Graphics Workshop
A Graphical Bitmap Utility Program

by

Alex Raftis

Trademarks

Table of Contents

Table of Figures

Introduction

GraphicsWorkshop allows the user to read, write, view, and manipulate a variety of bitmap formats, however, this is only the surface. Underneath GraphcsWorkshop is a complex method for controlling run time loadable converters, allowing users to change the list of bitmaps availble to them. There's also another object controlling the use of these objects to make life simple for the application programmer.

Reasons for the Project

Graphics Workshop was conceived because I'm an avid collector of computer images, which come in many formats, and having become a recent owner of a NeXTstation Color™, I wanted a method to view and use my images. However, I like to store my images in their original formats. This made things difficult on the NeXT™ where most programs only deal with TIFF's.

Conception

My primary consideration when thinking up the program was usability. I could've written a program that loaded and saved bitmaps within the confines of one application, but I wanted more. I figured it would be a lot easier if the program I was using could read the bitmaps directly rather than needing to go through another application first. I also concluded that every application writer that needs to load and save bitmaps should not have to reinvent the wheel each time.

I was also interested in flexibility. I figured that while I might support one format or maybe six, I could never predict what formats might appear in the future, nor what formats all people might want to use. This led to the concept of loadable converters, allowing anyone to write a converter that could be used from within any application that uses my objects.

This eventually led to an almost completely modifiable, re-writable system of control objects. At compile time, the programmer needs only to link in about thirty lines of code in one object. Everything else is loaded from disk at run time. This means that if I update my software, or for that matter, if someone else does, application programmers don't need to update their code. It's already done for them.

Finally, while I concluded that each individual converter should be able to load and save, the converter writer shouldn't have to worry about mundane things like pixel retrieval and image processing. To this end, I include a set of bitmap utilities which are at the disposal of the converter writer. This means that converting a twenty-four bit color image to a one bit black and white image is only a few lines of code.

Project Summary

When I initially sat down to begin design, I wasn't sure of the scope of the project. I eventually settled on a design that would allow for the loading and saving of just bitmaps. This led to the current design, which I can modify later.

In the earlier stages, the design required the linking of many objects into the control application. After a time, I decided that it was ridiculous to assume I could write the image utilities better than anyone else, so I decided to also make these generic and loadable. Then I got to thinking. I might also make mistakes that could be easily corrected if the control object was also generic. This finally led to the develop of the ControlLoader object.

The basic design of the project now stands with the ControlLoader object being called to link the bitmap control object which in turn links the image utilities and the converters. This simplifies greatly the dependencies programmers who use the object have on my source code. Take a look at the object hierarchy on page if you're interested in the final relationship of all the objects.

On a historical basis, the project was first conceived in 1990. From this point, I began to decide what I'd like in such a program. At the time, I was designing for an Apple IIGS™. When I later became deeply involved in NeXT and its environment, my mind drifted over to that platform. By early 1991, I'd pretty much decided on what to do and how to do it. As the summer of 1991 approached, I decided it would be a good time to begin my Senior Project at school, so I talked to a faculty advisor and arranged to begin the project, officially.

From that point, I dove into the programming. Development of the original plan was quick, as I entered the project with enthusiasm. This later slowed a bit as classes imposed time constraints on me. By the end of summer, however, the project was just about ready to enter alpha testing. At this point, it could handle everything but JFIF™ (.jpg) files.

Over my fall quarter of 1991 I spent time in alpha testing. This allowed me to get input from other programmers about what they might like to see. This led to a small increase in the generic nature of the control object, the addition of low level messages, and the ability to specify open parameters. It also saw the addition of error and copyright handling capabilities. I should also mention the elimination of numerous bugs.

This proceeded until the end of fall and into early winter. Just before I left for my winter break, I released the code to the world in general by placing

it on public ftp at sonata.cc.purdue.edu. I did this since mail distribution was becoming too large, and I also felt project and code were stable enough to warrant public consumption.

Once on the internet, many additional problems with installation and bugs were reported. This led to a few new developments. To begin, I now distribute the different parts of the program as Installer packages. This makes installation for non-UNIX experts much easier. I also worked to decrease dependencies on libraries that NeXT does not distribute to all NeXT owners. Finally, I modified the GIF converter to handle a more varied number of GIF's, since I was running into obnoxious errors with the code I was using.

GraphicsWorkshop

Overview

Graphics Workshop was written with three goals in mind. First, to provide a demonstration platform for my project; to make my project of use with programs that do not directly support the converters; and finally, to allow the use of a varied number of bitmap formats.

Basic usage is simple. When properly installed, the user is opened up into the world of graphic formats. To operate, you need to start by loading a bitmap. This can be accomplished via the open panel, one of the Workspace Manager's browser windows, or cut and paste. Any of these actions will open a new window to display the image.

Once you have an image, you may manipulate the graphic in simple ways, such as removing color information. In the future, I hope to greatly expand on these handling capabilities of the program by including edge enhancement, image rotation and scaling, and many other functions.

When you are done with the bitmap, you may save it into any of the supported formats found within your library folder, or you may copy it onto the paste board as a TIFF for inclusion in other programs.

Installation

Normal Installation

For most people, this is the only installation that you'll need to do.

To begin the installation, all you need to do is double click on the packages from the Workspace Manager. There are three packages for Graphics Workshop. The first, GraphicsWorkshop.pkg contains the executable. If you want to run Graphics Workshop, you'll need to install this package. It can go either into /LocalApps or ~/Apps, depending on your needs and the needs of your site.

The next package, Converters.pkg, contains the converters used by Graphics Workshop to load and save images. This package needs to be installed in a library folder for Graphics Workshop to work properly. If you installed the first package in /LocalApps, then this package should go into /LocalLibrary, and likewise, if you installed the first package in ~/Apps, then this package should go into ~/Library. This will create a sub-directory called Converters containing all the run time code needed by the program.

Finally, the last package, GWSource.pkg, contains all the source code Graphics-Workshop. You will not need to install this package on your

system unless you wish to work with the source code. For that matter, you can probably delete this package if you do not have the extended NeXTstep release or an interest in programming, since it will take up about 1.5 to 2 megabytes of disk space.

At this point, if you have an extended release system, Graphics Workshop should run properly. If it doesn't, check the following:

- Did you properly install the application? To check this, look in /LocalApps or ~/Apps and make sure there's a sub-directory named GraphicsWorkshop.app. If this doesn't exist in either directory, you need to install the GraphicsWorkshop.pkg package.
- Did you install the converters correctly? Check in /LocalLibrary or ~/Library for the existence of the Converters sub-directory. If this doesn't exist, you need to install the Converters.pkg package.
- The program runs, but aborts, indicating that the converters are not properly installed. If this happens, check in the Converters sub-directory for the existence of Bitmaps.tools, Bitmaps.controls, Converter.bcvt, and tiff.bcvt. At least these files must exist in order to run. If they do not, then the Converters.pkg file did not install properly. Try re-installing it.
- If you can run the program, but other users can't, the problem is most likely caused by not having both the converters and application installed in the /Local directories. Check and make sure they are there. If they aren't, you may need to move them from your own directories.You may also need to check file permissions and make sure all directories are readable and executable. Check the UNIX man page for chmod for more information on changing permissions. You can also do this via the file inspectors in the Workspace Manager.

If you don't have an extended release system, you'll need to also do the following. From somebody that has an extended release system, you'll need to copy the following files: /lib/libsys_s.a, /usr/lib/libNeXT_s.a, /usr/lib/libcs.a, and /usr/lib/libm.a. You should be able to get a copy of these from some with the extended release of NeXTstep, and yes, if you own a NeXT, you may legally copy these files. This is necessary because of the run time linking involved with the converters. Once these files are on your system, Graphics Workshop should work; however, if there are still problems, check the above list.

Advanced Installation

This section describes how to use the Install script. This script can be used to coordinate the building of various parts of GraphicsWorkshop, along with creating different installations of the program. Most people will not need to use this, as it's included for people who wish to change the source code.

The first thing you need to do is change into the source code directory. This will be wherever you placed the GWSource.pkg. From there, type “Install <return>”. This will start the install script, and you'll see the following:

```
Running this script will install GraphicsWorkshop on your
system. It give you
many oportunities to specify the arrangement of the
install. You also be asked
if you'd like to compile the various objects, even though
all the objects come
precompiled and ready to install. These are here in the
event that you have a
full developer's release and have modified code.

Please note that this install script works in cbreak
mode, so you do not nead
to hit return after single character input. Also, the
input lines are editbable
in a manner similar to tcsh. Here's some quick notes,
arrow keys more left
right, ^b back, ^f forward, ESC b back word, ESC f,
forward word, ESC B first
char, ESC F EOL, ^i toggle insert, ESC u Convert Word to
uppers, ESC l Convert
Word to lowers, and ^X delete line.

First of all, would you like the code built in debug
mode? Doing this will
generate profuse amounts of messages to the console
window, that can help a lot when debugging. If you're
just installing, say 'n'. Please enter y or n:
```

This question asks whether or not you'd like to build files in debug mode. You most likely will not need to worry about this and should just answer `n'. If, however, you're experiencing problems with the program and would like additional debug information printed to the console window, you can answer `y' and any makes will be done with the “-DDEBUG” flag.

Next you'll see the following:

```
You need to enter the name of the a directory telling
where you want to install
the software. The default is ~/Library/Converters, but
you are free to change
this location. Please note, however, that the location
must be
```

```
~/Library/Converters,     /LocalLibrary/Converters,     or
/NextLibrary/Converters.
However, since most people should not insert files
into /NextLibrary, one of
the other two is recommended. Also note, you will
probably need to be root to
install items into /LocalLibrary.

Your Directory Choice [~/Library/Converters]?
```

Here you need to input a directory for the installation of the converters. If you are not root, you'll see the above default. If you are root, you will see “/LocalLibrary-/Converters”. If you'd like to change the directory, enter the new one, otherwise, just hit return. Please note, however, that if the converters cannot be found in a standard library folder, the program will not function properly.

This will be followed by two questions:

```
The next questions are concerned with the converters.
These are the actual
objects that read and write bitmaps. You must have at
least one in your
library's converter folder for the program to function
properly.

Would you like to compile the converters? (y or n)?
```

Answering `y' to this first prompt will build the converters. You should not need to do this on initial installation. It's here to make modifying converters a little easier. It takes about five minutes to compile the converters. Progress will be shown by the appearance of `.' characters.

```
Would    you    like    move    the    converters    into
~/Library/Converters? (y or n)?
```

The next question asks if you'd like the converters moved into the library folder. You should answer `y' to this question if this is a first time installation. Later on, it may be appropriate to answer `n'.

You will then be asked very similar questions about two files, Bitmap.controls and Bitmap.tools. Once again, you probably will not need to compile the; however, you will need to move them into the library folder. Without these files, the program will not function.

The install script will then ask if you'd like to compile GraphicsWorkshop. Like the converters, you will probably answer no, but the next question,

```
Would you now like to modify GraphicsWorkshop's icon
header to include bitmaps
of the known converters? (y or n)?
```

should be answered yes. This will allow you to double click on a bitmap in the Workspace Manager and have GraphicsWorkshop launched to display it.

You will then be asked if you'd like GraphicsWorkshop installed on your system. If you answer yes, you will be asked where you'd like the program installed. If you are not root, the default will be in “~/Apps”, otherwise it will choose “/LocalApps” as the default.

Once you've determined the default directory, you may be prompted about what to do with a previous version of GraphicsWorkshop. You can decide to erase or back up the old version. Erasing it completely removes it from your system, while backing it up will rename the folder “GraphicsWorkshop.app” to “GraphicsWorkshop.app~”.

That ends the installation. You may need to log out and then log back in for the icons to be visible in the Workspace Manager.

Running

Perhaps the most straightforward method of running the program is by double clicking on its icon from the Workspace Manager. GraphicsWorkshop's icon is shown at the left, and is what you should see from within the Workspace Manager. You may, however, also see the icon at the right, assuming GraphicsWorkshop has been fully installed. Don't be too concerned if you cannot view the icon at the right. It will be associated with the bitmaps known by GraphicsWorkshop.

An important way to recognize bitmaps is by their file extensions. Here are the extensions recognized by GraphicsWorkshop, and their corresponding formats:

gif	Graphics Interchange Format
jpg	Joint Photographic extension Group, a.k.a. JPEG or JFIF
pbm	Portable Bitmap
pgm	Portable Graymap
ppm	Portable Pixmap
tiff	Tagged Image File Format
xbm	X-11 Bitmap

You should be able to double click on any of these file types to launch

GraphicsWorkshop. Doing so will run the program and bring up the desired bitmap. Note that the extension must appear as above. This is especially true of “tiff”, which must not be just “tif”. The latter is sometimes seen from bitmaps ported from PC's. Also note that “jpg” is sometimes seen as “jpeg” and should be shortened.

Menus

The main menu is pretty much like any other NeXT application. In brief, the Info panel leads to information about the application, the Document menu to options for opening and writing bitmaps, Edit to cutting and pasting, Tools to image manipulation options, Convert To to basic image conversion utilities, Services to standard NeXT services, Print to the print panel, Hide hides the application, and Quit exits the application.

Under the current release not everything will work, or work as expected. The main area you'll notice lacking is in services and printing. Currently, the services are not started, and no services are registered. With printing, a straight, unformatted bitmap is sent to the printer, often resulting incropped or misaligned images.

Info Panel...

The info panel will pop up a small panel describing me, the author, and telling you important things like the version number.

Help...

This panel is useful for finding out how to do things. In large part, it contains this document in a shortened form, but it is all in all easy and straightforward to use. It has two controls, a pop up menu and a scroll view. The pop up menu is used to select sections to view. You can then use the scroll view's slider to move about the text.

Preferences...

In its current incarnation, this menu item will always be disabled. In the future it will contain basic global items of control, such as selecting default conversion or the choice of whether or not to cache images.

Open...

This will pop up the open panel and allow you to make choices about opening a new bitmap. This panel is a bit more involved than what you might be used to from within most NeXTstep applications.

Below you'll find a snapshot of the open panel followed by a discussion of

how to use it. Take note of its operation, as this may appear in other applications using the same objects.

Within this panel, the browser, and the home, cancel, and okay buttons, all work like the standard NeXT Open Panel. However, please notice the custom view located between the browser and buttons. This allows for the specialization of open parameters for the converter objects along with a few other extra interactions.

The first button, in the top left, labeled “Info” will pop up a panel describing copyright information along with anything else the author of the converter wishes to point out to you. Once you've read the information in that panel, click its “Okay” button.

The next point of interest is the pop up menu on the top right. This menu allows you to select the input type for the kind of picture you wish to read. Normally, this button defaults to “All Types”. This default will allow you to read all manner of bitmaps, assuming there are converters for them on the disk. It will also allow you to select multiple file names.

When you click on the pop up, you'll be presented with a list of file extensions representing the types of bitmaps you can read. When you select a specific type of bitmap, a few things happen. First, the program goes out onto disk and loads that converter. It then asks the converter if it has anything specific that it'd like to ask of the user. If it does, this information will appear just below the “Format” label.

At this point, the text in the text box will change to reflect the name of the new format, and the Info button will return more specific information about a particular converter. You will no longer be able to select multiple file names. This is because the control object assumes that if you wish to specify certain parameters, they are particular to one image only.

Selecting “All Types” again from the pop-up menu will return the behavior to the original.

As a final point, you may be asking, why would a converter need to ask you about the picture it's loading. Well, this ability was added in order to load pictures that don't contain pixel configuration information. This behavior is exhibited in raw formats, sometimes used by various institutions during research. Currently, none of the default converters use this ability.

Make Image Index...

This allows for the creation of index files. Index files contain collections of bitmaps stored as small representations of their true selves. They also have the name of the original bitmap stored with them in the new bitmap.

When selecting this option, you will be presented with a slightly modified open panel. This panel allows for the additional input of three parameters, describing how the images should appear in the index image. The panel appears as the figure to the left.

You may use the Color Panel to choose the background color of the index file. White is the default. Click on the border of the Color Well to bring up the Color Panel, or you may select the Color Panel via the menus.

The Tile Width text box allows you to enter the dimensions of the images in the index file. The images will be automatically scaled to preserve their original aspect ratio, so the image will either be Tile Width tall or Tile Width wide, depending on the original aspect ratio of the bitmap.

The Tile Count textbox allows you to specify the number of index pictures that will appear in the x direction. The default is six images per line.

Save

This will save the current picture out to disk under its default format. If you are presented with a panel, as in the case of an untitled picture, or if you wish to save the picture in a different format, please see “Save As...”

Save As...

This option will pop up a panel similar to the open panel; however, there are a few differences in behavior. First, there is no “All Types” selection in the pop up menu. This is due to the fact that you can only save one image at a time in one format at a time.

Another point is that you do not need to add a file extension. The appropriate type will automatically be appended to the file name. Also note that if an extension you add is different from the type in the pop up menu, yours will be overridden.

The only other notable difference is that some of the converters do make use of the custom input parameters. Most notable, the TIFF converter will allow you to specify compression type and compression factor. This ability will also be added to the jpg converter when saving is implemented within it.

Revert to Saved

This isn't yet enabled, but when it finally is, it will allow you to re-load an image that has been modified.

Arrange in Front

This simply pops the selected window to the front of everything. This is useful when another application has tried to move something in front of your window.

Miniaturize Window

This has the same effect as clicking on the box in the upper left of a window. The window will be made small and appear as an icon along the bottom of the screen.

Close Window

This has the same effect as clicking the close button in a window.

v Warning There is no warning about closing modified windows. Be careful of what you do.

Cropping

This panel currently does nothing. It will eventually allow you to lasso a portion of the picture and cut away unwanted edges, as well as “auto-crop” the image to remove large areas of uniform color from the edges.

Data

This is the only panel in the tools menu that actually works. When displayed, it shows various information about the current picture, such a data configuration, bits per pixel, size, etc…

Edges

Hopefully, this will allow edge enhancement and recognition.

Gamma

This should eventually allow gamma color correction similar to that of Scene or Icon.

Orientation

This will allow the flipping and rotation of the image.

Black and White

This will make the picture's pixel depth one by applying a Floyd Steinberg Ditherizing Algorhithm.

2 Level Gray Scale

This will remove all but two bits of black and white data. Note that this method does not dither or halftone, so you will often loose color information.

16 Level Gray Scale

This removes all but four bits of gray scale data. Although similar to the previous option in method, many images will retain better color definition due to the addition of more colors.

256 Level Gray Scale

Like the previous two, this removes color information, but since it keeps eight bits of color information, the picture is much closer to photographic quality.

12 Bit Color

This simply removes extraneous color information. Basically, if you've got a 24 bit image, but wish to view it without dithering on a NeXTstation Color, this is the option for you. This can often make an image appear sharper, but it can also create color banding.

24 Bit Color

This makes an image a full twenty-four bits. Note, however, that this cannot add color information to an image, so expanding it to this depth will not make the picture any better. This was added mainly to account for bad conversions causing crashes in programs like Icon.

Strip Alpha

This removes all transparency information. This is really only useful for images that will be saved back to TIFF, since most converters strip the information anyway.

Invert

Inverses the image. This can be kind of psychedelic with color images.

That pretty much describes the menus and how to use the program. Read on if you'd like to see where the program might go in the future.

Future

Well, I'd have to say that the first major improvements will be in getting all the bugs out and all the features working. From that point, there are a few areas that I'd like to explore.

The first is image scanning. I'd like to add a scanner module to the program in a generic fashion that would allow multiple scanners to be used by any program in a fashion similar to the current converters. Unfortunately, I don't currently have the money for a scanner, so this option is still a way off in the future.

The other thing I'd like to do is to get a lot of people using my objects so that programs on the NeXT can be just a little bit more productive for everyone.

Programmer's Overview

The Objects

If you'd like to program with the underlying objects that allow for the reading and writing of bitmaps, then continue reading.

GraphicsWorkshop is really just an control application used to make my set of objects useful to everyday users. However, these objects are written in a manner that allows any NeXTstep programmer easy access to the converters. The objects are arranged in a simple hierarchy of “uses” relationships, with control objects using other objects to achieve their eventual goal. Here's a quick look at the objects, followed by a detailed description of each object and how to use it.

Converters

Converters lie at the heart of the control object. Without these, the system would simply be another inflexible system for viewing bitmaps. The converters give the user and the programmer the ability to dynamically work with many different formats, as well as allow the system administrator and user to tailor their individual needs.

Overview

Each converter consists of a group of methods stored in a Mach-O segment on the disk. These methods allow any form of controlling application to load and save images in the converter's type. Each converter, to insure future expandability, is obligated to respond to a certain number of messages. At present, all converters respond to the same numbers and types of messages, but this may change in the future.

All the converters are stored in a library folder on disk and are searched for and used in following order: ~/Library/Converters, /LocalLibrary/Converters, and /Next-Library/Converters. Where a converter is placed is not truly important, but this search order was chosen to allow end users the most flexibility in controlling the bitmap converters they wish to use.

Inside these folders, you should find files with a “.bcvt” ending. This signifies a converter. The root of the filename is the extension by which the bitmap will be recognized on disk. For example, the file “gif.bcvt” is the GIF converter and whenever a file with a “.gif” extension appears on disk, it can be recognized by this converter, assuming that the file is in valid GIF format.

If you do not plan to write your own converters, you've probably read enough of this section. However, should you like to expand on the

capabilities of GraphicsWorkshop, and other programs utilizing converters, then continue reading.

Currently Supported Graphics Formats

As of the first release, the following converters with the following limitations are supported:

Format Name	Extension	Reading	Writing	Limitations
TIFF	.tiff	Yes	Yes	None
GIF	.gif	Yes	Yes	Will not show text extensions
JPEG	.jpg	Yes	No	Cannot write out images
XBM	.xbm	Yes	Yes	Does not extend to XPM
PBM	.pbm	Yes	Yes	Only writes in binary mode
PGM	.pgm	Yes	Yes	Cannot control ascii/binary write
PPM	.ppm	Yes	Yes	Cannot control ascii/binary write

Note that only the TIFF converter supports the reading and writing of multiple images to one file.

For more information on these file formats, please see the appendices. If you'd like to write a converter and have it appear as part of the standard distribution, then please send the source, binaries, accompanying libraries (if any), and a readme file describing the format and any installation notes to alex@data.ACS.CalPoly.EDU. However, if you are not willing to distribute source, then I am not willing to include your code as part of my distribution. Please feel free to distribute your code via anonymous FTP or whatever means possible. Also note that I will not support the distribution of non-freeware code. You may ask for a donation, but you may not require it as part of my distribution.

Writing Converters

Assuming you already know a graphics format well, then writing a converter is a fairly simple task. The easiest method is to simply copy the files template.h and template.m from inside the …/GraphicsWorkshop/Converters directory to wherever you wish to do your coding. Rename these files to whatever is appropriate, and begin your work. When completed, compile the code and put the resulting “.o” or Mach-O file in an appropriate Library folder with the “.bcvt” suffix.

Beyond implementing all the methods, which will be discussed in the next section, there are a few things you need to keep in mind.

First, do not strip the resulting output files. Remember, these files must be linked at run time. Were you to strip the files, the run time linker would not have enough information left in the file to do anything with it.

Secondly, it's easiest to use only one source file, but you may use more, if you'd like. The easiest method here is to create a library of support files to “pre-link” against your resulting object. This is necessary because at run time, your converter will be linked against only the standard C library, the math library, and the NeXT and system libraries. This is equivalent to using “-lm -lNeXT_s -lsys_s” on the command line with cc. To “pre-link”, you will likely type “ld -r *input*.o *libraries…* -o *output*.bcvt”. This creates an intermediate link stage that can be used later on by the run time linker. Note that you cannot use shared libraries when using this method. For an example, see the make file rule for jpg.bcvt in …/GraphicsWorkshop /Converters/Makefile.

As a few final points, the following can help you when writing your converters. Your converter is always re-linked when it's selected via the open or save panel. This means that you can work on a converter without having to constantly re-launch the control application. Also, feel free to print things to “stderr”. Anything printed here will make its way to the terminal window or the Console window, depending on how you launch the control application.

General Defines and Data

One of the most important things to remember when writing converters is to properly support all the functions. Below is a description of each of the methods, the parameters they take, and what they need to return, along with default defines and class variables.

When your converter wishes to report an error state, it should report one of the following:

```
#define CONVERT_ERR_NONE                0
#define CONVERT_ERR_WARNING             1
#define CONVERT_ERR_FATAL               2
```

The first reports no error has occurred and your converter should report this when it has none. The second is used to report that something is wrong, but that it wasn't fatal. A good example of this is with a truncated file. You can still load the picture, but may have to return a shortened version to the user. The final is returned when something that prevents returning even part of the file occurs. This includes access permission problems and illegal or corrupted file problems.

Beyond the basic error states, the following are the types of errors that may be reported:

```
#define ERROR_NO_ERROR                  0
#define ERROR_UNABLE_TO_OPEN            1
```

```
#define ERROR_PERMISSION_DENIED         2
#define ERROR_BAD_FORMAT                3
#define ERROR_TRUNCATED_FILE            4
#define ERROR_NEEDSWINDOWSERV           5
#define ERROR_UNABLETOLINK              6
#define ERROR_UNKNOWN                   7
```

These are used by the controlling program to print error messages for the user. If you've got an error to report that does not appear on the above list, please use **ERROR_UNKNOWN**. The first on the list reports no error. The second that the file could not be opened. The next reports an access problem. The fourth reports a problem with the file image header or data. This is followed by a truncated file error (or warning), and the final is used to report an unknown error type.

Besides the above standard defines, each converter has one standard class variable, “errState”. This variable holds the current error level of the converter. It is an integer and should only be set to one of the **CONVERT_ERR_...** defines.

Method Overview

You need to implement, under the current version, only fifteen methods. Of these, not all need to be functional, but your converter should respond properly to each of them. Failure to do so can result in an uncaught exception which will crash the application. Future versions may, and probably should, support exception handling; however, do not expect it.

For beginners, there are *free* and *init*. Many of the methods do not use even these; however, you are welcome to use them. *init* is called when your object is first linked and *free* is called just before it is unlinked. You can use them to allocate storage or initialize variables. Neither of them accepts any parameters.

The next methods you should be aware of are the reading and writing methods, *readFromStream:from*:, *write:toStream:from*:, *readAllFromStream:from:,* and *writeAll:toStream:from:*. Of these, it is only expected that you support read and write, since many formats support only one image per file. Any read method not supported should return **nil** and any write method not supported should return **NO**.

In general, you should try to support as many of these methods as possible. As stated earlier, however, some image formats to not support multiple bitmaps. In this event, you can implement *readAllFromStream*:*from:* to return an NXImage with only one bitmap and *writeAll:toStream:from:* to write the first image in the NXImage.

You may also wish to accept input and output parameters from a controlling program. There are two methods at hand to accomplish this. The two, *customSaveView*: and *customOpenView:* ask you to create a view that can be inserted at the bottom of their appropriate panels. These methods accept a width to help you align controls. You don't need to worry about saving any input beyond an unlink and the next link. They're one shot only.

The next method allows a protocol for passing in and returning values. This method is not strongly encouraged, but here for the support of non-NeXTStep applications. It allows the programmer to ask for a variable by name. For example, you might send the message, *"setCustomParameter*: "compression" *withValue*: &c". This method is discouraged, however, since there is no version control on what parameters a converter should respond to. For example, one TIFF converter might respond to the above method while another expects "factor" for the string. Any values you do accept should be well documented in the header file, and your converter should deal with bogus variables gracefully.

There is also some built-in error handling facilities, alluded to earlier. You need to respond to *errorState* and *errorMessage*. Both return ints. The first returns an int describing the current error condition of the converters. This should be one of the **CONVERT_ERR_…** values. The second returns the error type. This should be one of the **ERROR_…** defines. Also, in the event that an error you cannot describe via the standard defines happens, you may set the error type to unknown and then respond to the *errorString* method, which returns a **NULL** terminated string describing what went wrong.

Another important method is *protocolVersion*. This method describes the current protocol version supported by the converter. At present, all should return "1.0". This value may change in the future. This is mainly to insure that future enhancements can be added while guarding against sending bogus messages.

The final two methods are for reporting informal information to the user. The first, *getFormatName*, returns a **NULL** terminated string describing the graphic format. This should, in general, be short. For example, tiff.bcvt returns "Tagged Image File Format (TIFF)". The second method, *copyrightNotice*, is a bit longer and presents a more detailed message. You might choose to use this to inform about shareware fees, copyrights, version, etc... For example, gif.bcvt returns, "GIF Converter\nby Alex Raftis\n\nCopyright (c) 1991 Cal Poly State University\nCopyright (c) 1989, 1990 University of Pennsylvania\nCopyright (c) 1988, 1989 by Patrick J. Naughton\n\nEmail bugs to alex@data.ACS.CalPoly.EDU".

Finally, in the event that you need to use the window server, you should

ask your sender whether or not you're allowed to use it. This is basically a preventative measure, since a program attempting to access a non-existent window server can cause an application to crash. If not allowed the window server, then you converter should return a **nil** value and an **ERROR_NEEDSWINDOWSERV** error code.

With all of these implemented, you should have a fully functional converter. If you'd like more information, check out the supplied converters and also see the accompanying references. These are done in the style of the NeXT Reference manuals for ease of use. They appear in RTF format for use within the Digital Librarian.

Useful Notes

At this point, before diving into programming a converter, there are a few things you should know. First of all, since many converters need to perform a common suite of conversions, there is a package of supplied utilities for just this purpose. This is available upon request from your **sender**. This package allows easy methods for getting pixels from a bitmap, converting to black and white, performing color quantization, and converting pixel types. For more information, see the accompanying references.

Also note, you may add functionality to all converters by modifying the base class; however, I ask that you not do this, since more than one person modifying this code has the possibility of creating mass confusion. I would prefer that if you see a feature that absolutely must be added, you contact me about adding it to the base class. This makes sure that everyone is using the same version with the same version of the control object at a certain protocol level.

NXBitmapImageRepControl Object

This is the object that most people will be using. It is what actually deals with linking the converters, handling their requests, and making sure the flow of control between them and your program is fluid. Using this object allows the support of many different types of bitmaps in your program at the cost of about fives lines of code above and beyond what might be required to support just TIFF's.

Abilities and Limitations

This object basically handles all the dirty work. It deals with dispatching user requests to converters, reporting errors, and returning bitmaps to your program. It can also work in a “dumb” mode, where you can pass requests through it directly to the converters. This is sometimes desirable when not working with the window server.

It's not perfect however. The main limitation is that only one converter may be linked at a time. What this means to you, the programmer, is that for each new bitmap type loaded, the object must do an unlink, so if your program does some other form of dynamic linking, you need to make sure that you explicitly unlink any loaded converter before doing your own linking. This will prevent one of your own modules from unexpectedly disappearing.

Reasoning

The object was written in the way it was for a couple of reasons. It is a control object because I didn't really see implementing a subclass of **NXBitmapImageRep** for each type of bitmap type I wanted to support, and besides, I wanted everything dynamic.

I also, though experimentation, found that the run time linking of each object on an as needed basis does not cause noticeable run time lag. The user will spend much more time waiting for bitmaps to be read than waiting for a converter to be linked.

I did, however, wish to allow the linking of all converters simultaneously, but due to vagueness in the run time library documentation, and some really obscure error codes from the run time linker, I've been unable to get this done. This may, in fact, change in the future, if I can finally figure out how to do it properly.

Finally, the program is almost completely dynamic to make it very versatile. Any portion of the code, from the control objects to the converters themselves, can be replaced without the required relinking at compile time seen with other objects.

Programming With It

As stated before, programming with this object is very simple, and requires only a few steps. You just need to make one method call to get an id for the converter. From that point on, all messages for loading and saving bitmaps should be sent to this **id**.

ControlLoader Object

This is the one object you must link at compile time. It's as simple as possible to ensure that it won't need to be needlessly updated, requiring re-compiles for people using the object. Basically, here's the one message you'll need to send to this object:

images = [[[ControlLoader *loadControl*: "Bitmap"] *alloc*] *init*];

This makes a new, temporary instance of control loader, that goes out onto disk and loads a file named “Bitmap.controls” from a library's Converter folder. It checks ~Library, /LocalLibrary, and /NextLibrary respectively. For a return value, it returns the **id** of the **NXBitmapImageRepControl** object. Note that it frees itself when it's done with this step, so you don't need to worry about keeping an **id** for it. If a fatal error occurs, images will have been set to **nil**.

In your application, when you need to use this object, you'll need to include “ControlLoader.h” and you will also need to add libconverter.a to your Makefile. This can be done from within InterfaceBuilder™ via your project window. Also, depending on whether or not you've installed the header files and libraries in /usr/lib and /usr/include respectively, you may need to specify full paths. For the library, use the -L*pathname* in the **$(CFLAGS)** line of your makefile.

Loading Images

When you'd like to load an image you can go about it in two ways. First, you can use the NeXTstep desktop environment, or you can ask for a file to be loaded directly. In the first instance, you need to get your application's open panel so that you can pass this to the appropriate method in the control object. Here's an example of how this is done:

```
…
id          openPanel = [OpenPanel new];
id          image;
char        **files;

[openPanel allowMultipleFiles: YES];
```

```
if (![images runOpenPanel: openPanel]) return self;

files = (char **)[openPanel filenames];
…
```

This runs the open panel and gets a list of files names you can load. All file names will have a valid extension representing some bitmap the user wishes to load. Note that you could also insert an accessory view into the panel before passing the panel along. The view will be incorporated into the control object's own view and put back when done.

Once you've done this, you then need to load the files from disk. This is also the only step you'd need to take when not using the NeXTstep environment. This can be simplified by not doing error checking, but we know that we always should do that, so here's a complete example. This code follows the above.

```
…
for (x = 0; files[x]; x++) {
        sprintf(buffer, "%s/%s", [openPanel directory], files[x]);
        if (image = [images openAndReturnImage: buffer]) {
                if ([images errorState] != CONVERT_ERR_NONE) {
                        NXRunAlertPanel(   "Alert",
                                           errors[[images
errorMessage]],
                                           "Continue", NULL, NULL);
                }
                …
                Here you can handle the bitmap, “image” however
                you'd like. At this point, I'd normally display it in a
                window.
                …

        } else {
                NXRunAlertPanel(   "Alert",
                                   "Unable to open file: %s\n",
                                   "Continue", NULL, NULL,
                                   files[x]);
        }
}
…
```

As you can see, the actual line that opens and gets the image is quite short, being only one line in the code example above. The error checking is a little more complicated, since you can get two types of errors, a warning or a fatal error. The first kind in the above example warns the user

that something went wrong, but attempts to open the bitmap anyway. The second part tells the user the bitmap could not be loaded, and does not open a new document.

That's all that's required to open a new bitmap.

Saving Images

Saving an image is similarly simple. Like the open method, you simply get your application's save panel and pass this on to the control object. Like the open panel, you can also insert a custom view. Here's an example of saving a bitmap.

```
…
savePanel = [SavePanel new];

if ([images runSavePanel: savePanel withFilename: buffer]) {
        if (![images saveImage: [currentDoc getImage]
                        toFile: [savePanel filename]]) {
                sprintf(buffer, "Unable to save file: %s",
                                                [savePanel filename]);
                NXRunAlertPanel(   "Save",
                                   buffer,
                                   "Continue",
                                   NULL, NULL, NULL);
        }
}
…
```

As you can see, you can save an image in just two lines of code, sans error checking, just like the open method. Simply cut out the first message to images if you are not running under the window manager.

Advanced Techniques

Of course, there may be times when the above, simple methods are not called for. For this reason, there are more advanced methods for dealing with the objects. In the hopes of a consistent interface, however, it's preferable that you try to use the above methods whenever possible.

Manually Linking and Unlinking

One of the first things you might need to do is manually link and unlink converters. There are basically two methods for dealing with this, *handleLink:* and *unlinkConverter*. The first message accepts a **NULL** terminated string that represents the file format desired. For example, you could pass in “gif” if you wished to link the GIF converter. If another

converter is already linked, it will be unlinked to make room for the new converter. The other method, *unlinkConverter* simply unlinks the current converter.

v Warning If you have dynamically linked another portion of code and call *unlinkConverter*, your code, and not the converter, will be unlinked.

Of course, if you're doing this, you may want to know what the legal types happen to be. You can request this information by sending the *getTypeList* message to the control object. This returns an array of NULL terminated strings with the last entry set to nil.

Supporting Sender Messages

If you'd like to send messages to converters directly, you need to be aware of what the converters might request of you. Mainly, they'll need to request an instance of ImageTools. The easiest way to satisfy their needs it to pass your instance of NXBitmapImageRepControl as the sender of the message, but if you wish to pass yourself, you need to implement the messages *getImageControl:*, *filename,* and *usesNeXTStep*. The first of these methods returns an instance of an ImageTools object. You may either dynamically link the one from the Libraries directory or supply one of your own. However, whatever you supply must respond to all the messages in ImageControl.h. The second message returns the current file name of the image. This information is occasionally needed by converters to put into a file header. The final message allows a converter to query whether or not it's allowed to use the window server to accomplish its needs. The best example of this is a EPS converter that uses the window server to load images.

Manual Loading

When combined with the linking functions of the **NXBitmapImageRepControl** object, manual loading is a fairly simple process. After doing what was stated above for linking the **NXBitmapImageRepControl** object into your code, you also need to do the following:

```
…
[bitmaps handleLink: type];
converter = [bitmaps getCurrentConverter];
fprintf(stderr, "Using Format: %s\n", [converter getFormatName]);
myStream = NXOpenFile(fileno(stdin), NX_READONLY);
image = [converter readFromStream: myStream from: bitmaps];
if ([converter errorState]) {
        fprintf(stderr, "Converter Error: %d\n", [converter errorMessage]);
```

```
        exit(1);
    }
    …
```

In the above example, bitmaps is the instance of **NXBitmapImageRepControl**. This example will read an image from stdin. It's used in a short program for copying files into the pasteboard from the command line.

Manual Saving

Saving is almost as easy. There's one more level of functionality that you can have with a converter, however. This lies in setting a converter's parameters, which is especially useful when saving an image while using the command line.

Basically, what you'd do is link the converter, and then set its custom parameters. For more information on this, read the documentation for converters. You will also need to examine a converter's header file in order to see to what custom parameters it can respond. Here's an example of setting these parameters and saving an image:

```
    …
    [images handleLink: "tiff"];
    curConvert = [images getCurrentConverter];
    [curConvert   setCustomParameter:
    TIFF_COMPRESS_METHOD
                  withValue:          &compress];
    [curConvert   setCustomParameter:     TIFF_COMPRESS_RATIO
                  withValue:          &factor];
    if ([images saveImage: curImage toFile: newName] && removeIt) {
        remove(argv[x]);
    }
    …
```

This example simplifies the previous by using the **NXBitmapImageRepControl** object, images, to save the file. This means you don't need to create an **NXStream** by yourself, and simplifies responses to the converters. You could have also called the converter's *write:toStream:from*: directly by sending a message to curConvert. The first parameter would have been an id to an **NXBitmapImageRep**, the second to an **NXStream** opened for writing, and the final a *sender*. For the sender, you'd probably want to send images, since then you wouldn't need to worry about the converter's needs directly.

Final Notes

Hopefully, you'll find this object easy to use. The idea was to simplify and make using many different graphics formats as easy to use as possible. If you'd like more information on this object, please see the accompanying document, NXBitmapImageRepControl.rtf.

Image Control Object

Abilities and Limitations

When writing converters, the ImageControl object can do a lot for the programmer. It greatly simplifies access into the NXBitmapImageRep when working on a pixel by pixel basis. It will also handle some basic conversions of the image format for you. These conversions involve, in general, algorithms complex enough that I felt it would be a waste for people to re-write them constantly. They also encompass the elements that I felt people would not want to write, preventing them from writing converters.

On the down side, there are certain limitations to the what ImageControl object can do for you. It cannot put pixels back into a bitmap. Also, the CMYK color conversions do not work at this time.

Like the other objects implemented in GraphicsWorkshop, this one will also be linked from disk, allowing for great flexibility in the source code.

Reasoning

While it may seem to you, the programmer, that certain features of this control object are missing, I felt that certain features were not needed for it. These are in two parts. The first is putting pixels. In general, when reading an image, you know exactly what the pixels will look like in the finished image. This means that you can write simple algorithms to put the final pixels into the image. This would, in general, be much faster than a generic “putpixel” could be.

The other class of missing routines involve easy to implement functions. The most prominent among these would be a “to gray scale” conversion. I left this out because the code to do gray scale conversion in about ten lines of code, using the color conversion and get pixel operators.

Another important aspect of this object is that it can be linked from disk like the other objects. This is important in the event you are not satisfied with my final product, which is possible, and perhaps even likely due to the simplicity of the *convertToPalette* method and the lack of CMYK color conversion. For this reason, you could redesign the parts you don't like, place the new Mach-O file in the correct place on disk and all programs launched henceforth will use your new version.

Useful Methods

When using the ImageControl object, there are two classes of routines and two methods you will likely use the most. The first class of routines

deal with getting pixels from a bitmap and the second with converting the color values in those pixels to other color types. The two methods you'll use are for creating one bit dithered images and converting twenty-four bit images into images with palettes.

The first step in using this object is to get a new instantiation of the object, passing it an **NXBitmapImageRep**. You'll usually, as a converter, do this by requesting it of your *sender* (usually the **NXBitmapImageRepControl** object). You do this by sending your *sender* the *getImageControl:* message. See the discussion on the **NXBitmapImageRepControl** object for more details. You may also message the control object's *new:* method. This is essentially what your sender will do.

Next, when you decide to get pixels from a bitmap, you have one of two choices. You may get them in a random access order, by specifying (x,y) pairs or you can get them sequentially. Unless you're doing some sort of special processing, you'll usually just want to get the pixels sequentially. Below is a discussion of this method, which can be easily applied to random access, if you choose.

You first need to declare a pointer to a function of type **Pixel *GetPixelNextFunc** (or **Pixel *GetPixelXYFunc(int, int)**). Note that regular C functions are used for optimization reasons over method calls, because you may need to call this function a million plus times. Your next step is to assign a pointer to this variable. You do this by sending the *getNextFunction* (or *getXYFunction*) message to the object.

Now that you've got the function pointer assigned a value, you can almost call it. In the case of random access, you can call it immediately, but the getNextFunction needs a *resetNext* message sent first. This sets up the object to return you the pixels, beginning at (0,0), the upper left-hand corner of the bitmap.

Converting between colors is as easy as getting the pixels in the first place. You will need to decide what you'd like to convert to, and then request a function that will do the job. Here you've got two choices: you can request a function pointer of type **ToGrayFunc** or **ToRGBFunc**. Both of these functions take a pointer to a **Pixel** and return a pointer to a **Pixel**. This allows for easy nesting of get pixel calls and color conversions. Here's an example of converting a color picture to gray scale.

```
…
id                  newImage;
id                  imageCon = [ImageControl new: imageIn];
unsigned char           *planes[5];
GetPixelNextFunc    nextPixel;
ToGrayFunc          toGray;
```

```
Pixel           *workPixel;
int             i, size;
double                  scale;
BOOL            alpha = [imageIn hasAlpha];

// First, create a new image to place the gray scale image.
newImage = [NXBitmapImageRep alloc];
[newImage   initData:           NULL
            pixelsWide:         [imageIn pixelsWide]
            pixelsHigh:         [imageInpixelsHigh]
            bitsPerSample:      8
            samplesPerPixel:    alpha ? 2 : 1
            hasAlpha:           alpha
            isPlanar:           YES
            colorSpace:         NX_OneIsWhiteColorSpace
            bytesPerRow:            0
            bitsPerPixel:       0];

// Get its data planes, so we can store our new image. Note that the
// newer image is planar to simplify dealing with alpha data.
[newImage getDataPlanes: planes];

// Get the next pixel function.
nextPixel = [imageCon getNextFunction];

// Get the toGray function
toGray = [imageCon getToGrayFunction];

// Precompute the number of pixels in the image.
size = [newImage pixelsWide] * [newImage pixelsHigh];

// Compute the scale factor. This is important because the returned
values
// are the same bits per sample as the original image.
scale = 256.0 / pow(2.0, (double)[imageIn bitsPerSample]);

// Reset the get next function so we can start at the beginning.
[imageCon resetNext];

// Loop for each pixel of the image.
for (i = 0; i < size; i++) {
        workPixel = toGray(nextPixel());
        planes[0][i] = workPixel->values[0] * scale;
        if (alpha) planes[1][i] = workPixel->values[1] * scale;
}
…
```

Before we leave these functions, you should know a bit more about just what are **Pixel**'s. Here's their typedef statement:

```
typedef struct {
        pixel   values[5];
        int     count;
} Pixel;
```

The array is an array of five pixel values. A pixel is basically an unsigned long. Count is the number of samples in the pixels. Therefore, a gray scale image would have a count of 1, while a CMYK pixel with alpha would have a count of 5. Values are stored in expected order. *I.e.*, RGBA is stored red, green, blue, and alpha.

That's all there is to getting pixels and converting colors. Now, you may need to do one of the most common color conversions. There are two methods to support this: *ditherImage* and *convertToPalette:andReturn:andPalettes:::*. The first transforms your image by applying a Floyd Steinberg Ditherizing Algorithm over the image. It returns to you a new **NXBitmapImageRep** with a one bit per sample image in it. The control object will still be valid for the original image, so you'll need to create a new image control object to access the one bit image.

The *convertToPalette:andReturn:andPalettes:::* method is similar in function, but more complex in parameters. It will return a new image, but it also returns three arrays of RGB values representing the palette found for the image. Under its current incarnation, this method applies a simple mapping algorithm that, while it produces decent images, may cause color banding. This is especially true with images that contain few colors in the resulting palette. If you're looking for a small project, this would be a good place to start by modifying the code to execute a Median Cut Algorithm. See the next section on how to modify this object.

Advanced Topics

In the course of using the object, you may decide that you've got a better way of doing things. If this is the case, feel free to change the code; however, there are a couple of guide lines you need to follow.

Re-writing the Object

When re-writing this object, you need to know that it has to use a flock of global variables stored inside the object implementation. This is due to the use of functions, but it means that inheriting a new kind of ImageControl object isn't very practical since you will not be able to access many important variables. Due to this, plan on modifying my code rather than

subclassing a newer version.

Next, when modifying the code, you can replace anything you'd like, add functionality, or whatever. However, you must keep all data types the same, as well as all currently implemented methods. You can only change the code, not the entry points. Also remember that if you add functionality, only your private converters should depend on it. I don't want to see a bunch of versions of converters, all of which depend on different versions of ImageControl objects. This only leads to frustration for the user.

Appendix A: Header Files

ControlLoader.h

NXBitmapImageRepControl.h

ImageControl.h

Converter.h

Appendix B: Source Code

ControlLoader.m

NXBitmapImageRepControl.m

ImageControl.m

Converter.m

Appendix C: Graphics Formats

GIF

Probably one of the most popular and yet one of the more limited graphics standards out, GIF, or Graphics Interchange Format, was introduced by CompuServe in order for its clients to be able to exchange graphics across many platforms. They'd also hoped to be able to implement simple animation through the GIF format for such applications as weather maps.

JPG

One of the newer standards on the scene is JPEG, standing for Joint Photographic Extension Group. This format is perhaps the best at compressing images; however, it does so at a “loss” of picture quality. This loss is dependant on the compression factor. JPEG was introduced as a standard for the compression of twenty-four bit and Grayscale images.

NeXT chose one of two standards for storing the JPEG files. Theirs, the TIFF encapsulation format, was the first to be widely seen publicly. The second is the stand-alone format being discussed here.

Some people worry about the “lossyness” of the JPEG format; however, it's important to note that you can control the amount of loss. At default levels, the JPEG format will lose much less information about the original picture than a color quantized GIF file. Also, when compressing “real world” images, JPEG shines, but the compression of generated images sometimes shows remnants. Still, considering the amount of disk savings and the fact that less is lost in the JPEG compression scheme than the GIF scheme, JPEG is quickly becoming a format of choice for 24 bit images.

Currently, this converter leaves a bit to be desired. There is no manner to adjust the compression factor of an image being saved, and for that matter, there is no manner to save the picture. I hope to have this cleared up as the JPEG libraries approach completion. For the time being, however, you may open .jpg files and save them back as JPEG's by encapsulating them as TIFF's.

PBM, PGM, PPM, and PNM

This collection of formats is less an exchange mechanism than a intermediate step. However, due the generic nature of the picture format, the PNM files may often be the best manner of transmission over ASCII only lines. The four formats are basically for three image types. The first, PBM, is for storing black and white bitmaps. The second, PGM, for storing gray scale images. The third for storing red, green, and blue color triplets.

The final for storing any of the previous as one file extension.

These formats were originally written to convert between graphics formats. For example, if you wished to convert a GIF to a PICT file, you might need to convert first to PPM to produce a twenty-four bit color image, followed by a conversion to PGM to get a gray scale image, followed by a conversion to PBM to get a one bit pictures and finally a conversion to PICT, which wants the one bit image. This method basically simplified the work of the image converter writers, freeing them from mundane tasks like pixel manipulation.

Over the years the PNM format (often referred to as PBM) has grown to support a wide variety of image formats and utilities. It's also become a bit of an exchange format due to its architecture-independent storage techniques and the ease with which it may be read and written. Unfortunately, this independence means no file compression or other techniques are employed, resulting in large images that require lots of disk space for storage.

These formats are included in large part to make conversion from non-specified formats easier. For example, since no PICT reader currently exists, you could convert to PBM and then use the PNM utility package, available from FTP, to convert to PICT.

TIFF

TIFF is probably the most versatile of all formats. Unfortunately, this also means it can be unreliable when people and programmers don't properly follow its specifications. This is most prevalent when coming from or going to the Macintosh environment. I've often had difficulty when working with TIFF's from this platform.

Some points of interest. The TIFF format is the NeXT platform's format of first choice, so you'll most likely be storing many of your images in it. To save space, note that JPEG within TIFF is supported, but be careful. Some programs, especially older ones, will not support reading JPEG compressed images, since they use older system calls to retrieve them. When saving as JPEG, the default compression factor is the recommended factor to preserve image quality, and yet get efficient use of space. You may raise this value for smaller, but “fuzzier” pictures.

Also note, should you wish to read these TIFF's on another platform, you should probably avoid using JPEG. Rather, choose PackBits™ or LZH. PackBits, developed by Apple Computer™, is best for one bit images, while LZH is better for color and grayscale images. However, don't expect nearly as small file sizes with either of these methods. You can also choose to save the images uncompressed, as may be necessary when

reading them on other platforms.

XBM

The final format on my list of supported formats is XBM. This is probably the most verbose and portable of any of the formats when traveling between UNIX environments. However, it's very limited. Mainly, the files generated are legal C code. This means that all bytes in the image are stored in the form 0xFF with at least a comma as a separator. This can produce massively large files. You will also lose all color data as XBM is a one bit format only.

Appendix D: References

Larkin, Don, Matt Morse, Jim Inscore, Sam Streeper, and Jackie Neider. *NeXTstep Reference Volume 1-2,* 2nd ed. Redwood City: NeXT Computer, Inc., 1990.

Larkin, Don, Matt Morse, Jim Inscore, Sam Streeper, and Jackie Neider. *NeXTstep Concepts,* 2nd ed. Redwood City: NeXT Computer, Inc., 1990.

CompuServe Incorporated, *GIF Graphics Interchange Format*, 1st ed. CompuServe Incorporated

Mish, Frederick C., Marchael Hawley, *The* NeXT *Digital Edition of Webster's Ninth New Collegiate Dictionary and Webster's Collegiate Thesaurus*, NeXT Computer and Merriam-Webster Incorporated, 1898, 1988

Index

Supplemental

Now that you've had a chance to see all the code that is part of my senior project, you might be interested in some of the other code and documentation. After this page follows the supplemental references, which are followed by the code for GraphicsWorkshop itself.