
}
```

Conclusion

While the code is done and you should be able to run it and get a nice fortune, we're not quite done yet. We'll finish it next time, but for now, enjoy the satisfaction of having written a good program!