Programming with Haiku

Lesson 21

Written by DarkWyrm

Haiku Replicants

"Fiery the angels rose, and as they rose deep thunder roll'd. Around their shores: indignant burning with the fires of Orc."

Replicants in the world of Haiku are nothing short of amazing, but they have nothing to do with *Blade Runner*. Instead, they are very similar to a component technology. For those unfamiliar, components are an object-oriented programming concept where a program consists of objects with a generic interface to their specific features. Built into the Haiku API is a means of interacting with a BView-based object that does not require knowing anything about it. Back in 1995, BeOS could embed a web browser into the desktop without creating a major security risk like Windows 95. Sadly, not much was done with replicants after Be introduced them, but more has been done with them by the Haiku developers in recent years. We'll take some time in this lesson to see how they are put together.

Archiving and Instantiation

One of the topmost classes in the hierarchy of the Haiku API is the BArchivable class. Its purpose is to provide an interface which enables child classes to save their state into a BMessage which can be sent to a target or flattened and saved to disk. In case you haven't noticed the trend over the course of these lessons, BMessage is both a floor wax and a dessert topping.

Only three functions must be implemented to make an object archivable: `Archive()`, `Instantiate()` and a version of the object's constructor which takes a BMessage as its only parameter. `Archive()` saves the object's state into a BMessage and `Instantiate()` loads its state from a BMessage. Archiving is easiest to understand in the context of an existing control, so let's examine some new code for the ColorWell class to explain.

```
status_t
ColorWell::Archive(BMessage *data, bool deep) const
{
	status_t status = BControl::Archive(data, deep);
	data->AddString("class","ColorWell");

	return status;
}
```

The `Archive()` method does four things: call the inherited method to make sure parent classes can save any appropriate properties, store the class' name into the `class` property of the BMessage, place any state information of the object into the BMessage, and save any extra information if `deep` is true.

In our example here, we don't have to store any extra information. The BControl version of `Archive()` saves the control's value. Our ColorWell class stores the value as both an `rgb_color` structure and an integer. When we unarchive a ColorWell instance, we'll just convert the integer value – which is saved by `BControl::Archive()` – to the `rgb_color` and we'll have the exact same state as what was saved into the message.

More complicated objects may place other properties into the BMessage archive. Unfortunately, there is no defined naming protocol for these properties, so we have to be careful about name choices to avoid collisions. The easiest way is to save properties using the class name like this:

```
msg->AddString("MyClassName::MyStringProperty","SomeString");
msg->AddBool("MyClassName::SomeOtherProperty",someBooleanFlag);
```

The deep flag is often ignored by simpler objects. Its purpose is to signal to the object to go above and beyond the call of duty to save its state. For example, if a BView is given a deep archive request, it also saves the states of its children.

Now that we have examined how to save the state of an object, loading is next. We will define the other two methods here:

```
ColorWell::ColorWell(BMessage *data)
 : BControl(data)
{
	// The inherited BControl constructor will pull the integer value
	// of the color out of the message for us, so all we have to do
	// is update fColor. Instead of copying and pasting code, we'll
	// call SetValue() to do all of the heavy lifting for us. This
	// kind of trick eases debugging and code maintenance.
	SetValue(Value());
}


BArchivable *
ColorWell::Instantiate(BMessage *data)
{
	if (validate_instantiation(data, "ColorWell"))
		return new ColorWell(data);

	return NULL;
}
```

`Instantiate()` is boilerplate code. `validate_instantiation()` is just a check to make sure that the BMessage holds the information for a ColorWell object. Assuming that the check passes, it returns a ColorWell instance using the unarchiving constructor mentioned above. This constructor just loads data from the BMessage in the same way that data was saved in `Archive()`.

To instantiate an archived object, call `instantiate_object()` like this:

```
BArchivable *archivable = instantiate_object(msg);

MyDesiredClass *object = NULL;
if (archivable)
	object = dynamic_cast<MyDesiredClass*>(archivable);
```

The Be Book advocates using the preprocessor macro `cast_as()`, -but this is deprecated. Use a `dynamic_cast`.

Although it's possible to unarchive just about any archive message, there is no defined protocol for discovering what is kept in an archive – Be expected developers to follow predefined protocols, such as what is used for replicants.

Creating a Replicant

A replicant is simply an archivable BView control. Assuming that each child control can be archived and instantiated properly, all that is needed to create a replicant is to add an instance of the BDragger class.

```
BDragger(BRect frame, BView *target, int32 resizeMode = B_FOLLOW_NONE,
            uint32 flags = B_WILL_DRAW);
```

BDragger objects are the key to creating replicants. They are a child class of BView and have a target BView, much like BScrollView. They come with a few conditions that are expected to be met:

1. A BDragger expects its target to be its parent, child, or sibling in the view hierarchy.
2. If the BDragger instance is the child of its target, it must be the target's only child view and its frame must be only as big as the drag handle – 7 pixels.
3. If the BDragger instance is the target's parent, then its frame must be at least as big as its target.

Here is an example of how simple it is to make a replicant. This is the source code to the main view from the Haiku demo application OverlayImage.

```
/*
 * Copyright 1999-2010, Be Incorporated. All Rights Reserved.
 * This file may be used under the terms of the Be Sample Code License.
 *
 * Authors:
 *               Seth Flexman
 *               Hartmuth Reh
 *               Humdinger        <humdingerb@gmail.com>
 */

#include "OverlayView.h"

#include <Catalog.h>
#include <InterfaceDefs.h>
#include <Locale.h>
#include <String.h>
#include <TextView.h>

#undef B_TRANSLATE_CONTEXT
#define B_TRANSLATE_CONTEXT "Main window"

const float kDraggerSize = 7;


OverlayView::OverlayView(BRect frame)
      :
      BView(frame, "OverlayImage", B_FOLLOW_NONE, B_WILL_DRAW)
{
      fBitmap = NULL;
      fReplicated = false;

      frame.left = frame.right - kDraggerSize;
      frame.top = frame.bottom - kDraggerSize;
	BDragger *dragger = new BDragger(frame, this, B_FOLLOW_RIGHT |
	                                               B_FOLLOW_BOTTOM);
	AddChild(dragger);

	SetViewColor(B_TRANSPARENT_COLOR);

	fText = new BTextView(Bounds(), "bgView", Bounds(), B_FOLLOW_ALL,
	                         B_WILL_DRAW);
	fText->SetViewColor(ui_color(B_PANEL_BACKGROUND_COLOR));
	AddChild(fText);
	BString text;
	text << B_TRANSLATE(
		"Enable \"Show replicants\" in Deskbar.\n"
		"Drag & drop an image.\n"
		"Drag the replicant to the Desktop.");
	fText->SetText(text);
	fText->SetAlignment(B_ALIGN_CENTER);
	fText->MakeSelectable(false);
	fText->MoveBy(0, (Bounds().bottom - fText->TextRect().bottom) / 2);
}


OverlayView::OverlayView(BMessage *archive)
	:
	BView(archive)
{
	fReplicated = true;
	fBitmap = new BBitmap(archive);
}


OverlayView::~OverlayView()
{
	delete fBitmap;
}


void
OverlayView::Draw(BRect)
{
	SetDrawingMode(B_OP_ALPHA);
	SetViewColor(B_TRANSPARENT_COLOR);

	if (fBitmap)
		DrawBitmap(fBitmap, B_ORIGIN);
}


void
OverlayView::MessageReceived(BMessage *msg)
{
	switch (msg->what) {
		case B_SIMPLE_DATA:
		{
			if (fReplicated)
				break;

			entry_ref ref;
			msg->FindRef("refs", &ref);
```

```
                  BEntry entry(&ref);
                  BPath path(&entry);

                  delete fBitmap;
                  fBitmap = BTranslationUtils::GetBitmap(path.Path());

                  if (fBitmap != NULL) {
                        if (fText != NULL) {
                              RemoveChild(fText);
                              fText = NULL;
                        }

                        BRect rect = fBitmap->Bounds();
                        if (!fReplicated) {
                              Window()->ResizeTo(rect.right, rect.bottom);
                              Window()->Activate(true);
                        }
                        ResizeTo(rect.right, rect.bottom);
                        Invalidate();
                  }
                  break;
            }
          case B_ABOUT_REQUESTED:
                  OverlayAboutRequested();
            break;

            default:
                  BView::MessageReceived(msg);
                  break;
      }
}


BArchivable *OverlayView::Instantiate(BMessage *data)
{
      return new OverlayView(data);
}


status_t
OverlayView::Archive(BMessage *archive, bool deep) const
{
      BView::Archive(archive, deep);

      archive->AddString("add_on", "application/x-vnd.Haiku-OverlayImage");
      archive->AddString("class", "OverlayImage");

      if (fBitmap) {
            fBitmap->Lock();
            fBitmap->Archive(archive);
            fBitmap->Unlock();
      }

      return B_OK;
}
```

```
void
OverlayView::OverlayAboutRequested()
{
	BAlert *alert = new BAlert("about",
		"OverlayImage\n"
		"Copyright 1999-2010\n\n\t"
		"originally by Seth Flaxman\n\t"
		"modified by Hartmuth Reh\n\t"
		"further modified by Humdinger\n",
		"OK");

	BTextView *view = alert->TextView();
	BFont font;
	view->SetStylable(true);
	view->GetFont(&font);
	font.SetSize(font.Size() + 7.0f);
	font.SetFace(B_BOLD_FACE);
	view->SetFontAndColor(0, 12, &font);

	alert->Go();
}
```

Once you've constructed the target BView and the associated BDragger and placed them appropriately into the view hierarchy, there isn't anything else you need to do. The user can drag the BDragger handle and it will replicate the original view. Of course, this doesn't do much good unless you can drop it somewhere. This is where another class, BShelf, comes in.

BShelf is a BHandler which attaches to a BView and accepts replicants. There are three versions of the constructor:

```
BShelf(BView *view, bool allowDragging = true, const char *name = NULL);
BShelf(entry_ref *ref, BView *view, bool allowDragging = true,
	const char *name = NULL);
BShelf(BDataIO *stream, BView *view, bool allowDragging = true,
	const char *name = NULL);
```

The last two versions of the constructor initialize the shelf to a file where the shelf will save any replicants. If `allowDragging` is true, the user is allowed to move replicants around once they have been placed on the shelf. Otherwise, they stay where they were initially placed. The name of the shelf is important – if it has a name, the system will check to make sure that any replicants dropped onto it have a `shelf_type` field which matches its name. Any replicants which do not have a matching name will be rejected.

Aside from the constructor, working with the BShelf is very straightforward. It is possible to programmatically add, remove, and count replicants attached to a shelf. The `Save()` method makes it possible for a shelf to have a state which is persistent from one program execution to another.

Concluding Thoughts

Long before the Windows Sidebar and Google Gadgets existed, BeOS had replicants. Like drag and drop support and scripting support, they are a wonderful technology which has been underutilized. Consider how your programs may use it. More than likely, you'll end up making work a little easier and a little more fun for your target audience.

This lesson also concludes our building of a full-featured, fully-implemented control. Most developers do not write their code this completely. Your code does not necessarily have to be, either. Pick and choose from the features which your program requires and as long as it works well, no one will know the difference or even care.