Programming with Haiku

Lesson 20

Written by DarkWyrm

All material © 2011 DarkWyrm

Drag and Drop Support

Drag and drop is one of the easiest ways to integrate your program with others in the system. Many times developers will create their new 133t project in a vacuum, forgetting the fact that the user does not use this new project in a vacuum, but often in conjunction with other programs. For example, working on a document in a word processor may also require occasionally jumping to an image editor for adding pictures to the user's document. By adding drag and drop support, you enable the user to work much faster. Depending on how you implement it in your programs, adding drop support can be dead simple, with drag support only a little harder.

According to the Be Book, within Haiku there are two ways to do drag and drop: the simple way and the negotiated way. Both rely on messaging to get the job done. The simple way packages some data into a message which is sent to the target when the button is released. The receiving program recognizes the data and takes appropriate actions. Negotiated drag and drop, while conjuring up images of a shady character in some back alley asking a mild-mannered businessman about a watch, works like this:

1. The sender places into the drag message format-related data fields, which we'll get to in a moment.
2. The receiver reads through the format fields and replies with a request for the desired data format and possibly the desired actions to be performed on the data.
3. The sender receives the reply, performs any requested actions on the data, and sends the data itself in the requested format.
4. The receiver reads the data from this second message and does whatever it chooses with it.

Simple Drag and Drop

If you do nothing else in your programs, handle simple drops. Very little effort on your part is required to do so because it's just a little extra code in a BView's `MessageReceived()` method which does this:

```
void
MyControl::MessageReceived(BMessage *msg)
{
	if (msg->WasDropped())
	{
		// Here's where your drop support goes.
		entry_ref ref;
		int32 i = 0;
		while (msg->FindRef("refs", i++, &ref) == B_OK)
			printf ("File dropped: %s\n", ref.name);
	}

	switch (msg->what)
	{
		default:
		{
			BView::MessageReceived(msg);
			break;
		}
	}
}
```

Our example here doesn't do very much. The most common form that drop support takes is dragging one or more files from a Tracker window and dropping them onto a control in your program. When this happens, the message has an `entry_ref` in the `"refs"` field for each file dropped. Once your program has read each ref, it can do whatever it likes. The above example just prints the name of each file dropped to the Terminal. Many times your program will check each files' type and open those which it supports.

Simple drag support is almost as easy. It will require the implementation of a few of BView's methods which we have not discussed before: `MouseDown()` and `MouseMoved()`. The main purpose behind `MouseDown()` is to start tracking the mouse. `MouseMoved()` is implemented in order to actually start the drag operation using the `DragMessage()` method. Once `DragMessage()` is invoked, the system handles the rest.

```
void DragMessage(BMessage *msg, BRect rect, BHandler *replyTarget = NULL);
void DragMessage(BMessage *msg, BBitmap *bitmap, BPoint pt,
                 BHandler *replyTarget = NULL);
void DragMessage(BMessage *msg, BBitmap *bitmap, BPoint pt,
                 drawing_mode mode, BHandler *replyTarget = NULL);
```

`DragMessage()`, which initiates the drag session itself, is the key function in implementing drag support. The message passed to it contains all of the data to be passed to the drop target. The BRect version displays an outline as the user is dragging. The other two versions of `DragMessage()` display a picture while the user is dragging. The third version also enables specifying a different drawing mode for displaying the bitmap, which makes possible nifty effects like drag previews which use transparency. `replyTarget` is only used for negotiated drag and drop, which we will learn about later – for simple drag and drop, we just leave it `NULL`.

Here is a simple example for a BView which performs drag and drop.

DragView.h

```
#ifndef DRAGVIEW_H
#define DRAGVIEW_H

#include <View.h>

class DragView : public BView
{
public:
                DragView(BRect frame);
      void      MouseDown(BPoint pt);
      void      Draw(BRect update);

private:
      BRect     fDragRect;
};

#endif
```

DragView.cpp

```
#include "DragView.h"

enum
{
```

```
	M_DRAG = 'drag'
};


DragView::DragView(BRect frame)
	:	BView(frame, "dragview", B_FOLLOW_LEFT | B_FOLLOW_TOP,
			B_WILL_DRAW),
		fDragRect(10, 10, 50, 50)
{
}


void
DragView::MouseDown(BPoint pt)
{
	// For our example, we only support dragging with the left button
	BPoint temp;
	uint32 buttons;
	GetMouse(&temp, &buttons);

	if (fDragRect.Contains(temp))
	{
		// SetMouseEventMask can only be called from within
		// MouseDown(). It forces all mouse events to be sent to this
		// view until the button is released. This saves us from having
		// to manually code the mouse tracking.
		SetMouseEventMask(B_POINTER_EVENTS, 0);

		BMessage dragMsg(M_DRAG);
		dragMsg.AddInt32("buttons", buttons);
		DragMessage(&dragMsg, fDragRect);

		// If you allocate the drag message on the heap, make sure
		// you delete it after DragMessage() returns. If you use a
		// BBitmap version of DrawMessage(), DON'T delete the bitmap
		// after calling DragMessage() because the system will do
		// that for you.
	}

}


void
DragView::Draw(BRect update)
{
	SetHighColor(0, 0, 160);
	FillRect(fDragRect);

	DrawString("Try dragging the square", BPoint(fDragRect.left,
			fDragRect.bottom + 50));
}
```

As you can see the only significant part of this code is the section in `MouseDown()`. Everything else is just fluff to make the demo a little nicer.

Negotiated Drag and Drop Support

This kind of drag and drop is only rarely used because the simple version requires very little effort and typically gets the job done. Still, this version provides a much greater opportunity

for other programs to interact with yours because the negotiation process bridges the gap between two programs that otherwise know nothing about each other.

Drag Initiation

While the actual process of detecting and initiating the drag operation is the same as for simple drag and drop, the message itself needs to follow a particular protocol. First of all, the `what` field of the message needs to be `B_SIMPLE_DATA`. Second, the message needs to add the formats your program is willing to provide in a BMessage. Last, the supported actions for the data need to be added. There are also some optional fields which may be added that are detailed below.

The data formats supported by your program are placed in the `be:types` field as a list of MIME type strings. If your program is willing to pass data by way of a file – which is a good thing for large chunks of data – your program will need to add the `B_FILE_MIME_TYPE` constant, defined in `<MimeType.h>`, to the `be:types` field and populate the `be:filetypes` and `be:type_descriptions` fields. Note that if your program can pass data via a BMessage or a file, you should add the other types first and add `B_FILE_MIME_TYPE` last. If your program will only send data via a file, then `B_FILE_MIME_TYPE` should be the first (and only) value in the field. The `be:filetypes` field is just like `be:types` – a list of MIME type strings. `be:type_descriptions` holds a string description of the format which may or may not be displayed to the user.

Not only is the data format negotiable, the delivery method is also. The `be:actions` field is populated by your program with constants for supported actions on the data. There are four of them plus two legacy actions which are specific to Tracker, but are no longer supported.

Constant	Description
`B_COPY_TARGET`	Your program can send a copy of the data.
`B_MOVE_TARGET`	Your program can send a copy of the data and delete the original, "moving" the data to the receiving program.
`B_LINK_TARGET`	Your program can send a symlink to the data.
`B_TRASH_TARGET`	Your program can delete the data without sending it. One possible use for this is to drag something from your program to the Trash.
`B_COPY_SELECTION_TO`	This one is Tracker-specific. It is sent when there are one or more items in a Tracker window and it is willing to copy them somewhere. The destination is specified in the `refs` field of the negotiation message if this action is requested. While supported at one time and documented in the Be Book, as of this writing Tracker no longer supports this action, but it may be in the future.
`B_MOVE_SELECTION_TO`	This is also Tracker-specific. It works just like `B_COPY_SELECTION_TO`, but moving the files instead. This, too, is no longer supported by Tracker but may be in a future revision.

Optional Drag Message Fields

Although `be:types` and `be:actions` are required fields, there are a few optional fields besides them. `be:clip_name` is a suggested name for the dropped data which the receiver can ignore or use if your program provides it. `be:originator` and `be:originator_data` are used for asynchronous messaging.

If your program needs to do negotiated drag and drop asynchronously, then using the `be:originator` and `be:originator_data` fields may not be a bad idea. The intent behind the fields is to provide a way for your program to identify its own drag and drop negotiation messages and track state information during the process. `be:originator` should be filled with something that identifies your program. What exactly this entails is up to you – it could be your program's signature, handler token for the sending BView, or something else. The only code that will be interacting with it will be your own. `be:originator_data` can be filled with data that your program will need to complete the drag and drop negotiations. Of course, if your program doesn't need this field, it can be easily left out without any ill effects.

`be:data` was used in the original drag and drop protocol, but it has been since deprecated. It is mentioned here in case your program interacts with very old or very long-lived BeOS applications.

The `_drop_point_` and `_drop_offset_` fields are BPoint fields which give some coordinate information for the drag. Both are automatically added by the system. `_drop_point_` contains the screen coordinates of the mouse when the data was dropped. `_drop_offset_` is the distance from the top left corner of the drag rectangle.

Drag Negotiation

Once the drag recipient has received the initial drag message, it is up to the recipient to decide what it wants done and how. The actions listed in `be:actions` are to become the `what` value for the reply. The message itself is expected to have at least one of these fields:

Field	Description
`be:types`	One or more strings specifying the formats the recipient is willing to accept data in a BMessage.
`be:filetypes`	One or more strings specifying the formats the recipient is willing to accept data in a file.
`directory`	An `entry_ref` pointing to the directory in which the data file is to be created. The recipient is responsible for ensuring that the directory is valid.
`name`	A string containing the name of the file for the data. The recipient is responsible for ensuring that the name is available.

None of the fields are required if they don't apply. The sender doesn't even have to respond to a `B_TRASH_TARGET` message – it just needs to toss the requested data in the can.

Receiving the Data

A data message is sent if and only if a message-based format was specified in the negotiation message – no message is sent if data is to be sent only by file. The data message has the value

`B_MIME_DATA` and contains one field. The field's name is the MIME type for the data, e.g. a "text/plain" field containing a string which is the requested data. Once the data has been received, the recipient is free to do with it whatever it likes.

The Be Book does not mention the procedure for obtaining data from a file-based negotiation and there are no known applications as of this writing which use negotiated drag and drop, but there is still a way to handle this. Considering that file-based transfers are best used for large chunks of data, asynchronously monitoring for the data file is best. Using the Node Monitor would be the easiest way to watch for the creation of the file.

A Negotiated Summary

No wonder negotiated drag and drop isn't used much! Here is a quick summary to remember everything:

1. The sender makes a `B_SIMPLE_DATA` message with format strings in `be:types` and `be:actions` and possibly `be:filetypes`, `be:type_descriptions`, `be:clip_name`, `be:originator`, or `be:originator_data`.
2. The recipient replies with a message of the action desired, such as `B_COPY_TARGET`, and attaches the desired `be:types` and possibly `be:filetypes`, `directory`, and `name`, depending on the action requested and the methods desired.
3. If the data is to be sent via BMessage, the sender replies to the negotiation message with a `B_MIME_DATA` message with one field named after the data's MIME type which contains the data.

Implementing Simple Drag and Drop

Our ColorWell class should definitely allow drag and drop, both for receiving colors dropped onto it and dragging its color elsewhere. There are no official standards, the protocol used by an ancient color picker called roColour is supported by many applications, including Tracker, so we will follow suit. Change the beginning of the `MessageReceived()` function of ColorWell.cpp to this:

```
void
ColorWell::MessageReceived(BMessage *msg)
{
	// Handle simple roColour-style drag and drop
	if (msg->WasDropped())
	{
		rgb_color *c;
		ssize_t size;
		if (msg->FindData("RGBColor", B_RGB_COLOR_TYPE,
			(const void **)&c, &size) == B_OK)
		{
			SetValue(*c);
			return;
		}
	}

	if (!msg->HasSpecifiers())
		BControl::MessageReceived(msg);
```

That little chunk of code is all that we need to enable the user to drop colors onto a ColorWell control and it will change to that color. The code for dragging colors to other targets involves a bit more. We'll need to implement `MouseDown()`.

```
void
ColorWell::MouseDown(BPoint pt)
{
	SetMouseEventMask(B_POINTER_EVENTS, 0);

	// Create a color bitmap to show the color drop. This version of the
	// BBitmap constructor allows for BBitmaps which can accept and be
	// drawn upon by BViews.
	BRect r(0, 0, 15, 15);
	BBitmap *bitmap = new BBitmap(r, B_RGB32, true);
	BView *view = new BView(r, "", 0, 0);

	bitmap->Lock();
	bitmap->AddChild(view);
	view->SetHighColor(fColor);
	view->FillRect(view->Bounds());
	view->Sync();
	bitmap->Unlock();

	BMessage msg;
	msg.AddPoint("click_location", pt);
	msg.AddData("RGBColor", B_RGB_COLOR_TYPE, &fColor,
		    sizeof(rgb_color));

	DragMessage(&msg, bitmap, BPoint(12, 12));
}
```

There are three main sections of code to our implementation of `MouseDown()`. The first section is the call to `SetMouseEventMask()`. This function exists for our convenience and can only be called from within `MouseDown()`. Our view will continue to receive mouse event messages until the user releases the mouse button, dramatically reducing the amount of code we need to write to handle the drag. The second section creates a small BBitmap which will add feedback during the drag operation. The last part creates the drag message, adds the color and click location to it, and initiates the drag operation.

Manipulating BBitmaps

Of special note in our `MouseDown()` code is the way that the BBitmap is created and modified. It is possible to create a BBitmap which accepts a BView just like a BWindow. BViews attached to a BBitmap draw on the bitmap instead of the screen. This is wildly convenient because we don't have to use an external library or learn another API to get some nice graphics. There are three constructors which enable this:

```
BBitmap(BRect bounds, color_space mode, bool acceptViews = false,
	bool needsContiguousRAM);
BBitmap(const BBitmap *source, bool acceptViews = false,
	bool needsContiguousRAM);
BBitmap(const BBitmap &source, uint32 flags);
```

The third version takes a series of flags. While there are others defined in Bitmap.h, here are the more useful ones:

Flag	Description
`B_BITMAP_CLEAR_TO_WHITE`	The bitmap is initialized to white when created.
`B_ACCEPTS_VIEWS`	The bitmap will accept child BViews.
`B_BITMAP_IS_CONTIGUOUS`	The physical memory allocated for the bitmap will be contiguous. This only really matters to drivers doing direct memory access.
`B_BITMAP_NO_SERVER_LINK`	This is a Haiku-only extension in which the bitmap is created without any connection to the app_server. This has the benefit of being very lightweight, but BViews cannot draw them with `DrawBitmap()`.

It is wise to understand some of the behind-the-scenes action which takes place when working with BBitmaps. For performance reasons, when a BBitmap is created, the app_server actually allocates the memory in an area shared with your program's BApplication object. While this speeds up `BView::DrawBitmap()` calls, it also forces your program to have a BApplication object to use them. Also, when you allocate a bitmap which can accept BViews, the app_server actually creates a BWindow counterpart on its side. Because of this, be conservative with the number of bitmaps which accept children. Otherwise, your application will crash. If you need to draw a lot of bitmaps, consider using an external graphics library.

Concluding Thoughts

After this thorough examination of Haiku's drag-and-drop facilities – especially the negotiated version – it may seem like this is more work than it is worth. In practice, adding drag and drop support to your programs is very little work because the negotiated version is used only very rarely. The simple method really is simple and giving the user the option to directly manipulate the interface of your program is a great way to save time for your target audience.