Programming with Haiku

Lesson 17

Written by DarkWyrm

The Interface Kit is all about creating and using controls for the graphical interface. In some of the earlier lessons, we touched on using some of the existing controls in the kit, but to fully understand how to use them, we will create our own custom control. In doing so, we will learn about the different parts of the Interface Kit and how to use them effectively.

The Haiku Way of Controls

If you have been involved in programming for a while, you will probably know something about good code design. If not, you can learn a little from the Haiku API. The developers at Be Inc. were careful to put together the API in a way that it's pretty easy to use. It also uses a few paradigms which would be wise to follow in the appropriate places.

One concept of control design used in the Interface Kit is using lightweight items for lists. The BListView and BOutlineListView classes exemplify this. Each can have potentially hundreds of items, so each must take up as little space as possible. BStringItem objects, instead of each one carrying the sizable overhead of being a BView individually, depend on their owning list for drawing themselves. They only concern themselves with a little data, such as in the case of BStringItem.

Another rule of thumb used in the Interface Kit is utilizing existing classes to implement your control whenever possible. BTextControl is just a single-line special case of BTextView, for example. By using standard controls in your own instead of implementing new ones, you can help keep the interface consistent for the user.

The API used to interact with your control should also follow conventions used by other classes in the Interface Kit. Provide hook functions for events. Better yet, if your control has a value and a label, make your control a subclass of BControl. Note that not all controls in the Interface Kit do so, such as BListView. If you don't subclass BControl, use the BInvoker class as a mix-in to easily change message targets and otherwise provide a familiar interface to messaging for your control. Make most data returning functions `const` functions – ones which tell the compiler that they don't change the state of the object. Lastly, don't write a new control unless one available in the kit isn't appropriate to the task at hand. Often one or two controls used together can perform the same task a new control coded specifically for the task. Work smarter, not harder.

Our New Control: ColorWell

One kind of control which the Haiku API lacks is a color display control. In addition to images and text, Haiku also allows colors to be dragged and dropped from one location to another. We will be creating a well which holds and displays a color value and, later on, supports dragging colors to other locations and receiving such dragged colors. Note that we will not be creating a color picker to modify the color – that task is easily handled by the BColorControl class.

There are a few things that we should keep in mind for our control. First, colors can be represented by 32-bit integers and as `rgb_color` objects, so we will want to support both ways of assigning and returning color values. Second, we should have both a round style and a square style to fit with the style of the program in which it might be used. Third, it should be possible to disable our control, so we should also think of a look for the control when it is disabled which is easy to discern as such.

Here's what our header should look like:

ColorWell1.h

```
#ifndef COLOR_WELL_H
#define COLOR_WELL_H

#include <Control.h>

enum
{
      COLORWELL_SQUARE_WELL,
      COLORWELL_ROUND_WELL,
};

class ColorWell : public BControl
{
public:
                              ColorWell(BRect frame, const char *name,
                                    BMessage *msg,
                                    int32 resize = B_FOLLOW_LEFT |
                                                      B_FOLLOW_TOP,
                                    int32 flags = B_WILL_DRAW,
                                    int32 style = COLORWELL_SQUARE_WELL);
                              ~ColorWell(void);

      virtual     void        SetValue(int32 value);
      virtual     void        SetValue(const rgb_color &color);
      virtual     void        SetValue(const uint8 &r, const uint8 &g,
                                    const uint8 &b);
                  rgb_color   ValueAsColor(void) const;

      virtual                 SetStyle(const int32 &style);
                  int32       Style(void) const;

      virtual     void        Draw(BRect update);

private:
                  void        DrawRound(void);
                  void        DrawSquare(void);

                  rgb_color   fDisabledColor,
                              fColor;

                  int32       fStyle;
};

#endif
```

Having given this a look, think for a moment what might go into each of these methods.

Before we go diving into code, let's take a quick side trip to learn a bit about how BViews handle drawing. There are four steps to how a BView draws itself: invalidation, drawing, child drawing, and post-child drawing. Invalidation just means that the BView tells the app_server that a certain section of its area needs to be redrawn. Invalid regions are caused by all sorts of things, such as if a window hid part of it and now it doesn't or perhaps an internal value has changed and the BView needs to update a label. Later on, the BView will actually draw itself, handled by the `Draw()` hook function. Once a view has drawn itself, it typically will make sure that any child views also are redrawn as needed. It is also possible for a BView to do some drawing after all of its child BViews have finished updating themselves.

Any drawing of this kind is done with the `DrawAfterChildren()` hook function. It is not often used, however.

Coding the ColorWell

ColorWell1.cpp

```
#include "ColorWell1.h"

ColorWell::ColorWell(BRect frame, const char *name, BMessage *message,
      int32 resize, int32 flags, int32 style)
      : BControl(frame,name,NULL,message, resize, flags)
{
      SetViewColor(ui_color(B_PANEL_BACKGROUND_COLOR));
      SetLowColor(0,0,0);

      fColor.red = 0;
      fColor.green = 0;
      fColor.blue = 0;
      fColor.alpha = 255;

      fDisabledColor.red = 190;
      fDisabledColor.green = 190;
      fDisabledColor.blue = 190;
      fDisabledColor.alpha = 255;
}


ColorWell::~ColorWell(void)
{
}


void
ColorWell::SetValue(int32 value)
{
      // It might seem strange to support setting the color from an
      // integer, but this behavior is expected for BControl-derived
      // controls, so we will do our best to support it.
      BControl::SetValue(value);

      // int32's can be used to pass colors around, being that they contain
      // 4 8-bit integers. Converting them to RGB format requires a little
      // bit shift division.
      fColor.red = (value & 0xFF000000) >> 24;
      fColor.green = (value & 0x00FF0000) >> 16;
      fColor.blue = (value & 0x0000FF00) >> 8;
      fColor.alpha = 255;

      SetHighColor(fColor);
      Draw(Bounds());
}


void
ColorWell::SetValue(const rgb_color &col)
{
      fColor = col;
```

```
        fColor.alpha = 255;

        // Calling the BControl version of this method is necessary because
        // BControl::Value() needs to return the proper value regardless of
        // which way the value was set.
        BControl::SetValue((fColor.red << 24) + (fColor.green << 16) +
                           (fColor.blue << 8) + 255);
        SetHighColor(col);
        Draw(Bounds());
}


void
ColorWell::SetValue(const uint8 &r,const uint8 &g, const uint8 &b)
{
        fColor.red = r;
        fColor.green = g;
        fColor.blue = b;
        fColor.alpha = 255;

        BControl::SetValue((fColor.red << 24) + (fColor.green << 16) +
                                (fColor.blue << 8) + 255);
        SetHighColor(r,g,b);
        Draw(Bounds());
}


void
ColorWell::SetStyle(const int32 &style)
{
        if (style != fStyle)
        {
                fStyle = style;
                Invalidate();
        }
}


int32
ColorWell::Style(void) const
{
        return fStyle;
}


void
ColorWell::Draw(BRect update)
{
        if (fStyle == COLORWELL_SQUARE_WELL)
                DrawSquare();
        else
                DrawRound();
}

rgb_color
ColorWell::ValueAsColor(void) const
{
        return fColor;
}
```

```
void
ColorWell::DrawRound(void)
{
	// Although real controls require more work to look nice, just a
	// simple black border will do for our purposes.
	if (IsEnabled())
		SetHighColor(fColor);
	else
		SetHighColor(fDisabledColor);

	FillEllipse(Bounds());

	SetHighColor(0,0,0);
	StrokeEllipse(Bounds());
}


void
ColorWell::DrawSquare(void)
{
	// The square version of our ColorWell doesn't get any more
	// complicated than the round one.
	if (IsEnabled())
		SetHighColor(fColor);
	else
		SetHighColor(fDisabledColor);

	FillRect(Bounds());

	SetHighColor(0,0,0);
	StrokeRect(Bounds());
}
```

None of these methods require very much thought. This shouldn't be surprising, though. The framework that Haiku provides for creating controls has just enough features to do much of the heavy lifting without being terribly complicated. Now let's round out the rest of the project with the necessary GUI to go with our new control.

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.jy-ColorWellDemo1")
{
	MainWindow *mainwin = new MainWindow();
	mainwin->Show();
}


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

MainWindow.h

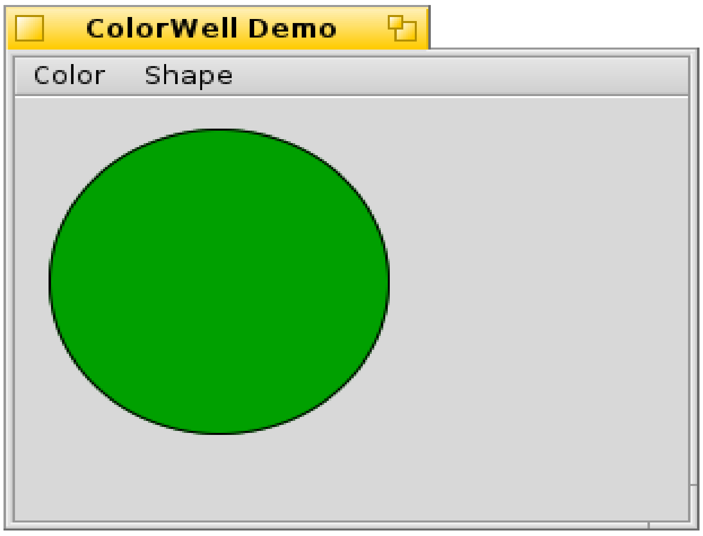
```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <Window.h>
#include <MenuBar.h>

class ColorWell;

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

private:
			BMenuBar	*fMenuBar;
			ColorWell	*fColorWell;
};

#endif
```

MainWindow.cpp

```
#include "MainWindow.h"

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

#include "ColorWell1.h"

enum
{
	M_SET_COLOR = 'stcl',
	M_COLOR_UPDATED = 'mcup',
	M_SET_SHAPE_CIRCLE = 'sscr',
	M_SET_SHAPE_SQUARE = 'sssq'
};
```

```
MainWindow::MainWindow(void)
	:	BWindow(BRect(100,100,500,400),"ColorWell Demo",
			B_TITLED_WINDOW, B_ASYNCHRONOUS_CONTROLS)
{
	BRect r(Bounds());
	r.bottom = 20;
	fMenuBar = new BMenuBar(r,"menubar");
	AddChild(fMenuBar);

	// Create a background view to make the window look semi-normal. If
	// you're going to have a group of controls in a BWindow, it's best
	// to create a background view.
	r = Bounds();
	r.top = 20;
	BView *background = new BView(r, "background", B_FOLLOW_ALL,
					B_WILL_DRAW);

	// SetViewColor() sets the background color of a view. ui_color() is
	// a global C++ function which returns an rgb_color given one of the
	// system color constants. B_PANEL_BACKGROUND_COLOR happens to be the
	// default background color for BViews. By default, this is
	// (216, 216, 216).
	background->SetViewColor(ui_color(B_PANEL_BACKGROUND_COLOR));
	AddChild(background);

	// Create our color well control. It's bigger than really necessary,
	// but it'll be fine for a demo.
	fColorWell = new ColorWell(BRect(15, 15, 165, 165), "color well",
					new BMessage(M_COLOR_UPDATED));

	// Note that we call the background view's AddChild method here. If
	// two views which have the same parent overlap each other, the
	// results, according to the Be Book, are unpredictable. In most
	// cases, though, this sometimes results in drawing errors and many
	// times the view which was added later doesn't receive mouse events.
	background->AddChild(fColorWell);

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

	// BMessages can have data attached to them. As such, they are an
	// incredibly flexible data container. Here we are just going to
	// attach color values to the message for each color menu item. This
	// is not the standard way of attaching a color, but it will work
	// well enough for this example.
	BMessage *msg = new BMessage(M_SET_COLOR);
	msg->AddInt8("red", 160);
	msg->AddInt8("green", 0);
	msg->AddInt8("blue", 0);
	menu->AddItem(new BMenuItem("Red", msg, 'R', B_COMMAND_KEY));
	msg = new BMessage(M_SET_COLOR);
	msg->AddInt8("red", 0);
	msg->AddInt8("green", 160);
	msg->AddInt8("blue", 0);
	menu->AddItem(new BMenuItem("Green", msg, 'G', B_COMMAND_KEY));

	msg = new BMessage(M_SET_COLOR);
	msg->AddInt8("red", 0);
	msg->AddInt8("green", 0);
	msg->AddInt8("blue", 160);
	menu->AddItem(new BMenuItem("Blue", msg, 'B', B_COMMAND_KEY));

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

	menu->AddItem(new BMenuItem("Square",
			new BMessage(M_SET_SHAPE_SQUARE), 'S', B_COMMAND_KEY));
	menu->AddItem(new BMenuItem("Circle",
			new BMessage(M_SET_SHAPE_CIRCLE), 'C', B_COMMAND_KEY));
}


void
MainWindow::MessageReceived(BMessage *msg)
{
	switch (msg->what)
	{
		case M_SET_COLOR:
		{
			// Yank the stored color values and plonk them into the
			// ColorWell.
			int8 red, green, blue;
			msg->FindInt8("red", &red);
			msg->FindInt8("green", &green);
			msg->FindInt8("blue", &blue);

			fColorWell->SetValue(red, green, blue);
			break;
		}
		case M_SET_SHAPE_CIRCLE:
		{
			fColorWell->SetStyle(COLORWELL_ROUND_WELL);
			break;
		}
		case M_SET_SHAPE_SQUARE:
		{
			fColorWell->SetStyle(COLORWELL_SQUARE_WELL);
			break;
		}
		default:
		{
			BWindow::MessageReceived(msg);
			break;
		}
	}
}


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

With all of this code in place, build your program and you should have a working demo app which displays a big swatch of color which can be changed to suit our whims. It doesn't do much except display the current color – right now it makes a good companion to a BColorControl because one edits the color and the other displays it. Not bad.



Final Thoughts

At this point we have a very basic control. If this were going to just be a quick and dirty class to get a job done in a much larger project, we could probably get away with leaving it as it is, but since the goal is a full-fledged Haiku control, there is more that will need to be done. It isn't very often you will need to implement all of the different possible bells and whistles that we will be putting into our ColorWell class, but knowing how to use each set of features will open up new ideas that you may not have had otherwise.

Going Further

This isn't just some exercise. It's your code, too, so try thinking up some uses you might have for this control in your other projects and perhaps some extra features you might want to implement that could come in handy.