# Dragging stuff into Draw (file: `gvDrag.m`)

This file is intended to handle things that are dragged into a Draw document (files and colors mostly). There are a couple of things to keep in mind about the files case. The first is whether the file being dragged in is being ``linked'' (via ObjectLinks). The second is whether we are going to display the contents of the file or just the file's icon (or even just the little link button icon).

## Registering

The basic idea is that we call **`registerForDragging`** in **`GraphicView`**'s **`initFrame:`** and in `awake`. That calls the **`registerForDraggedTypes:count:`** method to let the system know which types of things (represented by **`Pasteboard`** types) we are interested in having dragged into our view.

## Reacting to a Dragging Session

Next, we implement some of the dragging protocol as follows:

1. **`draggingEntered:`**

This is sent to us the first time something that we are interested in enters our view. We respond by letting the system know whether we are actually interested in the dragged thing at the time the thing is dragged into our view (depending both on the state of our view and where the drag is) and what operations we will support.

Basically, if the dragged thing is a color, we return that we will perform the ``generic'' thing on that color if dropped only if the **`Graphic`** that is currently being dragged over can deal with having a color dropped on it (the **`acceptsColor:atPoint:`** method is what we have created to figure this out).

Otherwise, we will accept the dragged thing if it is a file of any sort or if **`NXImage`** says that it can make some sense out of the **`Pasteboard`** of stuff being dragged in (this last case is very rare, applications rarely let you drag raw EPS out of themselves and into other apps--maybe more applications will start doing this in the future, we'll have to see).

In reality, we shouldn't say that we can accept **any** sort of file. We can really only accept files that **`NXImage`** can handle, RTF files, and (most of the time) plain files (because their usually ASCII). We can easily determine if a filename is one **`NXImage`** can handle (and we can obviously tell whether it's an RTF file), but the only way we can tell if a file is a ``plain'' file (i.e. not a WriteNow file or some such) is by asking the Workspace Manager (see the message we send to **`[Application workspace]`** below). Unfortunately, since the drag protocol is a synchronous blocking protocol between the app and the Workspace, we can't talk to Workspace in the middle of the drag (what a drag, huh?).

So, what we do is just accept any file, and, if you drop a WriteNow file in (without linking, of course), then we just ignore it.

Speaking of linking, if the link key (the Control key) is down during the drag, then we really can accept any file because we can just drop its icon into the Draw document (like Mail does). Then we just use ObjectLinks to make the double-clicking on it open the file up. We also provide the user the option of creating a little Link Button instead of the file's icon.

2. **`draggingUpdated:`**

This is called repeatedly as the dragged thing is dragged about our view. Again, just like **`draggingEntered:`** we return whether we are interested in accepting the dragged thing depending on where it currently is in our view and our current state. This method must be pretty fast, and it is, because we have already examined the contents of the **`Pasteboard`** in **`draggingEntered:`**.

Basically all we really do of interest here is constantly reevaluate whether the **`Graphic`**

underneath the dragging is willing to accept a color (but only, of course, if it is a color we are dragging).

3. **`performDragOperation:`**

This method is called just as the user lets go of the thing she's dragging. This is normally where you do the work that the drop of the dragged thing causes. And, indeed, if the dropped thing is a color, we update the color of the dropped-on **`Graphic`** here.

Unfortunately, often dropping something on a Draw document causes two very time-consuming things to happen. First, a complicated PostScript or TIFF image might have to be drawn. Second, a question might need to be asked of the user about how to deal with the dropped thing (this happens when you link a file in and Draw wants to know whether you want the contents of the file to appear in Draw, the file's icon, or a link button).

Thus, we don't do the work that the drop results in in this method, instead, we wait until after the drag and drop is fully complete (as far as the system is concerned) and do the work in ...

4. **`concludeDragOperation:`**

This is called after the drag and drop is completely done (and Workspace is out of the loop). Thus, if there's an error, we can't do the ``slide-back'' animation. It's a bummer, but there's really no way around it in 3.0.

This method is implemented by first looping through any filenames that are in the dragged **`Pasteboard`** looking either for **`NXDataLink`** files (**`.objlink`** files) which represent links (these are similar to the little things dropped in the filesystem in the Publish/Subscribe mechanism) or for files that **`NXImage`** can handle (TIFF, EPS, other formats if you have filters lying around), or for RTF or plain files that the `Text` object can handle.

**createGraphicForDraggedLink:at:** handles the **.objlink** files, and **createGraphicForDraggedFile:withIcon:at:andLink:** handles all other files. **createGraphicForDraggedLink:at:** is implemented simply by instantiating the link from the file and then calling the one method in all of Draw that actually adds a linked thing to the document (**addLink:toGraphic:at:update:** -- see **gvLinks.m**). **createGraphicForDraggedFile:withIcon:at:andLink:** also calls that same method if we are linking the dragged file in. The method called when you do a Paste and Link from the menu also calls that method.

Finally, if we can't find any files that we can do anything with, we just call the same method that is called when you hit Paste in the menu (except that we go straight to handling only non-Draw formats). Nice code reuse, huh? Again, this last thing is pretty rare (at least it is today, who knows what tomorrow will bring?).

By the way, you may will ask what this **ERROR** thing is all about. Well, the create

functions return **YES** if they were able to successfully incorporate something into the document, **NO** if they weren't, and **ERROR** if they found a problem with the thing being incorporated (i.e. a PostScript error in a dragged EPS file, for example).