Object Links for Draw (file: `gvLinks.m`)

There are a number of things you have to do to implement Object Links in an application. Many of them are optional (depending on the level of functionality you want or are able to provide), but Draw does them ALL, so this should be a good reference point for you.

Please refer to the documentation in the system about Object Links to get an overall background in place before reading this document.

DrawDocument

Note first that Object Links only works on a document basis, so the **`GraphicView`** object cannot do links on its own. Only the **`DrawDocument`** object knows the name of the file, for example, and this is crucial to making links work. So, even though most of the implementation of Object Links in Draw is in **`GraphicView`** (actually, a category thereof found in **`gvLinks.m`**), you'll notice that it is the **`DrawDocument`** which creates (and is the delegate of) the **`NXDataLinkManager`**, etc. However, it usually forwards most of the messages it gets from the **`NXDataLinkManager`** onto the **`GraphicView`**.

Note also that a significant part of making Object Links work in Draw is all the messages that **`DrawDocument`** sends TO the **`NXDataLinkManager`** (grep for ``**`[linkManager`**'' in **`DrawDocument.m`** to find all those calls). **`DrawDocument`** is responsible for letting the system know when something about the document changes (e.g. the document is saved or closed or reverted to saved or whatever).

The ``Publish'' aspect of Draw is done via the **`saveLink:`** method in **`DrawDocument`**. You should be able to understand the implementation of this method after reading allRWd description of how Object Links works below.

It also calls **`updateLinksPanel`** from its **`windowDidUpdate:`** method to keep the Link Inspector panel up to date.

Now let's dive into how Draw actually implements the Object Links mechanism ...

Selections

The most important part of participating in Object Links is also the part that requires the most thought.   It is the process of representing a ``selection'' in your document.   It is appropriate that this be the most ``difficult'' thing to do in Object Links because it is the part of the Object Links mechanism that is purely application-dependent.   NeXTSTEP tries to do as much of the Object Links functionality for you, but it cannot do the things that are dependent upon what your application does for a living.

A ``linked-to'' selection (``source'' selections):

If you want people to link to documents in your application, you must be able to describe a selection that the user makes and then copies and pastes (and links) into another document in another application.   This selection description can be anything you want (it's a ``bag o' bits''), but it must survive and make sense no matter what happens to the source document (unless, of course, the items in the selection the user originally made eventually get deleted, but even that case you must detect).

How you represent this selection is really something you must think about carefully.   Draw actually has more than one way of representing the selection (this may well be true in the case of your application too).   Draw's selection-representation choice is purely for example purposes and you should, by no means, draw the conclusion that Draw's way is the only way (or even the best way) to represent a selection in an application that manipulates graphical elements (and obviously, Draw's way is not appropriate for text manipulation, spreadsheets, and other kinds of applications).

Okay, now that the disclaimer is out of the way, let's talk about how Draw represents selections that it exports to other applications. First, note that you can get the ``selection'' that the user has made in a **`GraphicView`**RWe at any time by calling the **`currentSelection`** method defined in this file. It returns an **`NXSelection`** object (the bag o' bits mentioned above) representing the current selection.

So, how does Draw represent is current selection?

1. **`[NXSelection allSelection]`**

This is the selection that is created when the user does Select All (and only in that case).   The **`allSelection`** method of **`NXSelection`** returns a ``special'' selection that Draw just chooses to know how to interpret.   Most applications will want to handle this special-case of **`allSelection`**.

2. Drag-Selection

When the user drags out a box to make a selection in Draw, the **`NXSelection`** that Draw uses to represent that state is the rectangle the user dragged out. Then, whenever Draw is asked about this **`NXSelection`**, it just intersects that rectangle with the current state of the **`Graphic`**'s in the view.

This is a particularly questionable type of selection because the user often ends up with ``not quite what she expected.''   On the other hand, it is a bit more accurate than selection type #3 below because it remembers a bit more of the semantics of what the user selected.   In any case, I have included it to show you what an alternative selection mechanism might be like and how to handle it.

The **`getRect:forSelection:`** method returns **`YES`** if the **`NXSelection`** passed to it is of the drag-select type (and, obviously, the ``rect'' that it ``gets'' is the rect the user dragged out to make her original selection).

3. Individual **`Graphic`** Selection

In this case, Draw just remembers the unique identifiers of each of the **`Graphic`**'s in the selection.   Then, when the system asks Draw about a selection of this kind, it looks in the current state of the Draw document for all of these items.   Note that it also includes any **`Group`** objects which include one of the **`GraRWf`**'s in the original selection.   Users can use this to, for example, have a background which they include in the original copy/paste link and then group whatever image they want to be the ``currently exported thing'' with that background.

The best selection mechanism would probably be some mixture of #2 and #3 (and perhaps some other types of selection mechanisms).   I've chosen these two because they are easy to understand.

The **`findGraphicsInSelection:`** method returns a **`List`** object with all the **`Graphic`**'s in the current document represented by the **`NXSelection`** passed to it.   This method can handle all three sorts of ``source'' selections (i.e. #1, #2, and #3 above).   This method calls the above-mentioned **`getRect:forSelection:`** method to handle case #2.

A ``linked-from'' selection (``destination'' selections):

If you allow the user to copy something from another application and Paste and Link it into the documents your application edits, you must be able to describe where in your document the thing was Paste and Link'ed.   This, too, is just a description of a selection in your document.

Since Draw only allows PostScript and TIFF (i.e. **NXImage**-handled data types) and RTF and ASCII (i.e. `Text` object-handled data types) to be Paste and Link'ed in (of course, these are the only types Draw allows to be normal-pasted in as well!), Draw represents this sort of ``destination'' selection by just remembering which **Image** or **TextGraphic** was created to import the data (since all objects in Draw have a unique identifier associated with them, this is an easy task).

There is a method implemented in the **Graphic** base class called ``selection'' which returns an **NXSelection** which describes the **Graphic** you sent the message to in terms of its unique identifier (i.e., it creates an **NXSeRWgion** and tosses the unique identifier of the receiving **Graphic** into the bag o' bits and returns it to you).   The **findGraphicInSelection:** method in this file searches through the document to find the **Graphic** with the corresponding unique identifier extracted from the **`NXSelection`** passed to it.

Importing/Exporting Link Data

Okay, so now you understand how Draw creates an **`NXSelection`** object to represent either a selection made in a Draw document which is going to be exported to another application via Copy/Paste and Link and also how it represents a selection which describes which **`Graphic`** is the receiving end of an Object Link.   Let's quickly talk about how Draw exports a link and how it imports a link.

Exporting:

It exports a link via the method **`writeLinkToPasteboard:types:`**.   This is a very simple method, but very important to the Object Links mechanism.   It does two distinct things:

1. It creates and writes an **`NXDataLink`** object to the **`Pasteboard`** which includes all the stuff another application would need to know to create an Object Link to the current selection the user has made in Draw (primarily just the **`currentSelection`** itself and the data types Draw will export (e.g. PostScript and TIFF)). This is the most important thing this method does.

2. It writes all of the links in the **`GraphicView`** to the **`Pasteboard`**.

Why, you may ask, does it do this? Well, if you copy an **`Image`** in Draw which is actually the destination of an Object Link (not the source of a link, but the DESTINATION), then if you pasted that **`Image`** into another Draw document, you want it to keep its ``linkness'', i.e., you want the thing you pasted to also get updates when the source of that **`Image`** gets updated. Simple, huh?

Which brings us to the

fc1readLinkForGraphic:fromPasteboard:useNewIdentifier: method.   It's the thing that is called every time you paste a **`Graphic`** into Draw to get that pasted **`Graphic`** properly linked up with the **`NXDataLinkManager`** in the Draw document you paste it into.

It is implemented by calling the **`addLinkPreviouslyAt:fromPasteboard:at:`** method in **`NXDataLinkManager`** which simply reestablishes the link that **`Image`** has to another document (that was at *oldSelection* in the old document) by setting the destination selection of the link to the selection which represents the **`Image`**'s location in the new document (**`[graphic selection]`**).

The *useNewIdentifier* thing is so that if you copy and immediately paste back into the same document, no actual change occurs (this is important in case someone else is linked to something that is in turn linked to something else--just trust me, you want copy/paste from/to the same document to be a net ``no-change'' in the document as far as links are concerned).

Importing:

Importing a linked thing happens **only** via the **`addLink:toGraphic:at:update:`** method.   No where else in Draw is a linked thing added to the document (except, of course in **`readLinkForGraphic:fromPasteboard:useNewIdentifier:`**, but that's a special case).

Let's quickly summarize how this method works:

The arguments are simple.   The *link* is an **`NXDataLink`** gotten either from a file (**`.objlink`**) or from a **`Pasteboard`** (during Paste and Link) or was alloc/init'ed pointing to a file.   See the callers of **`addLink:...`** to see about that.   The *graphic* is just an **`Image`** or **`TextGraphic`** created from the same **`Pasteboard`** we got the *link* out of or from the file that we alloc/init'ed the k to point to.   If *graphic* **`nil`**, then we probably got the *link* from a **`.objlink`** file, so we don't actually know what kind of data we're talking about yet. We take care of that first thing in this method (see the next paragraph). The *update* argument is used to describe whether this is a normal link, or a link which is never updated (link buttons and links to files represented by the file's icon are the classic examples of these) or a link which must be updated immediately because we don't yet have any data for it (again, see the next paragraph).

The first if-statement handles the case of pasting or dragging in an **`NXDataLink`** without any corresponding data (i.e. no PostScript or TIFF to go with it). This is always the case for a **`.objlink`** file, and could conceivably be the case for a Copy/Paste and Link if the app that copied the stuff in only copied the **`NXDataLink`** and forgot to (or chose not to for some reason) put the thing being linked to itself in the **`Pasteboard`**. Anyway, what that first if-statement does is figure out what data types the **`NXDataLink`** deals in (again, e.g., PostScript or RTF or some such) and creates an ``empty'' **`Graphic`** (an **`Image`** or **`TextGraphic`**) which will be filled in immediately when, later in the method, we force an **`updateDestination`** to occur (setting the update mode to **`UPDATE_IMMEDIATELY`** is what does this).

The second if-statement is what's doing all the work, of course.   First, it asks the **`Graphic`** which is going to be the destination of this Object Link (it'll be an **`Image`** or **`TextGraphic`**) for an **`NXSelection`** object which represents it.   Then it ``adds'' the link to the **`NXDataLinkManager`**.   If the link is successfully added, then we let the **`Image`** or **`TexRWpphic`** know about the link to it (only so that we can ask for it back later, the **`Image`** and **`TextGraphic`**'s never actually do anything themselves with the link).   Next, we put the **`Graphic`** into the document using the standard **`placeGraphic:at:`** method that we always use to add foreign data to the view (see **`gvPasteboard.m`**).

Finally, if we need to update the link immediately because we have no data, we do so by calling **`updateDestination`**, then ensuring that the update actually caused some data to flow over by seeing if the **`Graphic isValid`**. This works well for **`Image`**'s, but not so well for **`TextGraphic`**'s, I'm afraid (they always say they are valid!). Anyway, it's better than nothing.

That's it for exporting and importing links. Not so bad, is it?

Updating Links

Now, how do we actually update links (in either direction)? This, too, is simple. Whenever NeXTSTEP wants you to update someone else who is linked to you, it sends you the message **`copyToPasteboard:at:cheapCopyAllowed:`**. Whenever NeXTSTEP asks someone else to update something that is linked into your document, it sends you the message **`pasteFromPasteboard:at:`** (or **`importFile:at:`** if it's a whole file). All you have to do is to responds to these messages sensibly (you should assume that they can be called at any time). Return **`nil`** from these methods if the **NXSelection**'s in question no longer exist (in their entirety).

Draw's implementation of these methods is very straightforward (these methods are almost always really easy to implement if you already implement Copy/Paste or Services).

In **pasteFromPasteboard:at:**, it just finds the **Image** or **TextGraphic** represented by the NXSelection passed to it (see **findGraphicInSelection:**), then sends a message to that **Graphic** to reinitialize itself with the data in the **Pasteboard** passed to it.   It then updates the view and marks the view as edited.

The method **importFile:at:** is just like **pasteFromPasteboard:at:**, except that the source of the data comes out of a file instead of from a **Pasteboard**. This happens when you create an Object Link to a whole file without involving the application that knows how to edit that file (see **gvDrag.m** and the stuff where we drag a file into Draw with the Control key down (which means create a link to this file)).

In **`copyToPasteboard:at:cheapCopyAllowed:`**, there are basically two paths that can be taken depending on whether *cheapCopyAllowed* is true. *cheapCopyAllowed* just means that you can use the lazy pasteboard mechanism to the fullest because NeXTSTEP guarantees that no changes to your document can occur between the time this method is called and the time the lazy **`provideData:`** is called.   In other words, when *cheapCopyAllowed* is true, we don't actually have to write the Draw objects in the selection to the pasteboard by value, we can simply write a reference to them.

So, in Draw, when *cheapCopyAllowed* is true, we just declare that we can provide PostScript and TIFF, but write neither to the **`Pasteboard`** (we'll provide it lazily).   Of course, when the lazy **`provideData:`** is called, we have to know what part of our document to put into the **`Pasteboard`**, so we simply drop in the **`NXSelection`** that we were asked to **`copyToPasteboard:`**.

Thus, in the *cheapCopyAllowed* case, the actual work of putting the data in is done in the INSTANCE method **`pasteboard:provideData:`**!   It is okay to use the instance as the owner of the **`Pasteboard`** because thRWrstem has guaranteed us that our document would not be changed (especially not FREED!).   The implementation of **`provideData:`** is really simple since we already had methods lying around that could write the PostScript or TIFF for a list of **`Graphic`**'s into a stream (**`write{PS,TIFF}ToStream:usingList:`**).   We get the list of **`Graphic`**'s to write from the **`NXSelection`** we put in there (see how this all just dovetails together? Idn it great?).

When *cheapCopyAllowed* is not true, then we just do what we normally do when the user hits Copy, we just do it with the **`Graphic`**'s that are in the passed **`NXSelection`** instead of the ones in the current selection.   We plop the list of **`Graphic`**'s into the **`Pasteboard`** and let the normal lazy **`Pasteboard`** stuff take care of the rest (the CLASS method **`pasteboard:provideData:`** in this case, see **`gvPasteboard.m`**).

Miscellaneous methods.

There's a few other little methods you may want to implement.

You'll probably want something akin to **`updateLinksPanel`** which just keeps the Link Inspector panel up to date (it is called from **`windowDidUpdate:`** in **`DrawDocument`**).

The **`showSelection:`** method in **`gvLinks.m`** (the actual names of some of these methods is different, see **`DrawDocument.m`** which forwards them onto **`GraphicView`**) is sent by NeXTSTEP when the user asks to show the source of an Object Link that comes from your document.   It is very nice to respond properly to this message (the user will certainly be expecting this to work in your application).   It is very easy for Draw to get the bounding box of the **`Graphic`**'s in the passed selection (it even draws the little drag-sRWstion rectangle if that's the kind of **`NXSelection`** it is) since we already have methods lying around that, given a list of **`Graphic`**'s can find their bounding box.

There is one notable thing that Draw does when showing source selections.   It uses the fact that all the drawing done in a Draw document is actually done in an off-screen cache and composited to the screen.   When Draw shows a source selection, it draws them directly to the on-screen window, then remembers the areas in which it draw (this is the **`invalidRect`**).   Then, it leaves the source selection showing until the user touches the view (see **`drawSelf::`**) at which point, it just blows the **`invalidRect`** away by copying that rectangle from the off-screen cache.   If you do double-buffering like this in your application, this trick is easy and effective.

The **`breakLinkAndRedrawOutlines:`** method in Draw is what keeps the link outlines up-to-date. When the user chooses Show Links from the menu, all things that are linked into your document should show a border around them (there is a NeXTSTEP function to draw this border). These borders are kind of the opposite of what the **`showSelection:`** method draws (i.e. **`showSelection:`** shows what Object Links originate in your document, and Show Links shows the Object Links that are linked into your document from somewhere else). The argument to **`breakLinkAndRedrawOutlines:`** is a *link* that was recently broken by NeXTSTEP (this method is called from **DrawDocument**'s **`dataLinkManager:didBreakLink:`** and **`dataLinkManagerRedrawLinkOutlines:`** methods which are sent by NeXTSTEP).

If the *link* argument is **`nil`**, it means that no link was broken, so Draw just redraws **all** the link outlines. If the argument is not **`nil`**, then the method searches for the **`Graphic`** which held that *link* and redraws it soRWtt it's outline goes away.   Furthermore, if it was a link that didn't show the source data (i.e. it was a link button or file icon or something), that `Graphic` is removed from the document (since it is now disconnected and useless--don't we all feel that way sometimes?).

Tracking Links

Finally, there is the task of tracking the sources of links.   This is optional behaviour but is really a must if you want to implement Continually updating links.   The idea here is that you tell NeXTSTEP when a selection which is the source of a link which you export has changed.   Otherwise, NeXTSTEP has to assume that every time your document is edited that all the links that you export have changed.   In other words, this is a performance optimization, but a valuable one.

Note that you don't have to track **all** your links, only the ones that are showing up in other documents that are on the screen at the same time. NeXTSTEP (through the **`NXDataLinkManager`**) will tell you when to start and stop tracking links (NeXTSTEP is such a polite entity, is it not?).

Draw tracks links very easily by making the assumption that if any region of the Draw document which is redrawn overlaps the source of a link, that link must have changed and needs to be updated.   Since Draw has a nice knothole through which all updates to the document go (**`cache:`**), this is a mere matter of keeping track of the boundaries of the sources of links which Draw exports.

Draw does this by keeping a **`Storage`** object which a struct in it that has three pieces of information.

1. The rectangle which encloses the source of the link.
2. The link in question.

3. What type of selection is involved (all, drag or normal).

Almost every time **`cache:`** is called (sometimes **`cache:andUpdateLinks:`** is called with `NO` as its argument, but not very often, grep the code and you find out the times when that is necessary) the method `updateTrackedLinks:` is called.   This method has a two-fold purpose:

1. Notify the **`NXDataLinkManager`** if any ofRX currently-being-tracked links intersects the area which was just **`cache:`**'ed.
2. Reevaluate the bounds of any of the source selections that intersects the area which was just **`cache:`**'ed.

We must do step #2, because the thing that might have caused **`cache:`** to get called could have been that the user resized one of the objects which are linked to.   Thus, step #2 is not necessary for the drag-selection (since that originally dragged-out box can never ``change size'') and **`allSelection`** cases.   Step #2 is implemented simply by getting the **`NXSelection`** from the link, calling **`findGraphicsInSelection:`**, then calling the already-existing **`getBBox:of:`** method.

All we do in **`startTrackingLinks:`** and **`stopTrackingLinks:`** is add/remove structs from the **`Storage`** object.

Summary

Well, that's all there is about links and Draw.   I hope this document is illuminating.   The take-home messages should be that Object Links should be simple to implement if you already implement Copy/Paste and/or Services.   The only ``hard part'' might be figuring out how to represent a selection in your document.   Good luck with that part. :-)