Learning to Program with Haiku

Lesson 19

Written by DarkWyrm

All material © 2010 DarkWyrm

The Interface Kit has been our primary source of functionality in the last couple of lessons, but today we'll be expanding our reach. This lesson will deal with the Translation Kit and part of the Storage Kit.

The Translation Kit

Originally written by Jon Watte in ancient Be history, the Translation Kit has nothing to do with language translation. Instead, it is used for converting images and text from one format to another. This usually amounts to loading bitmap images from disk. A **bitmap** is a kind of image which stores its picture information as a collection of dots. **Vector images** store their information as a series of drawing instructions. This is the difference between a photo, which is a bitmap image, and a clipart picture.

The Translation Kit uses a unique approach to loading pictures: its capabilities are defined and expanded by add-ons. Add-ons are a powerful programming approach used in many places in Haiku. What makes them so powerful in this case is that installing a Translator for a particular image format enables every Haiku program which utilizes the Translation Kit to suddenly support that format. It's a much better approach than can be found on any other operating system available. It also makes for much simpler image handling code.

There are two different ways that the Translation Kit can be used in our programs. Which one we should use in a given situation depends on our needs. The easy way is just for loading bitmaps. The more involved way is needed for instances when we are both loading and saving or when we are using the Translation Kit for something other than bitmap images. We'll be using the easy way today, the details of which we'll be seeing shortly.

Resources

Application resources are a way to package data associated with your program inside the executable itself. It's a handy way to make sure that pictures used in your program are always available and haven't been overwritten by the user's recipe for quiche or something. Icons for your program are resources. Pictures used in the GUI should be stored this way, too. Certain other program information, such as the file types it handles and launch characteristics, are also stored in a program's resources.

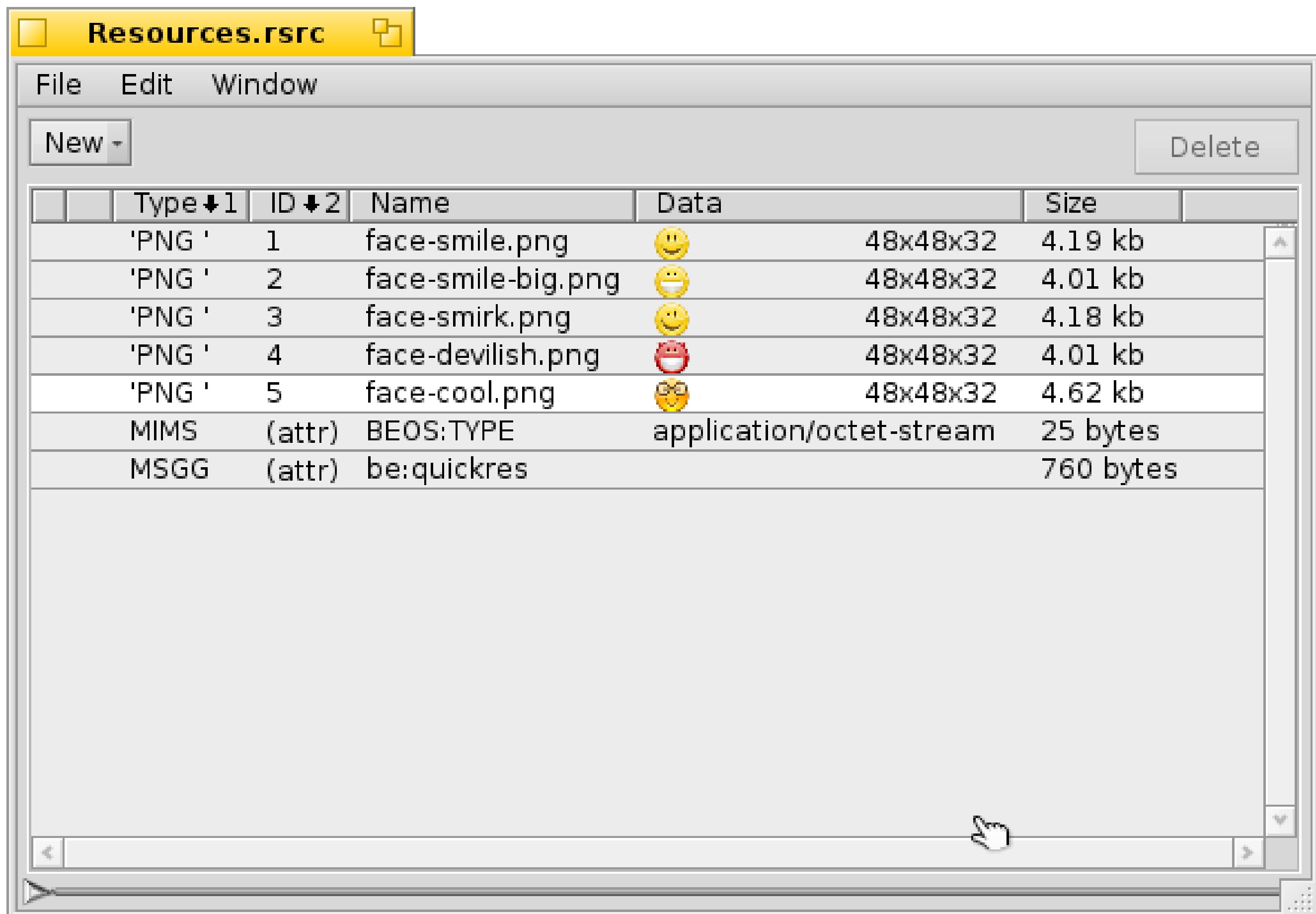
A number of tools are available for Haiku to add, delete, and edit program resources. QuickRes is the original, authoritative editor for program resources provided by Be. Unfortunately, it is not open source and not usable by GCC4-only Haiku installations. A resource editor called ResEdit is being developed for Haiku, but as of this writing, it has only been developed enough to be barely useful. rc (resource compiler) is a command-line utility which converts between text resource files and binary resource files. It is useful for making resources easily compatible with source control systems which don't deal well with binary files, such as CVS. xres is a command line utility which is used to manipulate resources. You can choose whatever tool you like, but for the purposes of these lessons, we will focus on QuickRes because it works well and eventual migration to ResEdit from QuickRes will be relatively painless when the time finally comes.

Although it's possible to directly edit a program's resources with an editor by opening the executable in the resource editor, this is not the usual – or recommended – way to add resources to an application. Resources are normally placed into a resource file that is added to the executable after it has been built. Resource files come in two formats: a binary format with a .rsrc extension and a text file format with a .rdef extension. Paladin supports both and will automatically bundle any resource files into your application if they are part of your project.

Project: Emo

Now we will be learning how to use both program resources and the Translation Kit in a little project called Emo. It loads a series of images from program resources and displays them one at a time, changing the picture whenever the user clicks on it. The sources can be found in 19Emo.zip. For the sake of space, only the parts of the code that are relevant to the lesson will be printed below.

First of all, let's start with the resource file for the project: Resources.rsrc. Creating a resource file isn't difficult, but let's look at the layout of the file when opened in QuickRes.



Each of the pictures has an entry in the file which shows its type, resource ID, name, size, and a miniature preview. Adding pictures to a resource file is just a matter of dragging them from Tracker and dropping them onto the QuickRes window. Deleting is just a couple of clicks, so nothing here is terribly complicated. The IDs are arbitrary, but they must be unique within each data type stored in the file. The name for each entry is optional, but highly recommended. You can add just about any kind of data you like to a resource file and use your own resource type codes, but try to keep them small because they have a direct impact on the executable's size. Large resources should really be kept in their own separate file. Let's move on to the code.

Surprisingly, the Haiku API does not have a view which just displays a picture, but luckily for us, making one isn't much work. First, let's look at the header, PictureView.h:

```
#ifndef PICTUREVIEW_H
#define PICTUREVIEW_H

#include <Message.h>
#include <Bitmap.h>
#include <String.h>
#include <View.h>

class PictureView : public BView
{
public:
                        PictureView(void);
                        ~PictureView(void);

      void              Draw(BRect rect);
      void              MouseUp(BPoint pt);

private:
      BBitmap           *fBitmaps[5];
      int8              fBitmapIndex;
};

#endif
```

This doesn't appear to be anything particularly special. It's not. The class has an array of five BBitmap pointers and an index which we'll use to point to the current one. It gets only a little more complicated in the main sources for PictureView:

```
#include "PictureView.h"
#include <TranslationUtils.h>
#include <TranslatorFormats.h>

// This class is our own special control. It loads five images from the
// application's resources and places them in an array. Once loaded, it resizes
// itself to exactly fit the first bitmap. This assumption is OK since they are
// all the same size and any problems are most likely to be our fault. Most people
// don't edit program resources and if they do, let them reap the consequences of
// their actions. >:D
PictureView::PictureView(void)
 :    BView(BRect(0,0,100,100), "picview", B_FOLLOW_LEFT | B_FOLLOW_TOP,
            B_WILL_DRAW),
      fBitmapIndex(0)
{
      // Load up all our pictures using a loop. There are 5 different versions of
      // BTranslationUtils::GetBitmap. This is one of two which load images from
      // program resources.
      for (int8 i = 1; i <= 5; i++)
      {
            BBitmap *smiley = BTranslationUtils::GetBitmap(B_PNG_FORMAT,i);
            fBitmaps[i - 1] = (smiley && smiley->IsValid()) ? smiley : NULL;
      }

      if (fBitmaps[0] && fBitmaps[0]->IsValid())
            ResizeTo(fBitmaps[0]->Bounds().Width(),
                     fBitmaps[0]->Bounds().Height());
}
```

```
PictureView::~PictureView(void)
{
}


// The BView's Draw() function is called whenever it is asked to draw itself
// on the screen. This is one of the few places where a BView's drawing commands
// can be called.
void
PictureView::Draw(BRect rect)
{
	// Alpha transparency is ignored in the default drawing mode for performance
	// reasons, so we will change the drawing mode to utilize transparency
	// information.
	SetDrawingMode(B_OP_ALPHA);

	// Set the foreground color of the BView to white
	SetHighColor(255,255,255);

	// Fill the BView's area with white. Like with most BView drawing commands,
	// the last argument is the color to use which defaults to the high color.
	// Other color choices are B_SOLID_LOW, which uses the background color, and
	// B_MIXED_COLORS, which mixes the high and low colors.
	FillRect(Bounds());

	// Draw the current bitmap on the screen
	if (fBitmaps[fBitmapIndex])
		DrawBitmap(fBitmaps[fBitmapIndex]);

	// Set the foreground color to black
	SetHighColor(0,0,0);

	// Draw a black border around the view
	StrokeRect(Bounds());
}


// Mouse handling is kinda funny. BView has three hook functions for the mouse:
// MouseDown(), which is called whenever the user presses a mouse button
// while the pointer is over the view, MouseUp, which is called whenever the user
// releases a mouse button while the pointer is over the view, and MouseMoved(),
// which is called whenever the mouse changes position while over the view. This
// gives you, the developer, a great deal of control over how your view reacts to
// any kind of mouse event.
void
PictureView::MouseUp(BPoint pt)
{
	// Go to the next image in the array or loop around to the beginning if at
	// the end.
	if (fBitmapIndex == sizeof(*fBitmaps))
		fBitmapIndex = 0;
	else
		fBitmapIndex++;

	// Force a redraw of the entire view because we've changed pictures
	Invalidate();
}
```

This class does all of the real work for the application, including handling the mouse clicks and showing the appropriate image. The code in MainWindow.cpp just creates a background view and a PictureView instance.

The highlight of this project is the call to `BTranslationUtils::GetBitmap()`. The only difference between loading a PNG file and a JPEG file here would be a different type specifier – no need to figure out how to read a JPEG file. All the heavy lifting has been done for you. Woohoo!

Going Further

There's quite a lot that you could do with what you know now. It's more a matter of getting to know the API and how to use it and less about writing C++.

- Try figuring out how to use BView's `Pulse()` hook function to automatically change bitmaps each second. Check the entry in the BeBook for exact details, but you'll need to set the flags sent to the BView constructor to `B_WILL_DRAW | B_PULSE_NEEDED` to use the `Pulse()` function.

Classes and Methods to Remember

BTranslationUtils

- `GetBitmap(const char *name, BTranslatorRoster = NULL)` – Looks for a file in the path `name` and, if not found, looks for a resource named `name`. If there is more than one with the same name, the first one is returned.
- `GetBitmap(uint32 type, int32 id, BTranslatorRoster = NULL)` – Returns the image contained in the resource identified by `type` and `id`.
- `GetBitmap(uint32 type, const char *name, BTranslatorRoster = NULL)` – Returns the image contained in the resource identified by `type` and `name`.