Interactive Image Display for the X Window System

Written by John Bradley
(bradley@cis.upenn.edu)

Version 2.20

This work was partially funded by the following: Navy Grant N0014-88-K-0630, AFOSR Grants 88-0244, AFOSR 88-0296; Army/DAAL 03-89-C-0031PRI; NSF Grants CISE/CDA 88-22719, IRI 89-06770; the Du Pont Corporation, and the IBM Corporation.

Section 0: Release Notes

What's New in Version 2.20?

The following is a short synopsis of the major features that have been added to *xv* since version 2.10. See the `CHANGELOG` file in the *xv* source directory for complete details, and up-to-the-minute information.

- **Grab** command added to the *xv controls* window. This command lets you grab any window, or rectangular portion of the screen, into *xv*. The grabbed image can be manipulated, saved, printed, etc. exactly like any other image.

  This feature may very well make *xv* your "one-stop shopping" replacement for *xwd, xwdtopnm, xgrabsc, xwdps,* and *xmag*. Probably some others, as well. Also, if you use *xv*'s extensive color manipulation capabilities on a grabbed image, you'll have a fine replacement for *pikapix*, a color-chooser program for X10 that I've never seen ported to X11.

- Zoom and UnZoom controls. You can now zoom in on a portion of the image without going through the hassle of cropping and expanding.

- Compressed PostScript. You can now optionally choose to generate RLE-compressed 8-bit per pixel PostScript files (color or greyscale). If you're generating color PostScript, you can expect a 4:1 compression ratio over uncompressed PostScript. If you're stuck for space, or have a slow 9600 baud connection to the PostScript printer, then this might be the thing for you.

  Also, you can now optionally choose whether or not to include the EPSF b/w preview in your PostScript files. This preview may or may not be used by various desktop publishing programs.

- *xv* can now read Utah Raster Toolkit RLE files.

- Histogram Equalization. You can now automatically perform histogram equalization on any rectangular region of the image. This generally has the effect of improving the contrast of the image. It works best on greyscale images.

- You can now **Load** files from a Unix pipe, from inside the *xv load* window. *xv* will run the specified command(s), and try to load the stdout produced. For example, you could use this as a way to load images from a scanner or a video digitizer, assuming you had a program to that would talk to the actual hardware, and generate an image in one of the formats that *xv* can handle.

- Support for 4"x5" and 35mm PostScript film recorders.

- New versions of both the TIFF and JPEG support libraries have been merged in.

and, of course, there have been plenty of bug fixes and minor enhancements.

Section 1: Overview

XV is an *interactive* image manipulation program for the X Window System. It can operate on images in the GIF [1], JPEG, TIFF, PBM, PGM, PPM, X11 bitmap, Sun Rasterfile, RLE[2], and PM [3] formats on 1-, 2-, 4-, 6-, 8-, 16-, 24-, and 32-bit X displays.

XV lets you do a large number of things (many of them actually *useful*), including, but not limited to, the following:

- display an image in a window on the screen
- display an image on the root window, in a variety of styles
- grab any rectangular portion of the screen and turn it into an image
- arbitrarily stretch or compress the image
- rotate the image in 90° steps
- flip the image around the horizontal or vertical axes
- crop a rectangular portion of the image
- magnify any portion of the image by any amount, up to the size of the screen
- determine pixel values and x,y coordinates in the image
- adjust image brightness and contrast with a *gamma* correction function
- apply different *gamma* functions to the Red, Green, and Blue color components, to correct for non-linear color response
- adjust global image saturation
- perform global hue remapping
- perform histogram equalization
- edit an image's colormap
- reduce the number of colors in an image
- dither in color and b&w
- smooth an image
- crop off solid borders automatically
- convert image formats
- generate Encapsulated PostScript [4]

Unfortunately the *Automatic Checkbook Balancing Module* still isn't completely debugged, and is not included in this distribution.[5]

[1] GIF is a trademark of CompuServe Incorporated, an H&R Block Company.

[2] Of the Utah Raster Toolkit variety.

[3] An internal format developed and used in the GRASP Lab.

[4] A trademark of Adobe Systems Incorporated.

[5] It's a joke.

Section 2: Starting XV

Note: unless explicitly stated otherwise, the term *click* means "click with the *Left* mouse button."

Start the program up by typing '`xv`'. After a short delay, a window will appear with the default image (the *xv* logo, credits and revision date) displayed in it. If you change the size of the window (using whatever method your window manager provides), the image will be automatically *stretched* to fit the window.

Section 2.1: Displaying Pixel Values

Clicking (and optionally dragging) the *Left* mouse button inside this window will display pixel information in the following format:

```
196, 137 = 191,121,209 (287 42 81 HSV)
```

The first pair of numbers (`196,137`) are the *x* and *y* positions of the cursor, in *image coordinates*. These numbers remain the same regardless of any image resizing, or cropping. For example, if you click on the eye of the fish on the right side of the default image, you'll get (approximately) `251,129` regardless of the size of the displayed image. This allows you to zoom in for precise measurements.

The first triplet of numbers (`191,121,209`) are the RGB values of the selected pixel. The components will have integer values in the range 0-255. The values displayed are prior to any HSV/RGB modification, but *after* any colormap changes. See "Section 5: The Color Editor" for details.

The second triplet of numbers (`287 42 81`) are the HSV values of the selected pixel. The first component will have integer values in the range 0-359, and the second and third components will have integer values in the range 0-100. The values displayed are prior to any HSV/RGB modification, but *after* any colormap changes. See "Section 5: The Color Editor" for details. Also, see "Appendix D: RGB and HSV Colorspaces" for more information about what these numbers mean.

Note: If you actually want to measure some pixels, it will probably help to crop to a small region of your image, and expand that region so that you can see the individual pixels.

This string is automatically copied to your X server's *cut buffer* whenever you measure pixel values. This lets you easily feed this information to another program, useful if you're doing manual feature extraction, or something. Try it: measure a pixel's value, and then go click your *Middle* [6] mouse button in an *xterm* window.

Section 2.2: Cropping

Bring up the *xv controls* window by typing the '**?**' key or clicking the *Right* mouse button inside the image window.

Clicking and dragging the *Middle* button of the mouse inside the image window will allow you to draw a *cropping rectangle* on the image. If you're unhappy with the one you've drawn, simply click the *Middle* button and draw another. If you'd like the rectangle to go away altogether, click the *Middle* button and release it without moving the mouse.

You can determine how large the cropping rectangle is (in image coordinates) by bringing up the *xv info* window. Do this by clicking the **Info** button in the *xv controls* window or by typing the '**i**' key into any open *xv* window.

The *xv info* window will display, among other things, the current size and position of the cropping rectangle in terms of image coordinates. For example, if it says:

```
114x77 rectangle starting at 119,58
```

it means that the current cropping rectangle is 114 image pixels wide, 77 image pixels high, and that its top-left corner is located 119 image pixels in from the left edge of the image, and 58 image pixels in from the top edge. These values will be updated as you drag the cropping rectangle around.

If you want to set the size or position of the cropping rectangle precisely, you can use the arrow keys on your keyboard. First, make the *xv info* window visible as described above (if it's not already visible). Second, use the mouse to draw a rough approximation of the cropping rectangle that you want. You can now use the arrow keys to move the cropping rectangle around the image. Once you've gotten the top and left sides of the cropping rectangle precisely where you want them, you can move the bottom-right corner of the cropping rectangle (only) by holding the **<shift>** key down while using the arrow keys. Pressing the up arrow will make the rectangle shorter, and pressing the down arrow will make the rectangle taller.

Once you have a cropping rectangle that you can live with, you can proceed with the actual cropping operation. Click the **Crop** button in the *xv controls* window, or type the '**c**' key in any open *xv* window. The image window will shrink to show only portions of the image that were inside the cropping rectangle.

Note: if you are running a window manager such as *mwm*, which decorates windows with a title bar, resizing regions, and such, it is quite possible that the aspect ratio of the cropped image will get screwed up. This is because certain window managers enforce a minimum window size. If you try to crop to a rectangle that is too small, the window manager will create the smallest window it can, and the image will be stretched to fit this window. If this happens, you can press the **Aspect** button in the *xv controls* window, or type the '**a**' key in any open *xv* window. This will expand the

[6] If your workstation only has a two-button mouse, you can *probably* emulate a middle button by pressing both buttons simultaneously.

image so that it has the correct aspect ratio again.

You can crop a cropped image by repeating the same steps (drawing a new cropping rectangle and issuing the **Crop** command), ad infinitum.

You can return to the original, uncropped image by using the **UnCrop** command. Simply click the **UnCrop** button or type the '**u**' key in any open *xv* window. Note that using the **UnCrop** command will turn off image smoothing (the **Smooth** command), due to the potentially long time it can take to generate a large, smoothed image.

Note that if you try to make the cropping rectangle too small in either width or height (under 5 *screen* pixels), it'll just turn itself off. If you want to crop a very small portion of an image, you'll probably have to do it in two passes. First, crop to a small (but large enough to still be enabled) rectangle, expand that image, then crop again.

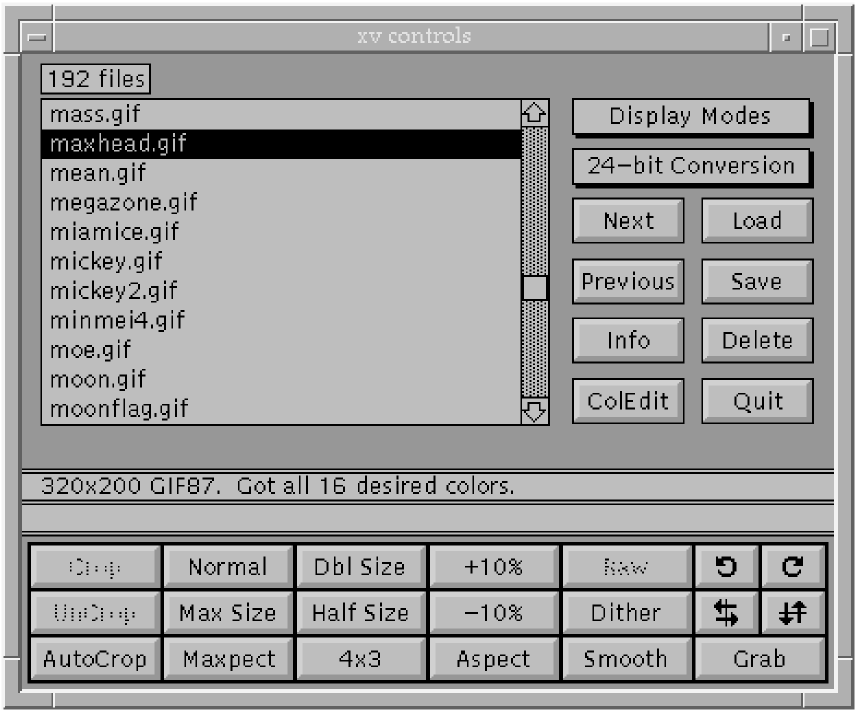
Section 2.3: Zooming

You can zoom in by a factor of two (or four, or eight, etc.) on any rectangular region of the image by holding down the **<ctrl>** key on your keyboard and clicking the *Left* mouse button in the image window. A rectangle will flash, centered around the cursor position, and the region inside that rectangle will be doubled in size. The image window should remain the same size. You can repeat this operation to zoom in by a factor of four, or eight, or whatever, as many times as you wish.

You can zoom out by a factor of two (if possible) by holding down the **<ctrl>** key and clicking the *Right* mouse button inside the image window. You can not zoom out beyond the point where the entire image fills the window.

Section 3: The Control Window

The *xv controls* window is the central point of control for the program, hence the name. It provides controls to resize the current image, flip and rotate it, load and save different files, and bring up the other *xv* windows. It can be brought up by clicking the *Right* mouse button in the image window, or by typing the '**?**' key inside any open *xv* window. Doing either of these things while the *xv controls* window is visible will hide it.



All of the following commands may be executed by either clicking the appropriate command button, or typing the keyboard equivalent (where given) into any open *xv* window.

Section 3.1: Resizing Commands

Note that none of the 'resizing' commands modify the image in any way. They only affect how the image is *displayed*. The image remains at its original size. This allows you to arbitrarily stretch and compact the image without compounding error caused by earlier resizing. In each case, the displayed image is recomputed from the original internal image.

Command	Key	Description
Normal	'**n**'	Attempts to return the image to its normal size, where one image pixel maps to one screen pixel. For example, if the image (or the current cropped portion of the image) has a size of 320x200, this command will attempt to make the image window 320 screen pixels wide by 200 screen pixels high. This command may fail in two cases. If you're running a window

manager (such as *mwm*) that enforces a minimum window size, and the 'normal' size is too small, the image may get distorted. See the note in "Section 2.2: Cropping" for more information.

Also, if the image is larger than the size of your screen, it will be 'halved' until it fits on the screen. For example, if you try to display a 1400x900 image on a 1280x1024 screen, the **Normal** command will display a 700x450 image.

Max Size '**m**' This command will make the displayed image the same size as the screen. If you are running a window manager that puts up a titlebar, you'll find that the titlebar is now off the top of the screen. To get the titlebar back, simply shrink the image to anything smaller than the size of the screen. The window will be moved so that the titlebar is once again visible.

Maxpect '**M**' Makes the image as large as possible, while preserving the aspect ratio. This avoids the generally unwanted image distortion that **Max Size** is capable of generating. For example, if you have a 320x200 image, and an 1280x1024 screen, doing the **Maxpect** command will result in an image that is 1280x800. **Max Size**, on the other hand, would've generated an image of size 1280x1024, which would be appear 'stretched' vertically.

DblSize '**>**' Doubles the current size of the image, with the constraint that neither axis is allowed to be larger than the screen. For example, given a 320x200 image and a 1280x1024 screen, the image can be doubled once (to 640x400), a second time (to 1280x800), but a third time would make the image 1280x1024. You'll note that on the third time, the width didn't change at all, since it was already at its maximum value. Also note that the height wasn't allowed to double (from 800 to 1600), but was truncated at its maximum value (1024).

HalfSize '**<**' Halves the current size of the image, with the constraint that neither axis is allowed to have a size less than 1 pixel. Also, you may run into 'minimum size' problems with your window manager. See the note in "Section 2.2: Cropping" for more information.

Note that the window size is maintained as a pair of integers. As a result you may see some integer round-off problems. For example, if you halve a 265x185 image, you'll get a 132x92 image, which is just fine. However, if you **Dbl Size** this image, you'll get a 264x184 image, not the 265x185 image you might have expected.

+10% '**.**' Increases the current size of the image by 10%, subject to the constraint that the image cannot be made larger than the screen size (in either axis). For example, issuing this command on a 320x200 image will result in a 352x220 image.

-10% '**,**' Decreases the current size of the image by 10%. Neither axis of the image is allowed to shrink below 1 pixel. Also, you run the risk of running into 'minimum window size' problems with your window manager.

It should be noted that the **+10%** and **-10%** commands have no concept of an 'original size'. They simply increase or decrease the current image size by 10%. As a result, they *do not undo each other.* For example, take a 320x200 image. Do a **+10%** and the image will be 352x220. If you issue the **-10%** command now, the image will be made (352 - 35.2)x(220 - 22), or 316x198.

4x3	'**4**'	Attempts to resize the image so that the ratio of width to height is equal to 4 to 3. (e.g., 320x240, 400x300, etc.) This is useful because many images were meant to fill the screen on whatever system they were generated, and nearly all video tubes have an aspect ratio of 4:3. This command will stretch the image so that things will *probably* look right on your X display (nearly all of which, thankfully, have square pixels). This command is particularly useful for images which have really bizarre sizes (such as the 600x200 images presumably meant for CGA, and the 640x350 16-color EGA images).
Aspect	'**a**'	Applies the 'default aspect ratio' to the image. This is done automatically when the image is first loaded. Normally, the default aspect ratio is '1:1', but certain GIF files may have an aspect ratio encoded in them. You can also set the default aspect ratio via a command-line argument or an X resource. See 'Section 9: Modifying XV Behavior' for more info. The idea behind this command is that you'd stretch the image manually (via your window manager) to roughly the size you'd like, and then use the **Aspect** command to fix up the proportions. Normally **Aspect** expands one axis of the image to correct the aspect ratio. If this would result in an image that is larger than the screen, the **Aspect** command will instead shrink one of the axes to correct the aspect ratio.

Section 3.2: Rotate/Flip Commands

Command	Key	Description
Turn CW	'**t**'	Rotates the image 90° clockwise.
Turn CCW	'**T**'	Rotates the image 90° counter-clockwise.
Flip H	'**h**'	Flips the image horizontally (around the vertical center-line of the image).
Flip V	'**v**'	Flips the image vertically (around the horizontal center-line of the image).

Section 3.3: Smoothing Commands

Command	Key	Description
Raw	'**r**'	Returns the displayed image to its 'raw' state (where each pixel in the displayed image is as close as possible to the corresponding pixel in the internal image). In short, it turns off any dithering or smoothing.

When dithering or smoothing haven't been done, this command is disabled.

Dither	'**d**'	Regenerates the displayed image by dithering with the available colors in an attempt to approximate the original image. This is only relevant if the color allocation code failed to get all the colors it wanted. If it did get all the desired colors, the **Dither** command will just generate the same display image as the **Raw** command. On the other hand, if you didn't get all the desired colors, the **Dither** command will try to approximate the missing colors by dithering with the colors that were obtained. If you're running *xv* on a 1-bit display the **Dither** command will be disabled, as the image will always be dithered for display.
Smooth	'**s**'	Smooths out distortion caused by integer round-off when an image is expanded or shrunk. This is generally a desirable effect, however it is fairly time-consuming on large images on most current workstations. As such, by default, it is not done automatically. See "Section 9: Modifying XV Behavior" for more details.

Section 3.4: Cropping Commands

Command	Key	Description
Crop	'**c**'	Crops the image to the current cropping rectangle. This command is only available when a cropping rectangle has been drawn on the image. See "Section 2.2: Cropping" for further information.
UnCrop	'**u**'	Returns the image to its normal, uncropped state. This command is only available after the image has been cropped. See "Section 2.2: Cropping" for further information.
AutoCrop	'**A**'	Crops off any constant borders that exist in the image. It will crop to the smallest rectangle that encloses the 'interesting' section of the image. It may not always appear to work because of minor invisible color changes in the image. As such, it works best on computer-generated images, and not as well on scanned images.

Section 3.5: The Display Modes Menu

In addition to displaying an image in a window, *xv* can also display images on the root (background) window of your X display. There are a variety of ways that *xv* can display an image on the root window. The **Display Modes** popup menu lets you select where (and how) *xv* will display the image.

Click on the **Display Modes** button in the *xv controls* window, and hold the mouse button down. This will cause the **Display Modes** menu to pop up. The current display mode will be shown with a checkmark next to it. To select a new mode, drag the mouse down to the desired mode, and release the mouse button.

It is not possible for *xv* to receive button presses or keyboard presses in the root window. As such, there are several functions that cannot be used while in a 'root' mode, such as pixel tracking and image cropping. If you want to do such things, you'll have to temporarily return to 'window' mode, and return to 'root' mode when you're finished. Also, when you are in a 'root' mode, you will not be able to get rid of the *xv controls* window. At best you can iconify it (using your window manager). (The reason for this is that if you ever got rid of it there'd be no way to get it back.)

Mode	Description
Window	Displays the image in a window. If you were previously in a 'root' mode, the root window will also be cleared.
Root: Tiled	The image is displayed in the root window. One image is displayed aligned with the top-left corner of the screen. The image is then duplicated towards the bottom and right edges of the screen, as many times as necessary to fill the screen.
Root: Integer Tiled	Similar to **Root: Tiled**, except that the image is first shrunk so that its width and height are integer divisors of the screen's width and height. This keeps the images along the bottom and right edges of the screen from being 'chopped-off'. Note: using any of the 'resizing' commands (such as **Normal**, **Dbl Size**, etc.) will lose the 'integer'-ness of the image.
Root: Mirrored	Tiles the original image with versions that have been horizontally flipped, vertically flipped, and both horizontally and vertically flipped. This gets rid of the sharp dividing lines where tiled images meet. The effect is quite interesting.[7]
Root: Integer Mirrored	Like **Root: Mirrored**, but also does the integer-ization described under the **Root: Integer Tiled** entry.
Root: Center Tiled	Like **Root: Tiled**, but it positions the images so that one of them is centered on the screen, and the rest are tiled off in all directions. Visually pleasing without the image size distortion associated with **Root: Integer Tiled**.
Root: Centered	Displays a single image centered in the root window, surrounded by black.
Root: Centered, Warp	Displays a single image centered in the root window, surrounded by a black and white 'warp' pattern, which produces some mildly visually pleasing Moire effects.
Root: Centered, Brick	Displays a single image centered in the root window, surrounded by a black and white 'brick' pattern.

Note: The three 'centered' modes (**Root: Centered**, **Root: Centered, Warp**, and **Root: Centered, Brick**, but *not* **Root: Center Tiled**) require the creation of a Pixmap the size of the screen. This can be a fairly large request for resources, and will fail on a color X terminal with insufficient memory. They can also require the transmission of considerably more data than the other 'root' modes. If you're on a brain-damaged X terminal hanging off a slow network, you should probably go somewhere else. Barring that, you should certainly avoid the 'centered' modes.

[7] Exercise #1: Display the 'default' *xv* image (the one with the logo, and the fish). Draw a (roughly) 86x70 cropping rectangle (roughly) centered around the fish on the right. (This is a fine opportunity to acquaint yourself with the method of cropping rectangle specification via the arrow keys.) Crop the image, and select the **Root: Mirrored** display mode. Voila! The Official John Bradley Desktop Pattern. Ask for it by name!

Also note: If you quit *xv* while displaying an image on the root window, the image will remain in the root window, and the colors used by the image will remain allocated. This is generally regarded as correct behavior. If you decide you want to get rid of the root image to free up resources, or simply because you're sick of seeing it, the quickest route is to use run '`xv -clear`', which will clear the root window, release any allocated colors, and exit. Alternately, *xsetroot* and any other X program that puts things in the root window should be able to do the trick as well.

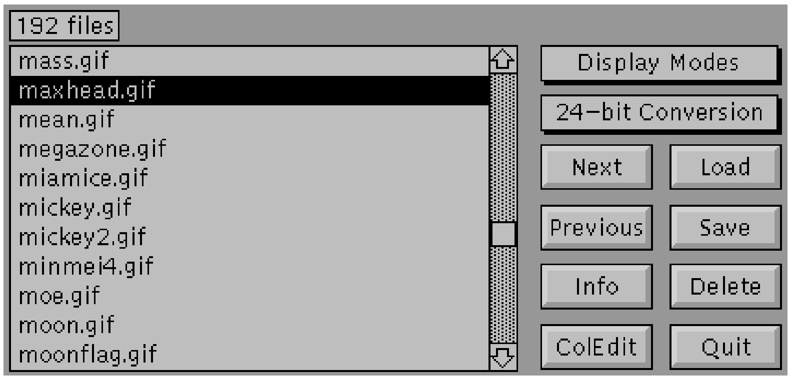
Section 3.6: The 24-bit Conversion Menu

xv currently can only operate on 8-bit images. Whenever you load a 24-bit image (such as JPEG, TIFF, PPM, etc.) it is immediately converted to an 8-bit colormapped image using one of three algorithms.

Algorithm	Description
Fast	Converts 24-bit images to 8-bit images by dithering with a fixed 6x6x6 RGB colormap. It is the quickest of the three algorithms, but also generally produces the worst images. It can also be selected via the '-quick24' command-line option or X resource.
Slow	The default algorithm. Takes about twice as long as the **Fast** algorithm. Uses the median-cut algorithm to pick a set of 256 colors, and then dithers with these colors. It can be selected via the '-slow24' command-line option or X resource.
Best	By far and away the slowest of the algorithms. Can take up to ten times as long as the **Slow** algorithm. Uses a cleverer version of the median-cut algorithm to pick a better set of 256 colors than the slow algorithm. Does *not* dither. This might look best if you're going to be expanding the image by very much, as the dithering in the other two algorithms becomes very noticeable. You can also select this option via the '-best24' command-line option or X resource.

Section 3.7: Working With Multiple Files

xv provides a set of controls that let you conveniently operate on a list of images. To use the following commands, you'll have to start up *xv* with a list of filenames. For example, you could type '`xv *.gif`' (assuming, of course, that you have a bunch of files that end with the suffix '.gif' in the current directory).



The filenames are listed in a scrollable window. The *current selection* is shown in reverse video. If there are more names than will fit in the window, the scrollbar will be enabled.

Section 3.7.1: Operating a List Window

The scrollbar operates as follows:

- clicking in the top or bottom arrow of the scrollbar scrolls the list by one line in the appropriate direction. It will continue to scroll the list as long as you hold the mouse down.

- The *thumb* (the small white rectangle in the middle of the scrollbar) shows roughly *where* in the list you are. You can change your position in the list by clicking and dragging the thumb to another position in the scrollbar. The list will scroll around as you move the thumb.

- You can scroll the list up or down a page at a time by clicking in the grey region between the thumb and the top or bottom arrows.

If you click on a name in the list, that name will become highlighted. You can drag the highlight bar up and down, and the list will scroll appropriately.

It is also possible to control the list window from the keyboard. In all cases, you must make sure that the window *sees* the keypress. Generally, this means you have to have the cursor inside the window, though your window manager may also require you to click inside the window first.

- The *up* and *down* arrow keys move the highlight bar up and down. If the bar is at the top or bottom of the window, the list will scroll one line.

- The *page up* and *page down* keys scroll the list up or down a page at a time.

- Pressing the *home* key will jump to the beginning of the list. Pressing the *end* key will jump to the bottom of the list.

Section 3.7.2: The File Commands

You can directly view any image in the list by double-clicking on its filename. If *xv* is unable to load the file (for any of a variety of reasons), it'll display an error message and put up the default image, the *xv* logo.

Command	Key	Description
Next	**<space>**	Attempts to load the next file in the list. If it is unable to load the next file, it will continue down the list until it successfully loads a file. If it gets to the bottom of the list without successfully loading a file, it will put up the default image.
Previous	**<backspace>**	Attempts to load the previous file in the list. If it is unable to load the previous file, it will continue up the list until it successfully loads a file. If it gets to the top of the list without successfully loading a file, it will put up the default image.
Delete	**<ctrl-D>**	This command lets delete the *currently selected* file from the list (and optionally delete the associated disk file). Note that the *currently selected* file is the one with the highlight bar on it. While this is generally the same as the currently displayed image, it doesn't have to be.

The **Delete** command will pop-up a window asking you what you want to delete. Your choices are:

- **List Entry**, which will remove the highlighted name from the list. (Keyboard equivalent: the *enter* key)
- **Disk File**, which will remove the highlighted name from the list and *also delete the associated disk file*. This removes unwanted images, just like manually typing '`rm <filename>`' in another window. (Keyboard equivalent: <ctrl-D>)
- **Cancel**, which lets you get out of the **Delete** command without actually deleting anything. (Keyboard equivalent: the *esc* key)

Section 3.7.3: Image Reloading

It is occasionally desirable to reload an image file because the contents of the file have changed. For example, you could be downloading a file, and you might want to keep reloading the file to check on the progress of the download. Or perhaps you have a program that generates images, and you'd like to view these images without any manual intervention.

XV provides a way to reload an image via an external signal. If you send the *xv* process a `SIGQUIT` signal ('`kill -QUIT <pid>`', or '`kill -3 <pid>`' on most systems), the program will reload the currently selected file. (The one that is currently highlighted in the *xv controls* window filename list.) This behavior is exactly the same as hitting '<return>' in the *xv controls* window. If *xv* is currently in a state where hitting '<return>' in the controls window won't load an image (ie, some pop-up dialog box is grabbing all such events), then sending this signal won't work either.

An idea: You could write a 'clock' program that, once a minute, generates a really *spiffy* looking picture of the current time (with color gradations, 3d extruded numbers, whatever), then sends *xv* the signal to reload the generated image. If anyone ever does this, I'd like to hear about it.

Note: This will *not* work if the current file was read from <stdin>.

Section 3.8: The Grab Command

The **Grab** command works as follows: click on the **Grab** button in the *xv controls* window, or type a **<ctrl-G>** in any active *xv* window (except for the JPEG, PostScript, and TIFF "save" dialog boxes). The terminal will beep once, and the cursor will change to a cross. The screen will remain frozen until you complete the **Grab** command.

You can **Grab** an arbitrary region of the screen by clicking the *Left* mouse button and dragging a rectangle in exactly the same way you draw a cropping rectangle. When you let go of the mouse button, the contents of this rectangle will be read from the screen and loaded into *xv*.

Alternately, you can grab the entire contents of a window (including its frame) by clicking the *Middle* mouse button in a window. If you click the *Middle* button somewhere on the root window, the entire screen will be loaded into *xv.*

Or, alternately, you can simply abort the **Grab** command by clicking the *Right* mouse button anywhere on the screen.

You can use the **Grab** command for a wide variety of purposes. For example, you can use it to print contents of any window (or the whole screen) by grabbing the appropriate image, and then saving as a PostScript file.

You can use the **Grab** command, in conjunction with *xv's* Zoom and UnZoom commands, as an effective replacement for the *xmag* program.

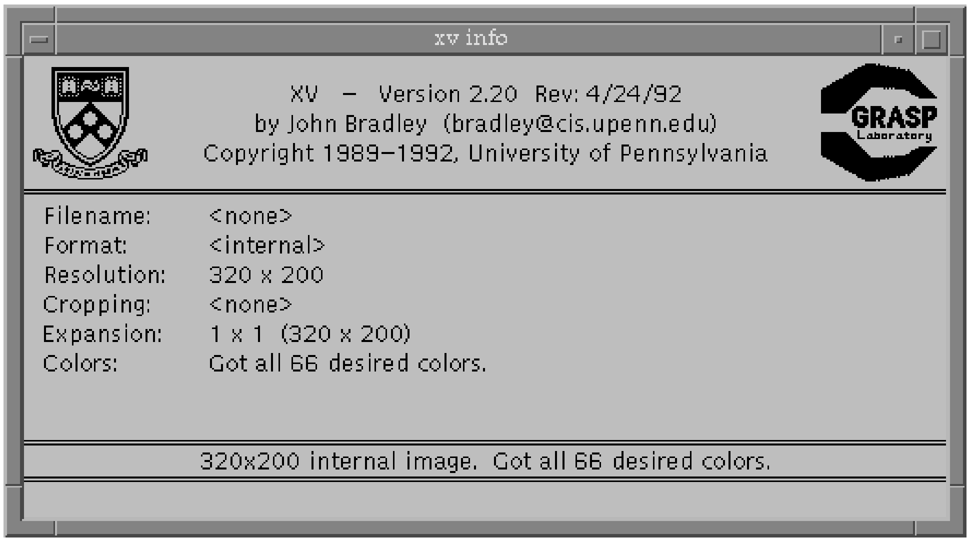
You can also use the **Grab** command to pick 'just the right colors' for any application. Simply start the application in question, **Grab** the window into *xv*, and use the colormap editor to twiddle the colors around to your heart's content.

Note: the **Grab** command does not work on Macintoshes running MacX in a 'rootless' mode, which isn't too surprising, if you think about it...

Section 3.9: Other Commands

Command	Key	Description
Info	**'i'**	Opens and closes the *xv info* window. See "Section 4: The Info Window" for more details.
ColEdit	**'e'**	Opens and closes the *xv color editor* window. See "Section 5: The Color Editor" for more details.
Load	**<ctrl-L>**	Opens the *xv load* window. See "Section 6: The Load Window" for more details.
Save	**<ctrl-S>**	Opens the *xv save* window. See "Section 7: The Save Window" for more details.
Quit	**'q'**	Quits out of the program.

Section 4: The Info Window



Section 4.1: Overview

xv provides a window to display information about the current image, color allocation, expansion, cropping, and any error messages. This window can be opened by issuing the **Info** command. (Click on the **Info** button in the *xv controls* window, or type '**i**' in any open *xv* window.) You can close the window by using the **Info** command while the window is open. You can also close the window by clicking anywhere inside it.

The top portion of the window displays the program name, revision date, and patchlevel. It also shows the University of Pennsylvania shield, the GRASP Lab logo, the copyright notice, and of course, the author's name.

Section 4.2: The Fields

The "Filename" field displays the name of the currently loaded file. The name is displayed without any leading pathname. If there *is* no currently loaded image (you're looking at the default image) this field will display "<none>".

The "Format" field displays information describing what image format the file is stored in, and how large the file is (in bytes).

The "Resolution" field shows the width and height (in image pixels) of the loaded image. Note that this does not necessarily have anything to do with the size of the image currently displayed on your screen. These numbers do not change as you modify the display image.

The "Cropping" field displays the current state of any cropping activity. If you are looking at the entire (uncropped) image, and there is no cropping rectangle drawn, this field will show "<none>". If you draw a cropping rectangle, or if you are viewing cropped portion of image, this field will display something like "247x128 rectangle starting at 132,421". See "Section 2.2: Cropping" for more details.

The “Expansion” field gives you information about how the image is displayed. It will display something like “1.58 x 1.37 (505 x 273)”. This tells you that the current displayed image is 505 pixels wide and 273 pixels high, and that it is 1.58 times wider and 1.37 times higher than the internal image (which, in this case, had a size of 320x200).

The “Colors” field gives you detailed information on how well (or poorly) color allocation went. If everything went reasonably well it will display something like:

```
Got all 67 desired colors.  (66 unique)
```

This means that 67 entries in the image’s colormap were used in the image, but that only 66 of these colors were *different*, as far as the X server is concerned.

See “Appendix E: Color Allocation” for a complete discussion of how colors are allocated, and what the “Colors” field can tell you.

Note that the fields are filled in as information becomes available. As such, they can be used as a rough ‘progress indicator’ when loading images. When you begin loading, all the fields are cleared. Once the image has been successfully loaded, the top three fields (Filename, Format, Resolution) are filled in. Once the colors have been allocated, and the display image generated, the bottom three fields are shown (Cropping, Expansion, and Colors).

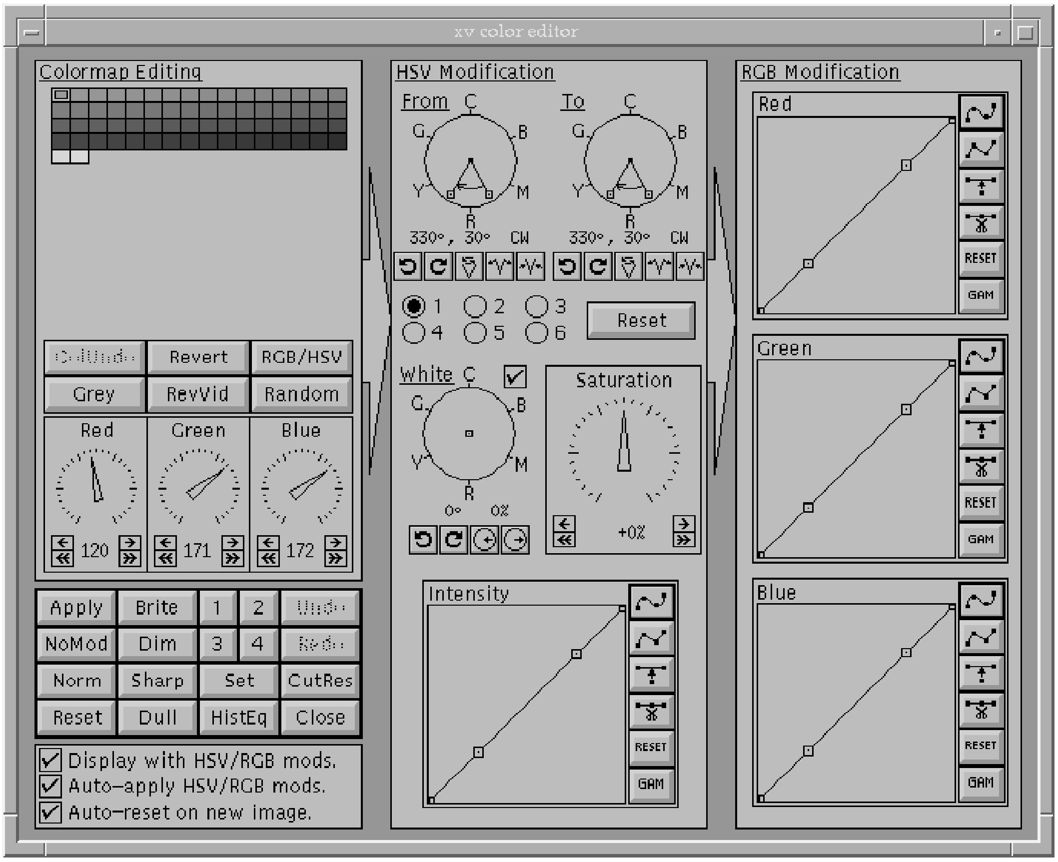
Section 4.3: Status Lines

The bottom two lines in the info window display various error messages, warnings, and status information. These two lines are also duplicated in the *xv controls* window.

The upper line is the more commonly used. It normally displays a one-line summary of the current image and color allocation success. If an error occurs, it will be displayed on this line as well.

The lower line is used to display warning messages.

Section 5: The Color Editor



Section 5.1: Overview

The *xv color editor* provides a powerful system for manipulating color images. Since there are many different reasons why a person would want to modify an image's colors, and many different types of images that may need modification, there is no *one* color manipulation tool that would be 'best' for all purposes. Because of this problem, *xv* gives the user *three* different color tools, all of which can be used simultaneously.

- **Colormap Editing**: This tool lets you arbitrarily modify individual colormap entries. Useful for modifying the color of captions or other things that have been added to images. Also works well on images that have a small number of colors, such as images generated by 'drawing' or CAD programs. It's also an easy way to spiff up boring 1-bit black and white images.

- **HSV Modification**: This tool lets you alter the image globally in the HSV colorspace. (See "Appendix D: RGB and HSV Colorspaces" for more info.) Here are examples of the sort of things you can do with this tool:
    - turn all the *blues* in an image into *reds*
    - change the *tint* of an image

- change a greyscale image into a *mauve*-scale image
- increase or decrease the amount of color saturation in an image
- change the overall brightness of an image
- change the overall contrast of an image

- **RGB Modification**: This tool lets you route the red, green, and blue color components of an image through independent mapping functions. The functions can either be the standard *gamma* function, or any arbitrary function that can be drawn with straight line segments or a cubic spline. See “Section 5.3.4: The Intensity Graph” for more info about graph functions.

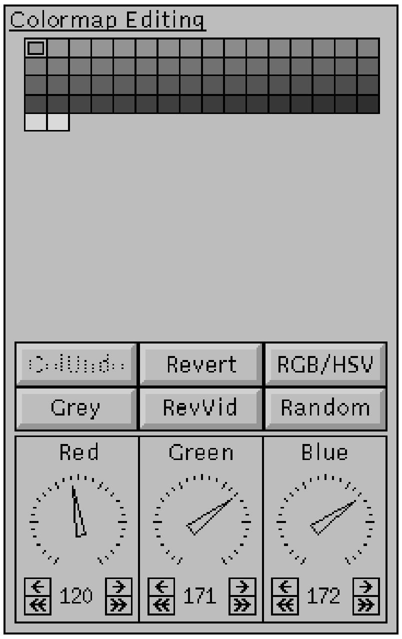
  The major use of the **RGB Modification** tool is to correct for the differing color response curves of various color monitors, printers, and scanners. This is the tool to use when the image is too *red*, for instance.

These three tools are tied together in a fixed order. The **Colormap Editing** tool operates on the original colors in the image. The output of this tool is piped into the **HSV Modification** tool. Its output is piped into the **RGB Modification** tool. The output from the **RGB Modification** tool is what actually gets displayed.

In addition there is a collection of buttons that control the *xv color editor* as a whole (more or less).

Don’t Panic! It’s not as complicated as it looks.

Section 5.2: The Colormap Editing Tool



The top portion of this window shows the colormap of the current image. There are 16 *cells* across, and up to 16 rows down, for a maximum of 256 color cells. Only cells actually *used* somewhere in the image are shown in this array.

The currently selected color cell is shown with a thick border. You can change the selection by clicking anywhere in the array. If you drag the mouse through this area, you'll see the dials at the bottom change to track the current pixel values.

You can also select a color cell by clicking anywhere in the *image* window. Whichever pixel value you were on when you let go of the mouse will become the new selected color cell.

You can define a smoothly gradated range of colors by *Left* clicking on the color cell that marks the 'start' of the range, and *Middle* clicking on the color cell that marks the 'end' of the range. Intervening color cells will be interpolated between the colors of the .'start' and 'end' colorcells.

As an example:

- View the 'default' image by running '`xv`' without specifying any filenames.
- Open the *xv color editor* window, and *Left* click on the first color cell.
- Turn this color cell 'red' by setting the RGB dials to 255,0,0.
- *Left* click on the 64th color cell (the rightmost color cell in the last complete row).
- Turn this color cell 'yellow' by setting the RGB dials to 255,255,0.
- *Middle* click on the first color cell. A smooth series of yellowish-reds will be generated from the 64th color cell to the first color cell. Note that the 'direction' doesn't matter.

Since certain images will have many colors that are the same, or nearly the same, it is sometimes convenient to *group* color cells together. Grouped color cells all take on the same color, and changing any one of them affects all of the other colors in the group.

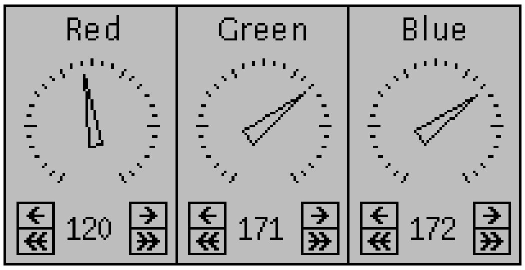
To group color cells together, do the following:

- Hold down the <**shift**> key.
- *Left* click on one color cell that you would like to be in the group
- *Right* click on other color cells that you wish to be in this group. (*Right* clicking on cells that are already selected will de-select them.)
- Release the <**shift**> key when you're done.

You can create as many groups as you like.

You can use this grouping/ungrouping technique to copy colors from one color cell to another. *Left* click on the *source* color cell, *Right* click on the *destination* color cell, and *Right* click on the *destination* color cell again (to ungroup it).

Section 5.2.1: Using the Dial Controls

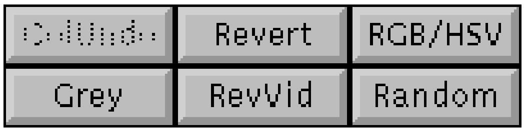


At the bottom the **Colormap Editing** tool are three dials that let you set the color of the current color cell (or group of cells). By default, the dials control the Red, Green, and Blue components of the RGB colorspace, but they can also control the Hue, Saturation, and Value components of the HSV colorspace. (The **RGB/HSV** button controls this.)

Regardless of what they control, all dials in *xv* work the same way. Clicking on the single arrows increase/decrease the value by 1. Clicking on the double arrows increase/decrease the value by a larger amount (16 in this case). If you click on one of the arrows, and hold the mouse button down, the increase/decrease will repeat until you release the mouse button.

You can also click in the general area of the pointer and simply drag it to the position you want. The further your mouse cursor is from the center of the dial, the more precise the control will be. While dragging, you do not have to keep the cursor inside the dial window.

Section 5.2.2: Colormap Editing Commands



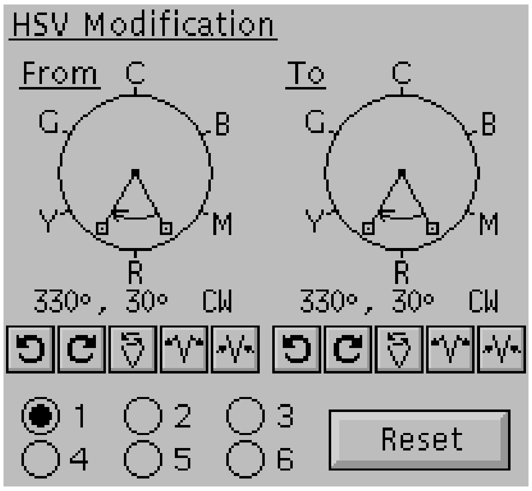
Command	Description
ColUndo	Undoes the last change made to the colormap that resulted in a color cell changing value. This includes grouping and ungrouping color cells, and changing any of the dials.
Revert	Undoes *all* color changes. Returns the colormap to its original state. Destroys any groups that you may have created.
RGB/HSV	Toggles the **Colormap Editing** dials between editing colors in terms of Red, Green, and Blue, and editing colors in terms of Hue, Saturation, and Value.
Grey	Turns color images into greyscale images by changing the colormap. This replaces each color cell with a greyscale representation of itself. Use the **Revert** command to restore the colors.
RevVid	This command behaves differently, depending on the setting of the **RGB/HSV** mode. (You can tell which mode you're in by the titles on the dials.) In RGB mode, each color component is separately 'inverted'. For example, Yellow (which is composed of full red, full green, and no blue) would turn to Blue (no red, no green, full blue). In HSV mode, only the Value (intensity) component is 'inverted'. The Hue and Saturation components remain the same. In this mode, bright colors turn to dark versions of the same color. For example, a Yellow would turn Brown.
Random	Generates a random colormap. This is of questionable usefulness, but it will occasionally come up with pleasing color combinations that you never would've come up with yourself. So it stays in. It works best on images with a small number of colors. Note that it respects cell groupings, so if your image has a lot of colors, you can create a few large groups and then use the **Random** command.

Note: It is HIGHLY RECOMMENDED that if you're using the Colormap Editing tool, you do NOT use the HSV Modification tool or the RGB Modification tool as well. If you do, the results can be quite confusing. For example, you might edit a color cell, and set its color values to produce a *purple*. However, because of HSV/RGB Modification further down the line, the actual color displayed on the image (and in the color cell) is *yellow*. Very confusing, indeed.

Section 5.3: The HSV Modification Tool

There are four separate controls in the **HSV Modification** tool. At the top of the window are a pair of circular controls that handle hue remapping. Lower down is a circular control that maps 'white' (and greys) to a specified color. There is a dial control that lets you saturate/desaturate the colors of the current information. Finally, at the bottom there is a graph window that lets you modify intensity values via an arbitrary remapping function.

Section 5.3.1: Hue Remapping Controls



These two dials are used to define a *source* and a *destination* range of hue values. Every hue in the *source* range (defined in the *From* dial) gets mapped to the value of the corresponding point in the *destination* range (defined in the *To* dial).

Each dial has a pair of radial lines with handles at their ends. Between the two lines an arc is drawn with an arrow at one end. The wedge drawn by these lines and the arc defines a range of values (in degrees). The direction of the arc (clockwise, or counter-clockwise) determines the direction of this range of values (increasing or decreasing).

Distributed around the dial are tick marks and the letters 'R', 'Y', 'G', 'C', 'B', and 'M'. These letters stand for the colors Red, Yellow, Green, Cyan, Blue, and Magenta, and they show where these colors appear on the circle.

The range is shown numerically below the control. By default the range is '`330°, 30° CW`'. This means that a range of values [330°, 331°, 332°, ... 359°, 0°, 1°, ... 28°, 29°, 30°] has been defined. Note that (being a circle) it wraps back to 0° after 359°.

The range can be changed in many different ways. You can click on the 'handles' at the end of the radial lines and move them around. If you click inside the dial, but *not* on one of the handles, you'll be able to drag the range around as a single object. There are also 5 buttons below the dial that let you rotate the range, flip the direction of the range, and increase/decrease the size of the range while keeping it centered around the same value.

In its default state, the *To* dial is set to the same range as the *From* dial. When the two dials are set to the same range, they are effectively 'turned off', and ignored.

An example of hue remapping:

- As a simple example of the sort of things you can do with the hue remapping control, we'll change the background color of the default (*xv* logo) image without changing any other colors in the image. Since the background is composed of a gradient of 64 colors, you would not want to do this with the Colormap Editing tool. It would take forever.

- First, get the default image up on the screen by running '`xv`' without giving any filenames. Open up the *xv color editor* window via the **ColEdit** command.

- Next, click the mouse in the image window and drag it around. You'll see that all the background pixels have the same Hue component value (240).

- To remap this hue, simply adjust the *From* dial so that its range includes this Hue value. The background should change from 'blue' to a reddish color, assuming the *To* dial is still set to its default range (centered around 'R'). If more than the background changed color, you can shrink the *From* range so that it covers fewer colors. In fact, it's possible to shrink the range to the point where it only covers only a single value.

Note that the values printed when you are tracking pixel values in the image are the values *before* the **HSV Modification** tool is applied. For example, the background of the default image will still claim to be blue, regardless of what color you may have changed it to. This is so that you know what Hue value you will need to remap if you want to change its color again.

If you press the **Reset** button that is located near the hue remapping controls, it will effectively disable the hue remapping by setting the *To* range equal to the *From* range.

Below the hue remapping controls are a group of 'radio buttons'. You can have up to six different hue remappings happening simultaneously. Higher numbered mappings take precedence over lower number mappings.

An example of multiple hue remappings:

- Draw a *From* range that is a complete circle. The easiest way to do this is to draw a range that is nearly a full circle, then click and hold down the 'increase range' button located below the *From* range dial until the range stops getting bigger.

- Copy this range to the *To* range by pressing the **Reset** button.

- Rotate the *To* range slightly, by either clicking and dragging anywhere in the *To* range dial, or by using the 'rotate clockwise' and 'rotate counter-clockwise' buttons located below the *To* range.

- You've just built yourself what is effectively a *tint* control.

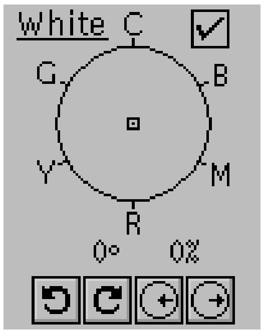
- Now, suppose, you'd like to adjust the background color of your (tint-modified) image, without affecting anything else. Clicking on the background in the image window reveals that the background still has an (original) hue of 240. To modify this hue without affecting anything else, we'll need a second hue remapping.

- Click on the **2** radio button. The dials will change to some other default setting. As before, set the *From* range to encompass the value 240, preferably as ‘tightly’ as possible, and set the *To* range to produce the desired background color.

Note that the six hue remappings are not ‘cascaded’. The output of one remapping is not fed as input into any of the other hue remappings. The hue remappings always operate on the hue values in the original image. In this example, if remapping #1 adds 32 to all hue values, thereby mapping the blue background (value 240) into a purple-blue (value 272), remapping #2 still sees the background at 240, and can remap it to anything it likes. Similarly, in the same example, if remapping #1 has mapped a green-blue color (value 208) into blue (value 240), remapping #2 will not map this into another color. As far as remapping #2 is concerned, that green-blue is still green-blue.

If it seems complicated, I’m sorry. It is.

Section 5.3.2: The White Remapping Control



In the HSV colorspace, ‘white’ (including black, and all the greys in between) has no Hue or Saturation components. As such, it is not possible to use the hue remapping controls to change the color of white pixels in the image, since they have no ‘color’ to change.

The white remapping control provides a way to add Hue and Saturation components to all the whites in the image. It consists of a movable point in a color dial. The angle of the dot from the center of the dial determines the Hue component. The distance of the dot from the center of the dial determines the Saturation component. The further the dot is from the center of the dial, the more saturated the color will be.

You can control the white remapping control in several ways. You can click on the handle and drag it around with the mouse. There are also four buttons provided under the dial. One pair allows you to rotate the handle clockwise and counter-clockwise without changing its distance from the center. The other pair of buttons lets you change the distance between the handle and the center without changing the angle.

The current Hue and Saturation values provided by the control is displayed below the dial. The first number is the Hue component, in degrees, and the second is the Saturation component, as a percentage.

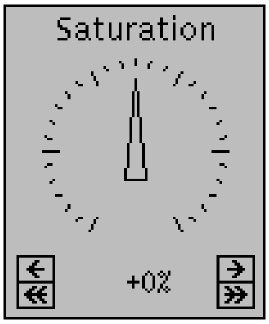
There is also a checkbox that will let you turn off the white remapping control. This lets you quickly compare your modified ‘white’ with the original white. You can also effectively disable the white remapping control by putting the handle back in the center of the control. The easiest way to do this is to click and hold the 'move towards center' button until the saturation gets down to 0%.

Example:

- Press the **Grey** control in the Colormap Editing tool. This turns all the colors in the image into shades of grey.

- Drag the handle in the white remapping control halfway down towards the 'R' mark. The Hue and Saturation values should be roughly 0° and 50%. The image should now be displayed in shades of pink.

Section 5.3.3: The Saturation Control



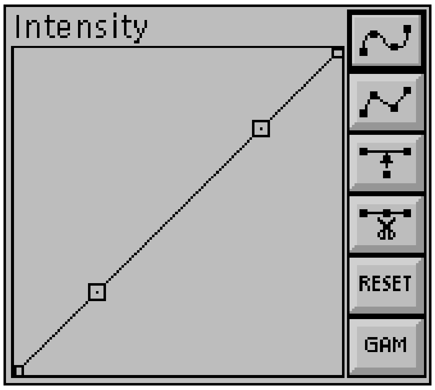
The saturation control lets you globally increase or decrease the color saturation of the image. In effect, it is much like the 'color' control on most color televisions.

The saturation control is a dial that operates exactly like the dials described in "Section 5.2.1 Using the Dial Controls". In short, you can click and hold down any of the four buttons in the bottom of the control to increase or decrease the control's value. You can also click on the dial itself and move the pointer around directly.

The saturation control has values that range from '-100%' to '+100%'. At its default setting of '0%', the saturation control has no effect on the image. As the values increase, the colors become more saturated, up to '+100%' where every color is fully saturated. Likewise, as values decrease, the colors become desaturated. At '-100%', every color will become a completely desaturated (i.e., a shade of grey). Note that this control is applied *after* the the White Remapping control, so if you 'greyify' the image by completely desaturating it, you will not be able to color it using the White Remapping control.

Unless you're trying for some special effects, the useful range of this control is probably '±20%'. Also note that the control will have no effect on shades of grey, as they have no color to saturate.

Section 5.3.4: The Intensity Graph



The intensity graph allows you to change the brightness of the image, change the contrast of the image, and get some unique effects.

The intensity graph is a function that lets you remap intensity values (the Value component in HSV Colorspace) into other intensity values. The input and output values of this function both range from 0 to 255. The input values range along the *x* axis of this graph (the horizontal). For every input value (point along the *x* axis) there is a unique output value determined by the height of the graph at that point. In the graph's default state, the function is a straight line from bottom-left to top-right. In this case, each input value produces an equivalent output value, and the graph has no effect.

There are a number of 'handles' along the graph. These provide your major means of interacting with the graph. You can move them around arbitrarily, subject to these two constraints: the handles at the far left and far right of the graph can only be moved vertically, and handles must remain between their neighboring handles for the graph to remain a proper function.

The handles are normally connected by a spline curve. To see this, move one of the handles by clicking and dragging it. (Note that the *x,y* position of the current handle is displayed while the mouse button is held down.) The function will remain a smoothly curved line that passes through all the handles. You can change this behavior by putting the function into 'lines' mode. Press the 'lines' button (the second button down from the top). The function will change to a series of line segments that connect the handles. Press the 'spline' button (the top button) to go back to 'spline' mode.

The next two buttons let you add or delete handles. The 'add handle' button will insert a handle into the largest 'gap' in the function. The 'delete handle' button will remove a handle from the smallest 'gap' in the function. You can have as little as 2 handles, or as many as 16. Note that as the number of handles gets large, the spline will start getting out of control. You may wish to switch to 'lines' mode in this case.

The 'Reset' button puts everything back on a straight line connecting bottom-left to top-right (a 1:1 function). It does not change the number of handles, nor does it change the *x*-positions of the handles.

The 'Gam' button lets you set the function curve by entering a single number. The function is set equal to the gamma function:

$$Y = 255 \cdot (I \div 255)^{\frac{1}{\gamma}}$$

where I is the input value (0-255), γ is the gamma value, and Y is the computed result.

Gamma values (for our purposes) can range between 0 and 10000, non-inclusive.

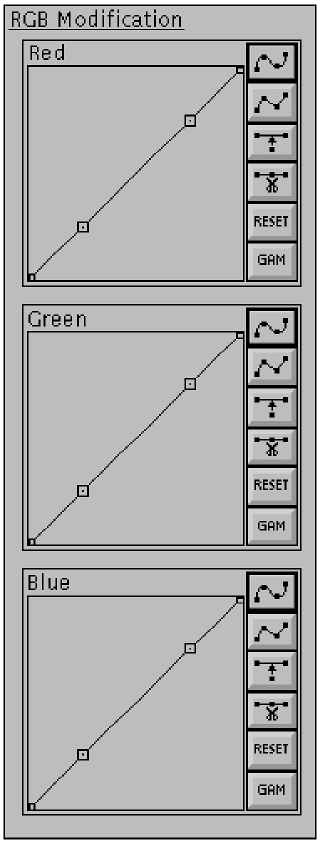
- A gamma value of ‘1.00’ results in the normal 1:1 straight line.
- Gamma values of less than 1.00 but greater than 0.00 result in ‘exponential’ curves, which will dim the image.
- Gamma values greater than 1.00 result in ‘logarithmic’ curves, which will brighten the image. Try it and see.

There is a shortcut for the ‘Gam’ button. Type ‘g’ while the mouse is inside the graph window.

Also, touching any of the handles after a ‘Gam’ command will put the graph back into its ‘normal’ mode. (Either ‘spline’ or ‘lines’ depending on which of the top two buttons is turned on.)

Generally, whenever you move a graph handle and let go of it, the image will be redrawn to show you the effects of what you’ve done. This can be time-consuming if you intend to move many points around. You can temporarily prevent the redisplay of the image by holding down a <**shift**> key. Continue to hold the <**shift**> key down while you move the handles to the new position. Release the <**shift**> key when you’re done, and the image will be redisplayed.

Section 5.4: The RGB Modification Tool



The **RGB Modification** tool is a collection of three graph windows, each of which operate on one of the components of the RGB colorspace. This tool lets you perform global color-correction on the image by boosting or cutting the values of one or more of the RGB color components. You can use this to correct for color screens that are ‘too blue’, or for color printers that produce ‘brownish’ output, or whatever.

The graphs work exactly as explained in “Section 5.3.4: The Intensity Graph”.

Neat Trick: In addition to color-correction, you can use the RGB modification tool to add color to images that didn’t have color to begin with. For instance, you can ‘pseudo-color’ a greyscale image.

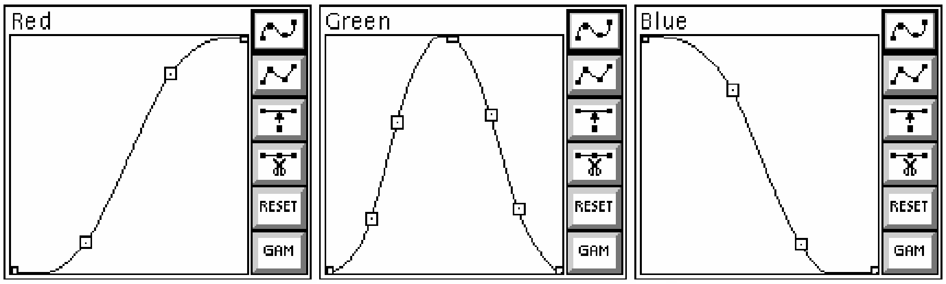
An example of pseudo-coloring:

- Adjust the Red graph so that there is a strong red presence on the right side of the graph, and none on the left, or in the middle.

- Adjust the Green graph so that there is a strong green presence in the middle of the graph, and none on the left or right.

- Adjust the Blue graph so that there is a strong blue presence on the left side of the graph, and none on the left, or in the middle.

- The graphs should look roughly like this:



You now have a transformation that will take greyscale images and display them in pseudo-color, using a ‘temperature’ color scheme. Neato!

Section 5.5: The Color Editor Controls

These buttons provide general control over the whole *xv color editor* window. You can display the image with or without color modification, save and recall presets, and undo/redo changes. Also, convenience controls are given for performing some of the most common operations on the Intensity graph.

Command	Key	Description
Apply	'**p**'	Displays the image using the current HSV and RGB Modifications. Also turns the 'Display with HSV/RGB mods' checkbox on. (See below.) This is only useful when the 'Auto-apply HSV/RGB mods' checkbox is off.
NoMod		Displays the image without any HSV or RGB Modifications. Also turns the 'Display with HSV/RGB mods' checkbox off.
Reset	'**R**'	Resets all HSV and RGB controls to their default settings. Doesn't affect the Colormap Editing tool. Note that these default settings can be changed using the **Set** command described below.
Undo		Undoes the last change to the HSV or RGB controls. It may be helpful to think of *xv* as maintaining a series of 32 'snapshots' of the HSV and RGB controls. You are normally looking at the last frame in this series. The **Undo** control moves you backwards in the series.
Redo		Only available after you've hit **Undo**. Moves you forward in the 'snapshot' series described above. Note that if you have hit **Undo** a few times (i.e., you're now looking at some frame in the middle of the series), and you change an HSV or RGB control, all subsequent frames in the series are thrown away, and the current state becomes that last frame in the series.
1,**2**,**3**,**4**		Pressing any of these buttons recalls a preset (a complete set of values for the HSV and RGB controls).
Set		Used in conjunction with the **Reset**,**1**,**2**,**3**,**4** buttons to store the current settings of the HSV and RGB controls into a preset. To do so, press the **Set** button, and then press one of the **Reset**,**1**,**2**,**3**,**4** buttons. The current HSV and RGB control settings will be stored in that preset, as long as *xv* continues running. The values will be lost when the program exits. It is also possible to save these values permanently. See the **CutRes** button (below) and "Section 9: Modifying XV Behavior" for more details.
CutRes		Copies the current settings of the HSV and RGB controls, as text, into the X server's cut buffer. You can then use a text editor to paste these values into your '.Xdefaults' (or '.Xresources') file. This lets you save the current settings 'permanently'. See "Section 9: Modifying XV Behavior" for more details.
Close		This button closes the *xv color editing* window.
Brite		Brightens the image by moving all the handles in the Intensity graph up by a constant amount.
Dim		Darkens the image by moving all the handles in the Intensity graph

down by a constant amount.

Sharp — Increases the contrast of the image by moving handles on the left side of the Intensity graph down, and handles on the right side up.

Dull — Decreases the contrast of the image by moving handles on the left side of the Intensity graph up, and handles on the right side down.

Norm ‘**N**’ — Normalizes the image so that the darkest pixels in the image are given an intensity of ‘0’, and the brightest pixels are given an intensity of ‘255’. Intermediate colors are interpolated accordingly. This forces the image to have the full (maximum) dynamic range.

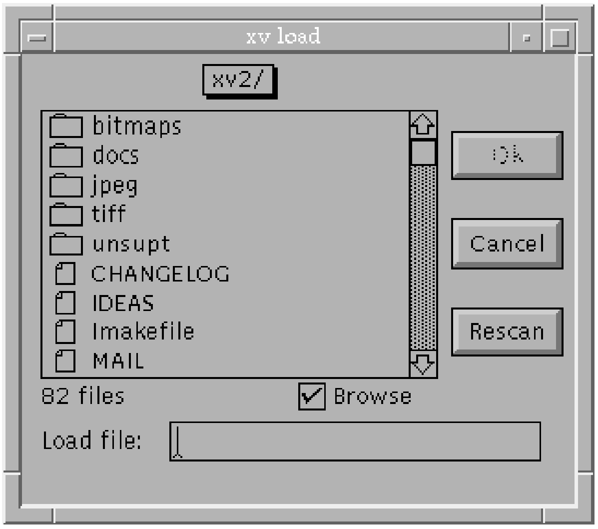
HistEq ‘**H**’ — Runs a histogram equalization algorithm on *the currently displayed region of the image*. That is, if you’re cropped, it will only run the algorithm on the cropped section. Note, however, that the only modification it makes to the image is to generate a bizarre corrective Intensity curve. As such, if you **HistEq** a section of the image, the rest of the image will probably *not* be what you’d want. Also note that the histogram curve will ‘go away’ if you touch any of the handles in the Intensity graph window, just like a ‘gamma’ curve would.

The ‘Display with HSV/RGB mods’ checkbox tells you whether or you’re looking at a modified image (checked) or the ‘raw’, unmodified image (unchecked). The **Apply** and **NoMod** buttons change the setting of this checkbox, and you can also change the checkbox directly by clicking on it.

The ‘Auto-apply HSV/RGB mods’ checkbox controls whether or not the program regenerates and redisplays the image after each change to an HSV or RGB control. By default, this checkbox is turned on, so that you can easily see the results of your modifications. However, in the case that you want to make a large number of changes at once, it might be preferable to turn automatic redisplay off for a while, to speed things up.

The ‘Auto-reset on new image’ checkbox controls whether or not the HSV and RGB controls are **Reset** back to their default values whenever a new image is loaded up. By default, this is also turned on, as when you’re playing with the HSV/RGB controls, you probably only want to affect the current image, and not all subsequently loaded images as well.

Section 6: The Load Window



The *xv load* window lets you load and view images interactively, without specifying them on the command line when you start *xv*.

The load window shows the contents of the current directory in a scrolling window. The files will be sorted alphabetically, with all the directories (and symbolic links to directories, if your operating system supports them) displayed first.

This list window operates in the same way that the one in the *xv controls* window works. (See “Section 3.7.1: Operating a List Window” for details.) In short, you can operate the scroll bar, drag the highlight bar around the window, and use the up-arrow, down-arrow, Home, End, Page Up, and Page Down keys on your keyboard.

Whenever you click on a name in the list (or otherwise change the position of the highlight bar), the name of the highlighted file is copied to the “Load file” text entry region, located below the list window. Pressing the **Ok** button (or typing <**return**>) will cause the program to attempt to load the specified file. If the load attempt is successful, the load window will disappear, and the new image will be displayed. Otherwise, an error message will be displayed, and the load window will remain visible.

The **Browse** checkbox overrides this behavior, and keeps the load window visible until explicitly closed (via the **Cancel** button). This is handy if you’re using *xv* to ‘wander around a directory tree’, and plan to be using the **Load** command quite often.

If the image is successfully loaded, its name will be added to the *xv controls* window list. This will let you quickly reload it later without have to go through the *xv load* window again.

You can also load a file by double-clicking on its name in the file list.

If the specified filename begins with a ‘!’ or ‘|’ character, the filename will be interpreted as a shell command to run. The leading ‘!’ or ‘|’ is dropped, and the rest of the line is fed to the default system shell. The command is expected to generate an image in one of the *xv*-recognized formats as its standard output. If the command returns non-zero, *xv* assumes it failed, and doesn't try to read the output produced. You can pipe multiple commands together. For example loading "! `xwd | xwdtopnm`" would run *xwd* to generate a window dump, pipe that to *xwdtopnm* to convert it to ppm file, and that file would be piped to *xv*.

If the specified file is a directory, *xv* will figure that out and (instead of loading it) will 'cd' to that directory, and display its contents in the list window.

Above the list window is a pop-up menu button, much like the **Display Modes** button in the *xv controls* window. It normally displays the name of the current directory. If you click this button, and hold the mouse down, the complete path will be shown, one directory per line. You can go 'up' the directory tree any number of levels, all the way up to the root directory, by simply selecting a directory name in this list.

For those who prefer the direct approach, you can simply type file or directory names in the "Load file" text entry region. If you type a directory name and hit <**return**>, *xv* will 'cd' to that directory and display its contents in the list window. If you type a file name and hit <**return**>, *xv* will attempt to load the file. You can enter relative paths (relative to the currently displayed directory), absolute paths, and even paths that begin with a '~'.

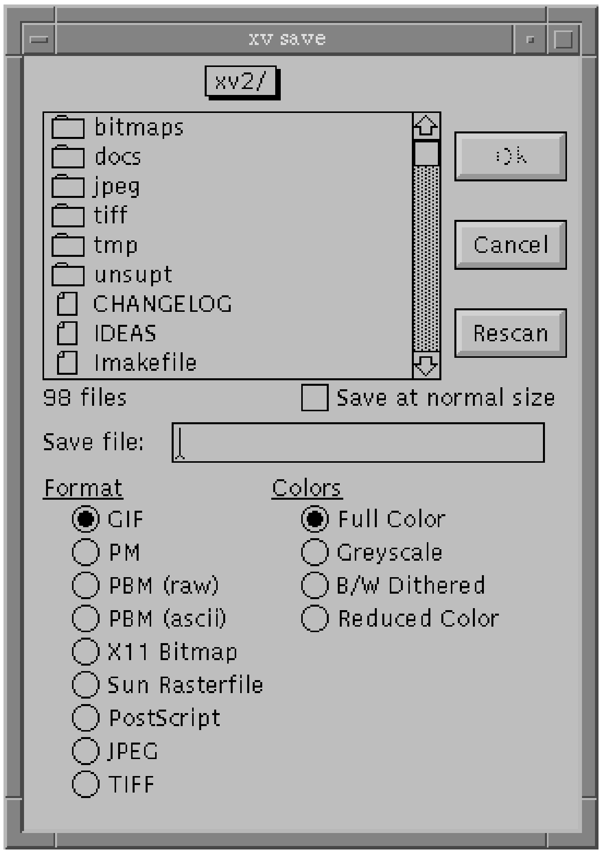
The "Load file" text entry region supports a number of *emacs*-like editing keys.

Ctrl-F	moves the cursor forward one character
Ctrl-B	moves the cursor backward one character
Ctrl-A	moves the cursor to the beginning of the line
Ctrl-E	moves the cursor to the end of the line
Ctrl-D	deletes the character to the right of the cursor
Ctrl-U	clears the entire line
Ctrl-K	clears from the cursor position to the end of the line.

If the filename is so long that it cannot be completely displayed in the text entry region, a thick line will appear on the left or right side (or both sides) of the region to show that "there's more over this way".

Pressing the **Rescan** button will rescan the current directory. While the contents of the current directory are read each time the load window is opened, it is perfectly possible (given a multitasking operating system) that some other program may add, delete, or rename files in the current directory. *xv* would not know if this happened. The **Rescan** button gives you an easy way of 'kicking' *xv* into looking again.

Section 7: The Save Window



The *xv save* window lets you write images back to disk, presumably after you've modified them. You can write images back in many different formats, not just the original format. **Warning**! *Images are saved as they are currently shown* (at the current size, with the current color modification, rotation, cropping, etc. applied). The only exception to this rule is if you are on a 1-bit B/W display. The fact that images have to be stippled in black and white in order to be displayed on such a screen doesn't count as 'modification', and the file won't be saved 'as displayed'. You can manipulate and save full-color images on such a display, even if you can't really *see* them.

For the most part, the *xv save* window operates exactly like the *xv load* window. (See "Section 6: The Load Window" for details.) Only the differences are listed here.

When the window is opened, it should have the filename of the currently loaded file already entered into the text entry region. If you click on a file name in the list window, this name will be cleared and replaced with the new name.

At the bottom of the window are a list of possible formats in which you can save the file. If you click on one of these formats, and your filename has a recognized suffix (i.e., '.gif', '.GIF', '.pbm', etc.), the suffix portion of your filename will be replaced with the new, appropriate suffix for the selected format.

You can pipe output from *xv* to other programs by using the *xv save* window. If the first character of the specified filename is '!' or '|', the rest of the filename is interpreted as a command to pipe output to, in the currently selected image format. A fine use for this feature is directly printing images to a PostScript printer by selecting 'PostScript' in the formats list, and typing something like "`| lpr`" as the filename. In this case, *xv* will create a temporary file, write the PostScript to that file, and cat the contents of that file to the entered command. *XV* will wait for the command to complete. If the command completed successfully, the *xv save* window will disappear. If the command was unsuccessful, the window will remain visible. In any event, the temporary file will be deleted.

There is a "Save at normal size" checkbox. Normally, when you save an image, it will be saved at the current expansion (ie, one screen pixel will map to one image pixel in the saved file). Sometimes, however, it's desirable to save an image at its original size. This is most relevant when you're viewing images larger than your screen. By default, *xv* will automatically shrink images so that they *do* fit on the screen. If you save these images, you'll find that you've lost a lot of data, that maybe you wanted to keep. That's what this checkbox is here for. Note: certain operations, such as **Smooth** and **Dither**, only affect the 'displayed' image. If you choose to save an image at its normal size, these effects will *not* be in the saved image.

At the bottom right side of the window there is a list of possible 'Color' variations to save. Most file formats support different 'sub-formats' for 24-bit color, 8-bit greyscale, 1-bit B/W stippled, etc. Not all of them do. Likewise, not all 'Color' choices are available in all formats.

In general, the 'Color' choices do the following:

Full Color — Saves the image as currently shown with all color modifications, cropping, rotation, flipping, resizing, and smoothing. The image will be saved with all of its colors, even if you weren't able to display them all on your screen. For example, you can load a color image on a 1-bit B/W display, modify it, and write it back. The saved image will still be full color, even though all you could see on your screen was some B/W-dithered nightmare.

Greyscale — Like **Full Color**, but saves the image in a greyscale format.

B/W Dithered — Like **Full Color**, but before saving the image *xv* generates a 1-bit-per-pixel, black-and-white dithered version of the image, and saves that, instead.

Reduced Color — Saves the image as currently shown, with all color modifications, cropping, rotation, flipping, resizing, and smoothing. The image will be saved *as shown* on the screen, with as many or few colors as *xv* was able to use on the display. The major purpose of this is to allow special effects (color reduction) to be saved, in conjunction with the '`-ncols`' command line option. You will probably never need to use this.

Format notes:

GIF — While *xv* can read both the GIF87a and GIF89a formats, it will only write GIF87a. This is in keeping with the GIF89 specification, which states that if you don't need any of the features added in GIF89 (which *xv* doesn't), you should continue to write GIF87, for greater compatibility with old GIF87-only readers.

Since GIF only supports one format (up to 8 bits per pixel, with a colormap), there will be no size difference between a **Full Color** and a **Greyscale** image. A **B/W Dithered** image, on the other hand, will be considerably smaller.

PM — **Full Color** images are saved in the 3-plane, 1-band, PM_C format. **Greyscale** and **B/W Dithered** images are both saved in the 1-plane, 1-band, PM_C format. As such, there is no size advantage to saving in the **B/W Dithered** format.

PBM (raw) — **Full Color** images are saved in PPM format. **Greyscale** images are saved in PGM format. **B/W Dithered** images are saved in PBM format. Each of these formats are tailored to the data that they save, so PPM images are larger than PGM images, which are in turn larger than PBM images.

In the raw variation of the PBM formats, the header information is written in plain ASCII text, and the image data is written as binary data. This is the more popular of the two dialects of PBM.

PBM (ascii) — Like **PBM (raw)**, only the image data is written as ASCII text. As such, images written in this format will be several times larger than images written in **PBM (raw)**. This is a pretty good format for interchange between systems because it is easy to parse. Also, since they are pure, printable ASCII text, images saved in this format can be mailed, without going through a *uuencode*-like program.

Note that *xv*-produced PBM files may break some PBM readers that do not correctly parse comments. If your PBM reader cannot parse comments, you can easily edit the PBM file and remove the comment lines. A comment is everything from a “#” character to the end of the line.

X11 Bitmap — Saves files in the format used by the ‘*bitmap*’ program, which is part of the standard X11 distribution. Since bitmap files are inherently 1-bit per pixel, you can only select the **B/W Dithered** option for this format.

Sun Rasterfile — **Full/Reduced Color** images are stored in a 24-bit RGB format, **Greyscale** images are stored in an 8-bit greyscale format, and **B/W Dithered** images are stored in a 1-bit B/W format.

PostScript — **Full/Reduced Color** images are stored in a 24-bit RGB format, **Greyscale** images are stored in an 8-bit greyscale format, and **B/W Dithered** images are stored in a 1-bit B/W format.

XV writes Encapsulated PostScript, so you can incorporate *xv*-generated PostScript into many desktop-publishing programs. *XV* also prepends some color-to-greyscale code, so even if your printer doesn’t support color, you can still print ‘color’ PostScript images. These images will be three times larger (in file size) than their greyscale counterparts, so it’s a good idea to save

Greyscale PostScript, unless you know you may be printing the file on a color printer at some point.

Also, you should probably never need to generate **B/W Dithered** PostScript, as every PostScript printer I've ever heard of can print greyscale images. The only valid cases I can think of are: A) doing it for a special effect, and B) doing it to generate a much smaller (roughly 1/8th the size) PostScript file.

Note: When you try to save a PostScript file, the *xv postscript* window will pop up to let you specify how you want the image printed. (See "Section 8: The PostScript Window", for details.)

JPEG

XV writes files in the JFIF format created by the Independent JPEG Group. **Full Color** images are written in a 24-bit RGB format, and **Greyscale** images are written in an 8-bit greyscale format. **B/W Dithered** images should *not* be used, as they will probably wind up being *larger* than **Greyscale** versions of the same images, due to the way JPEG works. Note: You cannot write a **Reduced Color** JPEG file. Trust me, given the method that JPEG uses to compress, it's not in your best interest to save **Reduced Color** JPEG files. If you attempt to, a **Full Color** JPEG file will be saved.

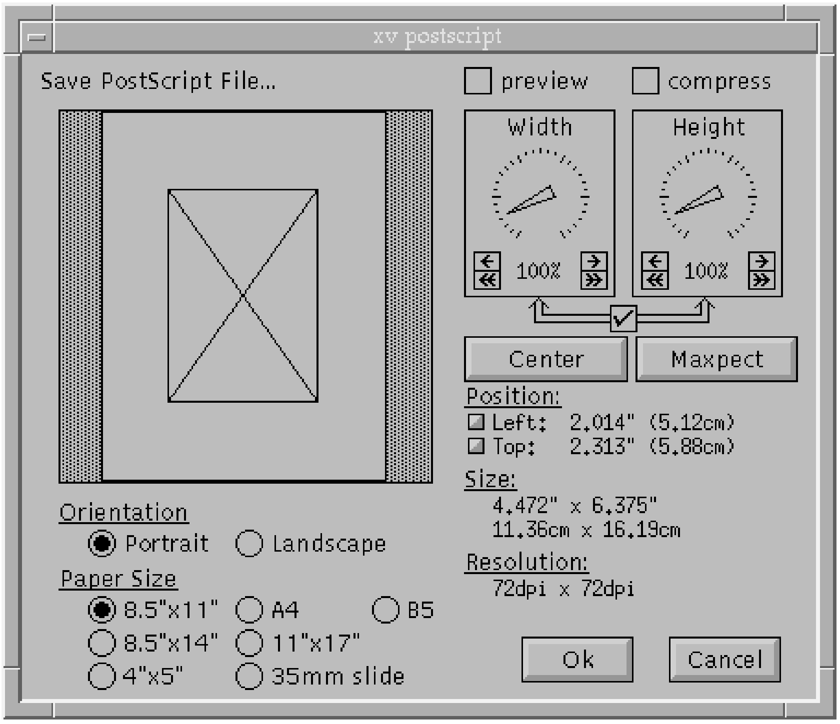
When you save in the JPEG format, a dialog box will pop up and ask you for a quality setting. '75%' is the default value, and really, it's a fine value. You shouldn't have to change it unless you're specifically trying to trade off quality for compression, or vice versa. The useful range of values is 5%-95%.

TIFF

Full/Reduced Color images are written in a 24-bit RGB format, and **Greyscale** images are written in an 8-bit greyscale format. **B/W Dithered** images are written in a 1-bit B/W format.

When you save in the TIFF format, a dialog box will pop up and ask you which type of image compression it should use. **None**, **LZW**, and **PackBits** compression types are available for **Full/Reduced Color**, **Greyscale**, and **B/W Dithered** modes. In addition, there are two **B/W Dithered**-only algorithms, **CCITT Group3** and **CCITT Group4**.

Section 8: The PostScript Window



The *xv postscript* window lets you describe how your image should look when printed. You can set the paper size and the image size, position the image on the paper, and print in ‘portrait’ or ‘landscape’ mode.

The majority of the *xv postscript* window is taken up by a window that shows a white rectangle (the page) with a black rectangle (the image) positioned on it. You can position the image rectangle anywhere on the page. The only constraint is that the center of the image (where the two diagonal lines meet) must remain on the page. Only the portion of the image that is on the page will actually be printed.

The image can be (roughly) positioned on the page by clicking in the image rectangle and dragging it around. As you move the image, the “Top” and “Left” position displays will show the size of the top and left margins (the distance between the top-left corner of the page and the top-left corner of the image).

You’ll note that you have limited placement resolution with the mouse. If you want to fine-position the image, you can use the arrow keys to move the image around. The arrow keys will move the image in .001” increments. You can hold them down, and they will auto-repeat. You can also hold a <**shift**> key down while using the arrow keys. This will move the image in .010” increments.

You can change the size of the printed image by adjusting the “Width” or “Height” dials. Normally, the dials are locked together, to keep the aspect ratio of the image constant. You can unlock the dials by turning the off the checkbox located below the dials. As you change the dials, the size of the image (when printed) is displayed below, in inches and in millimeters. The current resolution of the image is also displayed below. The “Resolution” numbers tell you how many image pixels will be printed per inch.

Located below the ‘page’ rectangle are a set of radio buttons that let you specify the current paper size (8.5” x 11”, 8.5” x 14”, 11” x 17”, A4, B5, 4”x5”, and 35mm), and orientation (Portrait and Landscape).

The **Center** button will center the image on the page. The **Maxpect** button will make the image as large as possible (maintaining half-inch margins on all sides) without changing the aspect ratio.

There are a pair of small buttons located next to the “Left” and “Top” displays. Clicking the “Left” one will cycle between displaying the “Left” margin, the “Right” margin, and the “Center X” position (the distance from the left edge of the paper to the center of the image).

Clicking the “Top” display’s button will cycle between displaying the size of the “Top” margin, the size of the “Bottom” margin, and the “Center Y” position (the distance from the top edge of the paper to the center of the image).

At the top of the *xv postscript* window are a pair of checkboxes. The ‘preview’ checkbox lets you specify whether or not to include a b/w preview of the image in the PostScript file. Certain desktop publishing programs may make use of such a preview.

The ‘compress’ checkbox lets you specify whether or not to generate compressed 8-bit per pixel PostScript. This is particularly handy if you’re generate color postscript, as color postscript files are normally three times larger than their greyscale counterparts. Compression can shrink these color postscript files by a factor of 4:1. It has a lesser effect on greyscale images. It should be noted, however, that compressed PostScript files may take 2-3 times longer to print than uncompressed PostScript files. However, if you are connected to your laser printer via a slow 9600 baud serial line, the decreased transmission time due to compressed data may more than make up for the increased execution time. You’ll have to decide for yourself.

Click the “Ok” button when you’re finished with the *xv postscript* window. If everything is successful, the *xv postscript* and the *xv save* window will both close. If *xv* was unable to write the PostScript file, the *xv postscript* window will close, but the *xv save* window will remain open, to give you a chance to enter a different filename.

Section 9: Modifying XV Behavior

XV supports literally dozens of command line options and X11 resources. Fortunately, it is doubtful that you'll ever need to use more than a small few. The rest are provided mainly for that 'one special case' application of *xv*... Note that you do not have to specify the entire option name, only enough characters to uniquely identify the option. Thus, '`-geom`' is a fine abbreviation of '`-geometry`'. The shortest legal abbreviations are shown in bold face.

Section 9.1: Command Line Options Overview

If you start *xv* with the command '`xv -help`', the current list of options will be displayed:

```
xv  [-] [-2xlimit] [-aspect w:h] [-best24] [-bg color] [-black color] [-browse]
    [-bw width] [-cecmap] [-cegeometry geom] [-cemap] [-cgeometry geom] [-clear]
    [-cmap] [-crop] [-cursor char#] [-DEBUG level] [-display disp] [-dither]
    [-expand exp] [-fg color] [-fixed] [-geometry geom] [-help] [-hi color] [-hsv]
    [-igeometry geom] [-imap] [-keeparound] [-lo color] [-loadclear] [-max]
    [-maxpect] [-mono] [-ncols #] [-nglobal] [-ninstall] [-nolimits] [-nopos]
    [-noqcheck] [-noresetroot] [-nostat] [-owncmap] [-perfect] [-quick24] [-quit]
    [-rbg color] [-rfg color] [-rgb] [-rmode #] [-root] [-rv] [-rw] [-slow24]
    [-smooth] [-visual type] [-wait seconds] [-white color] [-wloop] [filename ...]
```

Section 9.2: General Options

-help
: Print usage instructions, listing the current available command-line options. Any unrecognized option will do this as well.

-display *disp*
: Specifies the display that *xv* should attempt to connect to. If you don't specify a display, *xv* will use the environment variable `$DISPLAY`.

-fg *color*
: Sets the foreground color used by the windows. (Resource name: *foreground*. Type: string)

-bg *color*
: Sets the background color used by the windows. (Resource name: *background*. Type: string)

-hi *color*
: Sets the highlight color used for the top-left edges of the control buttons. (Resource name: *highlight*. Type: string)

-lo *color*
: Sets the lowlight color used for the bottom-right edges of the control buttons, and also the background of some windows. (Resource name: *lowlight*. Type: string)

-bw *bwidth*
: Sets the width of the border on the windows. Your window manager may choose to ignore this, however. (Resource name: *borderWidth*. Type: integer)

Section 9.3: Image Sizing Options

–geometry *geom* — Lets you specify the size and placement of the 'image' window. It's most useful when you only specify a position, and let *xv* choose the size. If you specify a size as well, *xv* will create a window of that size, unless `–fixed` is specified. The *geom* argument is in the form of a normal X geometry string (e.g. "300x240" or "+10+10" or "400x300+10+10"). (Resource name: *geometry*. Type: string)

–fixed — Only used in conjunction with the `–geometry` option. If you specify a window size with the `–geometry` option, *xv* will normally stretch the picture to exactly that size. This is not always desirable, as it may seriously distort the aspect ratio of the picture. Specifying the `–fixed` option corrects this behavior by instructing *xv* to use the specified geometry size as a *maximum* window size. It will, however, preserve the original aspect ratio of the picture.

For example, if you give a rectangular geometry of '320x240', and you try to display a square picture with a size of '256x256', the window opened will actually be '240x240', which is the largest square that still fits in the '320x240' rectangle that was specified. (Resource name: *fixed*. Type: boolean)

–expand *exp* — Lets you specify an initial expansion or compression factor for the picture. You can specify floating-point values. Values larger than zero multiply the picture's dimensions by the given factor. (i.e., an expand factor of '3' will make a 320x200 image display as 960x600).

Factors less than zero are treated as reciprocals. (i.e., an expand factor of '-4' makes the picture 1/4th its normal size.). '0' is not a valid expansion factor. (Resource name: *expand.* Type: floating-point)

–aspect *w:h* — Lets you set an initial aspect ratio, and also sets the value used by the **Aspect** control. The aspect ratio of nearly every X display (and, in fact, any civilized graphics display) is 1:1. What this means is that pixels appear to be 'square'. A 100 pixel wide by 100 pixel high box will appear on the screen as a square. Unfortunately, this is not the case with some screens and digitizers. The `–aspect` option lets you stretch the picture so that the picture appears correctly on your display. Unlike the other size-related options, this one doesn't care what the size of the overall picture is. It operates on a pixel-by-pixel basis, stretching each image pixel slightly, in either width or height, depending on the ratio.

Aspect ratios greater than '1:1' (e.g., '4:3') make the picture wider than normal. Aspect ratios less than '1:1' (e.g. '2:3') make the picture taller than normal. (Useful aspect ratio: A 512x480 image that was supposed to fill a standard 4x3 video screen (produced by many video digitizers) should be displayed with an aspect ratio of '5:4') (Resource name: *aspect*. Type: string)

Section 9.4: Color Allocation Options

-ncols *nc*
: Sets the maximum number of colors that *xv* will use. Normally, this is set to 'as many as it can get'. However, you can set this to smaller values for interesting effect. If you set it to '0', it will display the picture by dithering with 'black' and 'white'. (The actual colors used can be set by the `-black` and `-white` options, below.) (Resource name: *ncols*. Type: integer)

-nglobal
: Adjusts the way the program behaves when it is unable to get all the colors it requested. Normally, it will search the display's default colormap, and 'borrow' any colors it deems appropriate. These borrowed colors are, however, *not* owned by *xv*, and as such, can be changed without *xv's* permission, or knowledge. If this happens, the displayed picture will change, probably in a less-than-desirable fashion. If you specify the `-nglobal` option, *xv* will not use 'global' colors. It will only use colors that it successfully allocated, which makes it immune to any color changes.

It should be noted that 'use global colors' is the default because color changes aren't generally a problem if you are only using *xv* to display a picture for a short time. Color changes only really become a problem if you use *xv* to display a picture that you will be keeping around for a while, while you go and do some other work (such as using *xv* to display a background). In such cases you will want to specify `-nglobal`. Note: using the `-ncols` or `-root` options automatically turn on `-nglobal`. (Resource name: *nglobal*. Type boolean)

-rw
: Tells *xv* to use read/write color cells. Normally, *xv* allocates colors read-only, which allows it to share colors with other programs. If you use read/write color cells, no other program can use the colors that *xv* is using, and vice-versa. The only reason you'd do such a thing is that using read/write color cells allows the **Apply** function in the *xv color editor* window to operate much faster. (Resource name: *rwColor*. Type: boolean)

-perfect
: Makes *xv* try 'extra hard' to get all the colors it wants. In particular, when `-perfect` is specified, *xv* will allocate and install its own colormap if (and only if) it was unable to allocate all the desired colors. This option is not allowed in conjunction with the `-root` option. (Resource name: *perfect*. Type boolean)

-owncmap
: Like '`-perfect`', only this option forces *xv* to *always* allocate and install its own colormap, thereby leaving the default colormap untouched. (Resource name: *ownCmap*. Type boolean)

-cecmap
: Specifies whether *xv* installs the image's colormap in the *xv color editor* window, as well as in the image's window. By default, the program does install the colormap in the color editor window, however this can occasionally make the color editor window unreadable. (This option only applies when the '`-perfect`' or '`-owncmap`' options create their own colormaps.) (Resource name: *ceditColorMap.* Type boolean)

-ninstall
: Prevents *xv* from 'installing' its own colormap, when the `-perfect or` `-owncmap` options are in effect. Instead of installing the colormap, it will merely 'ask the window manager, nicely' to take care of it. This is the

correct way to install a colormap (i.e., ask the WM to do it), unfortunately, it doesn't actually seem to work in many window managers, so the default behavior is for *xv* to handle installation itself. However, this has been seen to annoy one window manager (*dxwm*), so this option is provided if your WM doesn't like programs installing their own colormaps. (Resource name: *ninstall*. Type: boolean)

Section 9.5: 24-bit Conversion Options

The following options only come into play if you are using *xv* to display 24-bit RGB data (PPM files, color PM files, JPEG files, the output of *bggen,* etc.). They have no effect whatsoever on how GIF pictures or 8-bit greyscale images are displayed.

-quick24 — Forces *xv* to use the 'quick' 24-bit to 8-bit conversion algorithm. This algorithm dithers the picture using a fixed set of colors that span the entire RGB colorspace. In versions of *xv* prior to 2.10, this was the default algorithm. It no longer is. (Resource name: *quick24*. Type: boolean)

-slow24 — Forces *xv* to use the 'slow' 24-bit to 8-bit conversion algorithm. This algorithm uses a version of Heckbert's median cut algorithm to pick the 'best' colors on a per-image basis, and dithers with those. This is the current default conversion algorithm. (Resource name: *slow24*. Type: boolean)

Advantages: The `-slow24` algorithm often produces better looking pictures than the `-quick24` algorithm.

Disadvantages: The `-slow24` algorithm is about half as fast as the `-quick24` algorithm. Also, since the colors are chosen on a per-image basis, it can't be used to display multiple images simultaneously, as each image will almost certainly want a different set of 256 colors. The `-quick24` algorithm, however, uses the same exact colors for all images, so it can display many images simultaneously, without running out of colors.

-best24 — Forces *xv* to use the same algorithm used in the program *ppmquant,* written by Jef Poskanzer. This algorithm also uses a version of Heckbert's median cut algorithm, but is capable of picking 'better' colors than the `-slow24` algorithm, and it *doesn't* dither. (Resource name: *best24*. Type: boolean)

Advantages: Generally produces slightly better images than the `-slow24` algorithm. Also, the images are undithered, so they look better when expanded.

Disadvantages: *Much* slower than the `-slow24` algorithm. Like, 5 to 10 times slower. The images produced aren't *that* much better than those produced by the `-slow24` algorithm.

-noqcheck — Turns off a 'quick check' that is normally made. Normally, before running any of the 24-bit to 8-bit conversion algorithms, *xv* determines whether the picture to be displayed has more than 256 unique colors in it. If the picture doesn't, it will treat the picture as an 8-bit colormapped image (i.e., GIF), and won't run either of the conversion algorithms. (Resource name: *noqcheck*. Type: boolean)

Advantages: The pictures will be displayed 'perfectly', whereas if they went through one of the conversion algorithms, they'd probably be dithered.

Disadvantages: Often uses a lot of colors, which limits the ability to view multiple images at once. (See the `-slow24` option above for further info about color sharing.)

Section 9.6: Root Window Options

xv has the ability to display images on the root window of an X display, rather than opening its own window (the default behavior). When using the root window, the program is somewhat limited, because the program cannot receive input events (key press and mouse clicks) from the root window. As a result, you cannot track pixel values, nor crop, nor can you use keyboard commands while the mouse is in the root window.

-root — Directs *xv* to display images in the root window, instead of opening its own window. Exactly *how* the images will be displayed in the root window is determined by the setting of the `-rmode` option. (Resource name: <none>)

-rmode *mode* — Determines how images are to be displayed on the root window, when `-root` has been specified. You can find the current list of 'modes' by using a *mode* value of '-1'. *XV* will complain, and show a list of valid modes. The current list at of the time of this writing is:

0: tiling
1: integer tiling
2: mirrored tiling
3: integer mirrored tiling
4: centered tiling
5: centered on a solid background
6: centered on a 'warp' background
7: centered on a 'brick' background

The default mode is '0'. See "Section 3.5: The Display Modes Menu" for a description of the different display modes. (Resource name: *rootMode*. Type: integer)

-noresetroot — Lets you turn off the clearing of the root window that happens when you switch from a 'root' displaymode back to the 'window' display mode. Handy if you're trying to create a neat mirrored root tile, and you have to keep adjusting your cropping. Or something like that. (Resource name: *resetroot*. Type: boolean)

-rfg *color* — Sets the 'foreground' color used in some of the root display modes. (Resource name: *rootForeground*. Type: string)

-rbg *color* — Sets the 'background' color used in some of the root display modes. (Resource name: *rootBackground*. Type: string)

-max Makes *xv* automatically stretch the image to the full size of the screen. This is mostly useful when you want *xv* to display a background. While you could just as well specify the dimensions of your display ('`-geom 1152x900`' for example), the `-max` option is display-independent. If you decide to start working on a 1280x1024 display the same command will still work. Note: If you specify `-max` when you *aren't* using `-root`, the behavior is slightly different. The image will be made as large as possible while still preserving the normal aspect ratio. (Resource name: <none>)

-maxpect Makes the image as large as possible while preserving the aspect ratio. (Resource name: <none>)

-quit Makes *xv* display the (first) specified file and exit, without any user intervention. Since images displayed on the root window remain there until explicitly cleared, this is very useful for having *xv* display background images on the root window in some sort of start-up script. This is only useful if you are using `-root`. (Resource name: <none>)

-clear Clears the root window of any *xv* images. Note: it is not necessary to do an '`xv -clear`' before displaying another picture in the root window. *xv* will detect that there's an old image in the root window and automatically clear it out (and free the associated colors). (Resource name: <none>)

Section 9.7: Window Options

XV currently consists of three main windows, plus one window for the actual image. These three windows (the *xv controls* window, the *xv info* window, and the *xv color editor* window) may be automatically mapped and positioned when the program starts.

-cmap Maps the *xv controls* window. (Resource name: *ctrlMap*. Type: boolean)

-cgeom *geom* Sets the initial geometry of the *xv controls* window. Note: only the position information is used. The window is of fixed size. (Resource name: *ctrlGeometry*. Type: string)

-imap Maps the *xv info* window. (Resource name: *infoMap*. Type: boolean)

-igeom *geom* Sets the initial geometry of the *xv info* window. Note: only the position information is used. The window is of fixed size. (Resource name: *infoGeometry*. Type: string)

-cemap Maps the *xv color editor* window. (Resource name: *ceditMap*. Type: boolean)

-cegeom *geom* Sets the initial geometry of the *xv color editor* window. Note: only the position information is used. The window is of fixed size. (Resource name: *ceditGeometry*. Type: string)

-nopos Turns off the 'default' positioning of the various *xv* windows. Every time you open a window, you will be asked to position it. (Assuming your window manager asks you such things. *mwm*, for instance, doesn't seem to ask) (Resource name: *nopos*. Type: boolean)

Section 9.8: Miscellaneous Options

-mono

Forces the image to be displayed as a greyscale. This is most useful when you are using certain greyscale X displays. While *xv* attempts to determine if it's running on a greyscale display, many X displays *lie*, and claim to be able to do color. (This is often because they have color graphics boards hooked up to b/w monitors. The computer, of course, has no way of knowing what type of monitor is attached.) On these displays, if you don't specify `-mono`, what you will see is a greyscale representation of one of the RGB outputs of the system. (For example, you'll see the 'red' output on our greyscale Sun 3/60s.) The `-mono` option corrects this behavior. (Resource name: *mono*. Type: boolean)

-white *color*

Specifies the 'white' color used when the picture is b/w stippled. (When '`-ncols 0`' has been specified, or when viewing a b/w image.) (Resource name: *white*. Type: string)

-black *color*

Specifies the 'black' color used when the picture is b/w stippled. (When '`-ncols 0`' has been specified, or when viewing a b/w image.) (Resource name: *black*. Type: string)

Try something like:
'`xv -ncols 0 -bl red -wh yellow <filename>`'
for some interesting, late-60's-style psychedelia effects.

-wait *secs*

Turns on a 'slide-show' feature. Normally, if you specify multiple input files, *xv* will display the first one, and wait for you to give the **Next** command (or whatever). The `-wait` option makes *xv* wait the specified number of seconds, and then go on to the next picture, without any user intervention. The program still accepts commands, so it's possible to 'abort' the current picture without waiting the full specified time by using the **Next** command. (Resource name: <none>)

-wloop

Normally, when running a slide-show with the `-wait` option, *xv* will terminate after displaying the last image. If you also specify the `-wloop` option, the program will loop back to the first image and continue the slide-show until the user issues the **Quit** command. (Resource name: <none>)

-rgb

Specifies that, by default, the colormap editing dials in the *xv color editor* window should be in RGB mode. This is the normal default behavior. (Resource name: *hsvMode.* Type: boolean)

-hsv

Specifies that, by default, the colormap editing dials in the *xv color editor* window should be in HSV mode. (Resource name: *hsvMode*. Type: boolean)

-dither

When specified, tells *xv* to automatically issue a **Dither** command whenever an image is first displayed. Useful on displays with limited color capabilities (4-bit and 6-bit displays.) (Resource name: *autoDither*. Type: boolean)

-smooth — When specified, tells *xv* to automatically issue a **Smooth** command whenever an image is first displayed. This is useful when you are using one of the image sizing options (such as '`-expand`' or '`-max`'). (Resource name: *autoSmooth.* Type: boolean)

-crop — When specified, tells *xv* to automatically issue an **AutoCrop** command whenever an image is first displayed. (Resource name: *autoCrop*. Type: boolean)

-browse — Turns on the **Browse** checkbox in the *xv load* window. This prevents the load window from being automatically closed whenever you successfully load an image. (Resource name: *browseMode*. Type: boolean)

-nostat — Speeds up the performance of the load and save windows. Prevents the program from doing a `stat()` system call on each file in the current directory whenever you change directories. This is handy on systems with a lot remote files, where doing the `stat()` calls takes too long. One downside: subdirectories will not be shown with the little folder icons, as it requires a `stat()` call to determine whether a file is a subdirectory or a data file. This will not affect the operation of the program, just the 'niceness'. (Resource name: *nostat*. Type: boolean)

-visual *vistype* — Normally, *xv* uses the default *visual* model provided by your X server. You can override this by explicitly selecting a visual to use. Valid types are *StaticGray*, *StaticColor*, *TrueColor*, *GrayScale*, *PseudoColor*, and *DirectColor*. Not all of these are necessarily provided on any given X display. Run '*xdpyinfo*' on your display to find out what visual types are supported. (Resource name: *visual*. Type: string)

-cursor *curs* — Specifies an alternate cursor to use in the image window (instead of the normal 'cross' cursor). *curs* values are obtained by finding the character number of a cursor you like in the 'cursor' font. (Run '`xfd -fn cursor`' to display the cursor font.) For example, a *curs* value of '56' corresponds to the (singularly useless) 'Gumby' cursor. (Resource name: *cursor.* Type: integer)

-keeparound — The '-keeparound' option is now poorly named, as I've changed the default behavior. By default, if you **Delete** the last file in the *xv controls* list, nothing unexpected will happen. If you specify the '-keeparound' option, which toggles the '-keeparound' flag *off* now, the program will automatically exit as a convenience. Should probably rename it '–goaway', or something... (Resource name: *keepAround.* Type: boolean)

-2xlimit — By default, *xv* prevents the image window from ever getting larger than the screen. Unfortunately, because of this, if you load an image that is larger than your screen, the image will be shrunk until it fits on your screen. Some folks find this undesirable behavior. Specifying the `-2xlimit` option doubles the size limitations. The image window will be kept from getting larger than 2x the width and height of your screen.

Just in case you're wondering why there are any size limitations: it's fairly easy to accidentally ask for a huge image to be generated. Simply crop a section of the image, zoom so you can see the individual pixels, and uncrop. If there were no size limitations, the (expanded many times) image could be *huge*, and might crash your X server. At the very least, it would take a long time to generate and transmit to your X server, and would freeze up your X server during part of it. Generally undesirable behavior. (Resource name: *2xlimit.* Type: boolean)

-nolimits

For the *truly* daring, this turns off *all* limitations on the maximum size of an image. (Well, there's still an X-imposed maximum size of 64K by 64K, but that really shouldn't be a problem.) Warning: as mentioned above, it is fairly easy to accidentally generate a huge image when you do an **UnCrop** command, and you may well crash *xv*, your X server, the host machine, or all three. Use At Your Own Risk!!! (Resource name: *nolimits* Type: boolean)

-rv

Makes *xv* display a 'negative' of the loaded image. White becomes black, and black becomes white. Color images will have 'interesting' effects, as the RGB components are individually reversed. For example, *red* (255,0,0) will become *cyan* (0,255,255), *yellow* will become *blue*, and so on.

-loadclear

If you were on a PseudoColor display, *xv* used to automatically clear the image window (and the root window, if using a root mode) whenever you loaded a new image. This was to prevent the potentially annoying/confusing 'rainbow' effect that happens when colormap entries are freed and reallocated with different colors. This has changed. By default, *xv* no longer clears the image/root window. This is for two reasons: I've decided the rainbow effect is semi-entertaining, in that it gives you something to look at while the next image is being loaded. Secondly, if you are viewing a series of images that have the same colors in them, it's possible for *xv* to animate them (by using the '-wait' command line option), albeit no faster than one frame every 1-2 seconds. For example, you can go get the satellite radar images from vmd.cso.uiuc.edu (in the directory wx), run 'xv -wait 0 -wloop SA*', and voila! Just like the evening news! (Resource name: *clearOnLoad* Type: boolean)

-DEBUG *level*

Turns on some debugging information. You shouldn't need this. If everything worked perfectly, *I* wouldn't need this. (Resource name: <none>)

-

Specifying '-' all by itself tells *xv* to take its input from *stdin*, rather than from a file. This lets you put *xv* on the end of a Unix pipe.

Section 9.9: Color Editor Resources

You can set default values for all of the HSV and RGB modification controls in the *xv color editor* window via X resources. The easiest way to explain this is with an example.

- Start *xv* and put it in the background by typing '`xv &`'.
- Type the command '`cat >foo`' in an active *xterm* window
- Bring the *xv color editor* window up.
- Issue the **Cut Resources** command.

- Click your middle mouse button in the *xterm* window. A set of resource lines describing the current *state* of the *xv color editor* controls will be 'pasted' into the window.
- You could type '<ctrl-D>' in the *xterm* to complete the *cat* command, edit this file, and put it in your .Xdefaults/.Xresources file.

The lines generated by **Cut Resources** will look like the following:

```
xv.default.huemap1: 330  30  CW 330  30  CW
xv.default.huemap2:  30  90  CW  30  90  CW
xv.default.huemap3:  90 150  CW  90 150  CW
xv.default.huemap4: 150 210  CW 150 210  CW
xv.default.huemap5: 210 270  CW 210 270  CW
xv.default.huemap6: 270 330  CW 270 330  CW
xv.default.whtmap:  0 0 1
xv.default.satval:  0
xv.default.igraf: S 4 : 0,0 : 64,64 : 192,192 : 254,254
xv.default.rgraf: S 4 : 0,0 : 64,64 : 192,192 : 254,254
xv.default.ggraf: S 4 : 0,0 : 64,64 : 192,192 : 254,254
xv.default.bgraf: S 4 : 0,0 : 64,64 : 192,192 : 254,254
```

These lines completely describe one *state* of the *xv color editor* controls. There are five different states that you can specify via X resources. The 'default' state (as shown) holds the settings used whenever the program is first started, and whenever the **Reset** command is used. You can also store settings in one of the four *xv* presets (accessed via the '1'-'4' buttons in the *xv color editor*) by changing the string 'default' in the above lines to 'preset1', 'preset2', 'preset3', or 'preset4' respectively.

There are four types of resource described in these lines: huemap, whtmap, satval, and graf.

Section 9.9.1: Huemap Resources

The huemap resources describe the state of the hue remapping dials. There are six huemap resources per state of the *xv color editor*. These huemap resources are numbered 'huemap1', 'huemap2', ... 'huemap6', and correspond to the '1'-'6' radio buttons under the hue remapping dials.

Each huemap resources takes six parameters:

1. The 'starting' angle of the *From* range, in degrees (integer).
2. The 'ending' angle of the *From* range, in degrees (integer).
3. The direction of the *From* range. Either 'cw' (clockwise) or 'ccw' (counter-clockwise).
4. The 'starting' angle of the *To* range, in degrees (integer).
5. The 'ending' angle of the *To* range, in degrees (integer).
6. The direction of the *To* range. Either 'cw' or 'ccw'.

Section 9.9.2: Whtmap Resources

The whtmap resource describes the state of the white remapping control. There is one whtmap resource per state of the *xv color editor* controls. The whtmap resource takes three parameters:

1. The hue to remap 'white' to, in degrees (integer).
2. The saturation to give to the remapped 'white', in percent (integer).
3. A boolean specifying whether the white remapping control is enabled. If '1', the control is

enabled. If ‘0’, the control is disabled.

Section 9.9.3: Satval Resource

The satval resource describes the value of the Saturation dial. There is one satval resource per state. The satval resource takes a single integer value, in the range ±100, which specifies how much to add or subtract to overall image color saturation.

Section 9.9.4: Graf Resources

The graf resources describe the state of the four ‘graph’ windows in the *xv color editor* window (Intensity, Red, Green, and Blue). The graf resources can be in one of two formats, ‘gamma’ and ‘spline/line’.

In ‘gamma’ format, the graf resource takes two parameters:

1. The letter ‘G’, specifying ‘gamma’ mode
2. A single floating point number specifying the gamma value.

In ‘spline/line’ mode, the graf resource takes a variable number of parameters:

1. The letter ‘S’ specifying ‘spline’ mode, *or* the letter ‘L’ specifying ‘line’ mode.
2. An integer number indicating the number of handles (control points) that this graph window will have. (Must be in the range 2-16, inclusive.)
3. For each handle, there will be a ‘:’, and the x and y positions of the handle, separated by a comma. The x and y positions can be in the range 0-255 inclusive.

Section 9.9.5: Other Resources

Also, there are the boolean resources ‘autoApply’, ‘displayMods’, and ‘autoReset’ which control the initial settings of the three checkboxes in the *xv color editor* window.

There are also boolean resources ‘saveNormal’, ‘pspreview’, and ‘pscompress’ which control the initial settings of the checkboxes in the *xv save* and *xv postscript* windows.

Section 10: Credits

Thanks go out to the following wonderful folks:

- First and foremost, John Hagan, friend, primary beta-tester, and driver of the Winnebago. The major difference between the *xv* that you see today, and the *xgif* of two years ago, is the years of continual harassment I've had to put up with because of alleged (read: actual) weakness in *xgif*. *XV* probably never would've been written, were it not for his input. Many of the features in the code were his idea. Centered Tiling (aka, "hagan-style tiling") is one of his.

- Thanks also go out to my auxiliary backup beta-tester, Robert Potter. He's been a source of many good ideas over the past year. Mirrored tiling was one of his.

- Helen Anderson has provided several fine ideas over the years, and also proofread this document.

- Filip Fuma, my supervisor, deserves some thanks for seeing the value of *xv*, and allowing, if not actually *encouraging*, me to write it.

- Myra VanInwegen gets a nod for cluing me in to merits of Floyd-Steinberg dithering.

- Patrick J. Naughton (naughton@wind.sun.com) provided 'gif2ras.c', a program that converts GIF files to Sun Rasterfiles. This program provided the basis for the original *xgif*, which eventually grew into *xv*. As such, it's safe to say that he "started it all." This code, slightly modified, is still in use in the module `xvgif.c`.

- Michael Maudlin (mlm@cs.cmu.edu) provided a short, understandable version of the GIF writing code. This code, essentially unmodified, is in the module `xvgifwr.c`.

- Dave Heath (heath@cs.jhu.edu) provided the Sun Rasterfile i/o support in the module `xvsunras.c`. Ken Rossman (ken@shibuya.cc.columbia.edu) fixed it up somewhat.

- Markus Baur (s_baur@iravcl.ira.uka.de) provided the interface code between the JPEG software and *xv*, that allows *xv* to read JPEG files. This module (`xvjpeg.c`) was modified by Tom Lane (Tom_Lane@g.gp.cs.cmu.edu), one of the few people who really understand the JPEG software.

- Of course, many thanks go out to Tom and all the rest of the folks in the Independent JPEG Group for providing a freely-distributable version of the JPEG software, and thereby providing the rest of us with the *new* standard graphics format (finally replacing GIF).

- Sam Leffler (sam@sgi.com) has not only come up with a freely-distributable library for doing TIFF file i/o (the *libtiff* package), but also wrote the *xv* interface modules `xvtiff.c` and `xvtiffwr.c`. Thanks Sam!

- Jef Poskanzer (jef@well.sf.ca.us) is responsible for coming up with several cool/whizo general image formats (pbm, pgm, ppm), and a package of programs for image manipulation and format conversion. Part of this code has been snarfed and incorporated into *xv* in the form of the `-best24` algorithm.

- David B. Rosen (rosen@cns.bu.edu) has written and graciously supplied the ‘fsQuick’ code, which is a very fast implementation of Floyd-Steinberg dithering for 1-bit b/w displays.

- David Elliot (dce@smsc.sony.com) gets a special thanks for being the guy who has submitted more bug fixes and feature requests than anybody else. You’d almost think he has more time to work on *xv* than I do!

The following folks have contributed to the development of *xv* in the form of bug fixes, patches, support for additional systems, and/or good ideas. See the `CHANGELOG` file for specifics:

Jimmy Aitken	jimmy@pyra.co.uk
Satoshi Asami	asami@is.s.u-tokyo.ac.jp
Markus Baur	s_baur@iravcl.ira.uka.de
Richard Bingle	bingle@cs.purdue.edu
David Boulware	dgb@landau.phys.washington.edu
Jon Brinkmann	jvb7u@astro.virginia.edu
Kevin Brown	brown@hpbsm15.boi.hp.com
Jeff Coffler	coffler@jeck.amherst.nh.us
Reg Clemens	clemens@plk.af.mil
Paul Close	pdc@lunch.wpd.sgi.com
Jan D.	jhd@irfu.se
Anthony Datri	datri@convex.com
Dean Elhard	elhard@system-m.za05.bull.com
David Elliot	dce@smsc.sony.com
Scott Erickson	sources@sherlock.ics.uci.edu
Stefan Esser	se@ikp.uni-koeln.de
Bob Finch	bob@gli.com
Peter Glassenbury	pete@cosc.canterbury.ac.nz
Robert Goodwill	robert@earth.cs.jcu.edu.au
Dave Gregorich	dtg@csula-ps.calstatela.edu
Charles Hannum	mycroft@gnu.ai.mit.edu
Dave Heath	heath@cs.jhu.edu
Mark Horstman	mh2620@sarek.sbc.com
Tetsuya Ikeda	tetsuya@is.s.u-tokyo.ac.jp
Dave Jones	jonesd@kcgl1.eng.ohio-state.edu
Kjetil Jorgensen	jorgens@lise.unit.no
Jonathan Kamens	jik@pit-manager.mit.edu
Tero Kivinen	kivinen@joker.cs.hut.fi
Rainer Klute	klute@irb.informatik.uni-dortmund.de
Marc Kossa	M.Kossa@frec.bull.fr
Bill Kucharski	kucharsk@solbourne.com
Tom Lane	tom_lane@g.gp.cs.cmu.edu
Sam Leffer	sam@sgi.com
Rolf Mayer	rz90@rz.uni-karlsruhe.de
Tom McConnel	tmcconne@sedona.intel.com
Craig McGregor	craig@csdvax.csd.unsw.edu.au
Arthur Olson	ado@elsie.nci.nih.gov
Mike Patnode	mikep@sco.com
Nigel Pearson	nigel@socs.uts.edu.au
Daniel Pommert	daniel@ux1.cso.uiuc.edu

Robert Potter	rpotter@grip.cis.upenn.edu
Eric Raymond	eric@snark.thyrsus.com
Arvind Sabharwal	arvind@brutus.ct.gmr.com
Hitoshi Saji	saji@is.s.u-tokyo.ac.jp
Mark Snitily	mark@zok.uucp
Greg Spencer	greg@longs.lance.colostate.edu
Matthew Stier	matthew@sunpix.east.sun.com
Andreas Stolcke	stolcke@icsi.Berkeley.edu
Steve Swales	steve@bat.lle.rochester.edu
Rich Thomson	rthomson@dsd.es.com
Bill Turner	bturner@cv.hp.com
Doug Washburn	washburn@hpmpea2.cup.hp.com
Drew Watson	dwatson@encore.com
Chris Weikart	weikart@prl.dec.com
Jamie Zawinski	jwz@lucid.com
Dan Zheme	dan@mordor.webo.dg.com

I'd also like to thank all the people from the GRASP Lab for serving as (unwilling) beta-testers of all the versions of *xv* you didn't see.

And finally, thanks to all the folks who've written in from hundreds of sites world-wide. You're the ones who've made *xv* a real success. Thanks!

Appendix A: Command Line Options

–	Tells *xv* to read an image from <stdin>.
–2xlimit	Allow image windows to be twice the size of the screen.
–aspect *w:h*	Sets the default ratio used by the **Aspect** command.
–best24	Use the 'best' (read: slowest) 24-bit to 8-bit color algorithm.
–bg *color*	Sets the background color.
–black *color*	Sets the 'black' color used in B/W dithering.
–browse	Keep the load window open until deliberately closed.
–bw *width*	Sets the border width of the windows.
–cecmap	Don't install image's colormap in the *xv color editor* window.
–cegeometry *geom*	Sets the initial position of the *xv color editor*.
–cemap	Automatically open the *xv color editor* on startup.
–cgeometry *geom*	Sets the initial position of the *xv controls* window.
–clear	Clears out the root window and exits.
–cmap	Automatically open the *xv controls* window on startup.
–crop	Automatically **AutoCrop** after loading an image.
–cursor *curs*	Sets the cursor used in the image window.
–DEBUG *level*	Displays debugging information.
–display *disp*	Specifies which X display to use.
–dither	Automatically dither images on initial load.
–expand *exp*	Automatically expand or contract images by the given factor.
–fg *color*	Sets the foreground color.
–fixed	Sets 'fixed aspect ratio' mode.
–geometry *geom*	Specifies initial size and position of image window.
–help	Prints a list of valid command-line options.
–hi *color*	Sets the 'highlight' color used by the buttons.
–hsv	Puts the colormap editing dials into HSV mode.
–igeometry *geom*	Specifies initial position of *xv info* window.
–imap	Automatically open *xv info* window on startup
–keeparound	Quit after deleting last image in list.
–lo *color*	Sets the 'lowlight' color used by the buttons.
–loadclear	Clear window to avoid 'rainbow' effect on PseudoColor displays.
–max	Make the image as large as possible.
–maxpect	Make the image as large as possible, preserving aspect ratio.
–mono	Display all pictures in greyscale.

-ncols *num*	Specifies maximum number of different colors to use.
-nglobal	Don't 'borrow' colors from other programs.
-ninstall	Don't 'install' colormaps. Have the WM do it for us.
-nolimits	Turn off all 'maximum size' limitations on the image.
-nopos	Don't automatically position the *xv* windows.
-noqcheck	Don't take shortcut in 24-bit to 8-bit color compression code.
-noresetroot	Don't clear root when going back to 'window' display mode.
-nostat	Speed up directory changing in load/save windows.
-owncmap	Always use and install a private colormap.
-perfect	Use and install a private colormap if necessary.
-quick24	Use the quickest 24-bit to 8-bit color algorithm.
-quit	Exit after displaying first image.
-rbg *color*	Root background color, used on some root display modes.
-rfg *color*	Root foreground color, used on some root display modes.
-rgb	Puts colormap editing dials in RGB mode.
-rmode *num*	Use specified display mode when using root window.
-root	Display images on root window.
-rv	Reverses RGB values when image is loaded.
-rw	Use read/write color cells for faster color editing.
-slow24	Use 'in-between' 24-bit to 8-bit color compression algorithm
-smooth	Automatically smooth image on initial load.
-visual *type*	Use a non-default visual of your X display.
-wait *sec*	Specifies time delay in slide show.
-white *color*	Sets the 'white' color used in B/W stippling.
-wloop	When in slide show mode, loop to start after last image.

Appendix B: X Resources

`2xlimit` *boolean*	Allow image windows to be twice the size of the screen.
`aspect` *string*	Sets the default ratio used by the **Aspect** command.
`autoApply` *boolean*	Sets initial setting of the "Auto-apply HSV/RGB mods" checkbox in the color editor window.
`autoCrop` *boolean*	Automatically do an **AutoCrop** on initial load.
`autoDither` *boolean*	Automatically dither images on initial load.
`autoReset` *boolean*	Sets the initial setting of the "Auto-reset on new image" checkbox in the color editor window.
`autoSmooth` *boolean*	Automatically smooth images on initial load.
`background` *string*	Sets the background color.
`best24` *boolean*	If true, use the 'best' 24-bit to 8-bit color algorithm.
`black` *string*	Sets the 'black' color used in B/W dithering.
`borderWidth` *int*	Sets the border width of the windows.
`browseMode` *boolean*	Keep the load window from automatically closing.
`ceditColorMap` *boolean*	Use the image's colormap in the color editor window.
`clearOnLoad` *boolean*	Clear window to avoid 'rainbow' effect on PseudoColor displays
`ctrlGeometry` *string*	Sets the initial position of the *xv controls* window.
`ctrlMap` *boolean*	Automatically open the *xv controls* window on startup.
`cursor` *int*	Sets the cursor used in the image window.
`displayMods` *boolean*	Sets the initial setting of the "Display with HSV/RGB mods" checkbox in the color editor window.
`expand` *float*	Automatically expand or contract images by factor.
`fixed` *boolean*	Sets 'fixed aspect ratio' mode.
`foreground` *string*	Sets the foreground color.
`geometry` *string*	Specifies initial position and size of image window.
`ceditGeometry` *string*	Sets the initial position of the *xv controls* window.
`ceditMap` *boolean*	Automatically open the *xv color editor* on startup.
`hsvMode` *boolean*	Puts the colormap editing dials into HSV mode.
`highlight` *string*	Sets the 'highlight' color used by the buttons.
`infoGeometry` *string*	Specifies initial position of *xv info* window.
`infoMap` *boolean*	Automatically open the *xv info* window on startup.
`keepAround` *boolean*	Don't quit after deleting last image in list. Default *true.*
`lowlight` *string*	Sets the 'lowlight' color used by the buttons.
`mono` *boolean*	Display all pictures in greyscale

`ncols` *int*	Specifies maximum numbers of different colors to use.
`nglobal` *boolean*	Don't 'borrow' colors from other programs.
`ninstall` *boolean*	Don't 'install' colormaps. Have the WM do it for us.
`nolimits` *boolean*	Turn off all 'maximum size' limitations on image.
`nopos` *boolean*	Don't automatically position the *xv* windows.
`noqcheck` *boolean*	Don't take shortcut in 24-to-8-bit color compression code.
`noresetroot` *boolean*	Don't clear root when switching to 'window' display mode.
`nostat` *boolean*	Speeds up directory access in load/save windows.
`ownCmap` *boolean*	Always use and install a private colormap.
`perfect` *boolean*	Use and install a private colormap if necessary.
`pscompress` *boolean*	Generate RLE-compressed PostScript files.
`pspreview` *boolean*	Include a b/w preview in PostScript files.
`quick24` *boolean*	If true, use the quickest 24-bit to 8-bit color algorithm.
`reverse` *boolean*	If true, reverses RGB values in the image.
`rootBackground` *string*	Root background color, used on some root display modes.
`rootForeground` *string*	Root foreground color, used on some root display modes.
`rootMode` *int*	Use specified display mode when using root window.
`rwColor` *boolean*	Use read/write color cells for faster color editing.
`saveNormal` *boolean*	If true, by default save images at 'normal' (original) size.
`slow24` *boolean*	Use alternate 24-bit to 8-bit color compression algorithm.
`visual` *string*	Use a non-default visual of your X display.
`white` *string*	Sets the 'white' color used in B/W dithering.

Appendix C: Keyboard Shortcuts

Key	Command
<**tab**>,<**space**>	**Next**
<**return**>	Load/Reload selected image.
<**bs**>,<**del**>	**Previous**
<**ctrl-D**>	**Delete**
<**ctrl-L**>	**Load**
<**ctrl-S**>	**Save**
'i'	**Info**
'e'	**ColEdit**
'q'	**Quit**
'?'	Open/Close the *xv controls* window
'c'	**Crop**
'u'	**UnCrop**
'A'	**AutoCrop**
'n'	**Normal**
'm'	**Max Size**
'M'	**Maxpect**
'>'	**DblSize**
'<'	**Half Size**
'.'	**+10%**
','	**-10%**
'4'	**4x3**
'a'	**Aspect**
'r'	**Raw**
'd'	**Dither**
's'	**Smooth**
't'	**Rotate Clockwise**
'T'	**Rotate Counter-Clockwise**
'h'	**Flip H**
'v'	**Flip V**
'R'	**Reset** (the *xv color editor*)
'p'	**Apply**
'N'	**Norm**
'H'	**HistEq**

Appendix D: RGB and HSV Colorspaces

Both the RGB and HSV Colorspaces provide a system of uniquely specifying colors via three numbers.

The RGB colorspace is the more commonly used of the two. For example, most color monitors operate on RGB inputs. In RGB colorspace, each color is represented by a three number 'triple'. The components of this triple specify, respectively, the amount of *red*, the amount of *green*, and the amount of *blue* in the color. In most computer graphics systems (and in *xv*), these values are represented as 8-bit unsigned numbers. Thus, each component has a range of 0-255, inclusive, with 0 meaning 'no output', and 255 meaning 'full output'.

The eight 'primary' colors in the RGB colorspace, and their values in the standard 8-bit unsigned range are:

Black	`(  0,  0,  0)`
Red	`(255,  0,  0)`
Green	`(  0,255,  0)`
Yellow	`(255,255,  0)`
Blue	`(  0,  0,255)`
Magenta	`(255,  0,255)`
Cyan	`(  0,255,255)`
White	`(255,255,255)`

Other colors are specified by intermediate values. For example, *orange* is chromatically between *red* and *yellow* on the color spectrum. To get an *orange*, you can simply average *red* (255,0,0) and *yellow* (255,255,0) on a component-by-component basis resulting in (255,127,0), which will be some *orange*-ish color.

You can change the brightness of the colors by raising or lowering all of their components by some factor. For example, if (0,255,255) is *cyan* (it is), then (0,128,128) would be a *dark cyan*.

Saturation of a color is a measure of how 'pure' the color is. Desaturated colors will appear 'washed-out', or pastel, whereas fully saturated colors will be 'bold', the sort of colors you'd paint a sports car. In the RGB colorspace, you can desaturate colors by adding *white* to them. For example, if you take *red* (255,0,0), and add a *medium grey* to it (128,128,128), you'll get a shade of *pink* (255,128,128). Note that the component values are 'clipped' to remain in the range 0–255.

The HSV colorspace works somewhat differently. It is considered by many to be more intuitive to use, closer to how an artist actually mixes colors.

In the HSV colorspace, each color is again determined by a three-component 'triple'. The first component, Hue, describes the basic color in terms of its angular position on a 'color wheel'. In this particular implementation, Hue is described in terms of degrees.

Unfortunately, since this document isn't printed in color, it is not possible to show this 'color wheel' in any meaningful way. Here is where the 'primary' colors live:

Red	0°
Yellow	60°

Green	120°
Cyan	180°
Blue	240°
Magenta	300°

The colors appear in the same order that they do on a standard color spectrum, except that they form a circle, with *magenta* looping back to *red*.

As with the RGB space, in-between colors are represented by in-between values. For example, *orange* would have a Hue value of 30°, being situated roughly halfway between *red* and *yellow*.

The second component of the HSV triple is Saturation, which, as described above, can be thought of as “how pure the color is”. In this implementation, saturation can range between 0 and 100, inclusive. Colors with a saturation of 100 are fully-saturated, whereas colors with a saturation of 0 are completely desaturated (in other words, *grey*).

The third component of the HSV triple is Value, which really should be called Intensity. It is a measure of how ‘bright’ the the color is. In this implementation, Value can range between 0 and 100, inclusive. A color with a Value component of 100 will be as bright as possible, and a color with a Value component of 0 will be as dark as possible (i.e., *black*).

Appendix E: Color Allocation in XV

Allocating colors on an X11 display is not as trivial a matter as it might seem on first glance. *XV* goes to a lot of trouble to allocate colors from what is essentially a scarce resource. This appendix is provided for those inquisitive types who'd be interested in learning how to successfully 'argue' with an X server.

Note: If you're using a TrueColor display, you can safely ignore this appendix, as none of the following actually happens on your system. On a TrueColor system, there is no colormap. Pixel values directly correspond to displayed color values. For example, in a common 24-bit TrueColor display, each pixel value is a 24-bit unsigned number, which corresponds to an 8-bit Red component, an 8-bit Green component, and an 8-bit Blue component, bitwise shifted and OR-ed together to form a 24-bit number. As a result, all displayable colors are *always* available for use.

Section E.1: The Problem with PseudoColor Displays

Most color X displays use a 'visual' model called PseudoColor. On a PseudoColor display, pixel values are small unsigned integers which point into a 'colormap', which contains an RGB triple for each possible pixel value. As an example, on a typical 8-bit color X display, pixel values can range between 0 and 255, inclusive. There is a 256-entry colormap which contains an RGB triple for each possible pixel value. When the video display hardware sees a pixel value of '7', for instance, it looks up color #7 in the colormap, and sends the RGB components found in that position of the colormap to the video monitor for display.

In the X Window System, entries on the display's colormap (called *colorcells*) are a scarce resource. At any time, out of the 256 colors available (in an 8-bit PseudoColor system), several of these colors may already be in use by your window manager, the cursor, and other applications. As such, *xv* cannot assume that it has 256 colors at its disposal, because it generally doesn't.

A word on the *xv* color allocation policy: The overall goal is to "make this one image being displayed right now look as good as possible, without changing the colors of any other applications." You can modify this goal slightly to suit your purposes, on the off chance that your goal isn't the same as my goal. See sections 9.4 and 9.5 for details.

Section E.2: XV's Default Color Allocation Algorithm

By default, *xv* will allocate 'read-only' colorcells. Since these colorcells cannot be changed by the application, they can be freely shared among applications. This is the default behavior because it is the most likely to succeed in getting the colors it needs. It does, however, slow down any color changes made in the *xv color editor* window. If you intend to be doing any serious color modification, you should probably run *xv* with the '`-rw`' option.

When allocating read-only colorcells, *xv* uses a four-step process to acquire the colors it wants.

The first step is to sort the desired colors by order of 'importance', so that we ask for the most 'important' colors first. See "Appendix F: The Diversity Algorithm" for more details on this step.

The next step (Phase 1 Color Allocation) is to ask for each color in the list. Colors that we failed to get (presumably because there are no more entries available in the colormap) are marked for use in the Phase 2 and Phase 3 Color Allocation steps.

If we successfully allocated all the desired colors in Phase 1, the algorithm exits at this time. Otherwise, it goes on to Phase 2. In Phase 2, the display's colormap is examined. For each color that went unallocated in Phase 1, the program looks for the color in the display's colormap that is the 'nearest' match to the originally desired color. It then tries to allocate these 'nearest' colors as read-only colorcells. The number of successful allocations in Phase 2 will be displayed in the string "Got ## 'close' colors.", visible in the *xv info* window.

If all the colors have been successfully allocated by this point, the algorithm exits. Otherwise, it continues on the Phase 3. In Phase 3, the display's colormap is once again examined. For each color that went unallocated in Phase 1 and Phase 2, the program looks for the color in the display's colormap that is the 'nearest' match to the desired color, as in Phase 2. In Phase 3, however, these 'nearest' colors are simply *used*, with no attempt made to allocate them. The number of colors 'acquired' in this step is displayed in the string "'Borrowed' ## colors.", visible in the *xv info* window.

Note that *xv* doesn't actually 'own' these colors. These colors can change without warning "out from under" the program. As such, this 'allocation' method is only good for short-term image display. You wouldn't want to use this third phase if you wanted to put up an image for long-term display, such as in the root window. In fact, whenever you display an image in the root window, *xv* automatically turns off the Phase 3 color allocation. The '`-nglobal`' option also turns the Phase 3 color allocation off.

When the Phase 3 color allocation code is turned off, any colors left unallocated after Phase 1 and Phase 2 are mapped into the 'nearest' colors that *were* allocated in Phase 1 or Phase 2. This may produce slightly worse effects than what you'd get if you used the Phase 3 code, but it does have the advantage that all the colors are 'owned' by *xv*, and therefore can't be changed by another X program.

Section E.3: 'Perfect' Color Allocation

If you'd like the image displayed "as nicely as possible on this display, and everything else be damned", you can run *xv* in 'perfect' mode, by specifying the '`-perfect`' option on the command line.

In 'perfect' mode, color allocation proceeds much like it does in 'imperfect' mode. The colors are sorted in decreasing order of 'importance'. Each of these colors is then requested, as in the Phase 1 color allocation code described above.

The big change comes on a failed allocation request. If a color is not successfully allocated in Phase 1, and this is the first failed request, we assume that the colormap is full. The program frees all the colors allocated so far, creates and installs a completely new colormap. When a new colormap is installed, everything else on the screen (including other *xv* windows) will go to hell. Only the image window will look correct. Generally, the colormap will remain installed as long as your mouse is inside the image window or the *xv color editor* window. It is, however, up to your particular window manager.

After the colormap has been installed, the program starts Phase 1 over again, allocating colors from the new, empty colormap. If any color allocation requests *still* fail, they are marked and dealt with in Phase 2. (It is possible for allocation requests from the new, empty colormap to fail, as the program may be asking for more colors than are available in a colormap. For example, you could be running *xv* on a 4- or 6-bit display, which only would have 16 or 64 colors (respectively) in a colormap.)

Phase 2 operates as described above, except that it looks for 'nearest' matches in the newly created colormap. Also, since *xv* already owns every color in this colormap, we don't technically have to 'allocate' any of them in this Phase. We already *have* allocated them once.

When you use 'perfect' mode, the Phase 3 allocation code is automatically turned off. There's no point in searching the colormap for 'near' matches and 'borrowing' these colors, as *xv* already owns all of them. Instead, any colors unallocated after the Phase 1 and Phase 2 allocation code are simply mapped into the closest colors that *were* allocated.

Note that 'perfect' mode only creates and installs a new colormap if it was necessary. If all the Phase 1 color allocation requests succeeded, a new colormap will not be created.

Section E.4: Allocating Read-Write Colors

It is sometimes desirable to allocate read-write colorcells instead of read-only colorcells. Read-write colorcells cannot be shared among programs. As such, unless you use 'perfect' mode as well, you are likely to successfully allocate fewer colors. That's the disadvantage. The advantage is that, since *xv* completely owns these colorcells, it can do what it wishes with them. Color changes (as controlled by the *xv color editor* window) will happen almost instantaneously, as the program only has to store new RGB values in the colorcells.

To allocate read-write colorcells, start *xv* with the '`-rw`' option. Colorcells are allocated one at a time. If an allocation request fails, the code stops allocating new colorcells. (Unless you've also specified 'perfect' mode. In 'perfect' mode, the first time an allocation request fails, all allocated colors are freed, a new, empty colormap is created and installed, and all colors are reallocated. If there is an allocation error in this second pass, the code stops allocating new colorcells.)

If there are still unallocated color remaining, these colors are simply mapped into the closest colors that *were* allocated.

For further information, and actual code that does everything described in this appendix, see the functions '`AllocColors()`' and '`AllocRWColors()`', both of which can be found in the source module '`xvcolor.c`'.

Appendix F: The Diversity Algorithm

The problem: You want to display an image that has *n* colors in it. You can only get *m* colors, where $m<n$. What colors do you use?

As explained in Appendix E, colors on a non-TrueColor X display are a scarce resource. You can't guarantee that you'll get as many colors as you might like. You can't even know ahead of time how many colors you *will* succeed in getting. As such, the first step of all of the color allocation algorithms (described in Appendix E) is to sort the colors in order of decreasing 'importance'. The colors are then allocated in this order, so that if the color allocation fails after *m* colors, then at least we allocated the *m* most 'important' colors.

This sorting algorithm is called the Diversity Algorithm, and is described in detail here. While the algorithms described in Appendix E are probably only of use to other X programmers (or programmers using other windowing systems with shared colormap resources), the Diversity Algorithm should be of use to anyone who has to display an image using fewer colors than they'd like to have. As far as I know, the Diversity Algorithm is an original for this program.

Section F.1: Picking the Most 'Important' Colors

There are many different criteria that one could use to define which colors in an image are 'important'.

The most naive approach would be to simply ignore the question, and just use the first *m* colors from the colormap. This is clearly unacceptable. The entries in a colormap are generally not sorted in any order whatsoever. Even when the colors *are* sorted in some order, it's not likely that it will be a useful order.

For example, in a normal greyscale picture, there is an implied colormap consisting of a continuous collection of greys, with black at the beginning, and white at the end. If a program were to only use the first few colors from this colormap, it would have several shades of 'black', but no 'whites', or even middle 'greys'.

A method of determining a color's importance to the overall picture quality is needed.

A color's 'importance' is defined by asking the question "If we can only use one of these two colors, which one would make the picture look better?". The goal is to have the picture be recognizable with very few colors, say 8 or so. Colors beyond the first few should smooth out color gradation, but should not add significant detail, nor change the color balance of the overall picture.

Picking colors in this order is not a trivial task, and is open to some degree of subjectivity. One method might involve calculating a histogram of the data to find out which colors are used the most often (i.e., which colors have the greatest number of pixels associated with them), and using those colors first. This is certainly a valid approach, but it places too much emphasis on large, uniformly colored regions, such as backgrounds. This is not generally where the 'interesting' portion of the picture is found.

For example, assume a picture that consists of a blue background, with a relatively small red square on it. Furthermore, suppose that the background isn't just one solid shade of blue, but is actually made up of three shades of blue (light blue, dark blue, and medium blue, to give them names). Finally, assume that a histogram has been computed, and light blue has been found to be the most prevalent color, followed by medium blue, dark blue, and red, in that order.

Now, attempt to display this picture using only two colors. Which two should be used? If the selection criteria is simply 'in order of decreasing usage', light blue and medium blue would be picked. However, if this is done the red square will disappear completely (red being mapped to one of the two blues).

Clearly the solution is to use red and one of the blues. Which blue, though? It could be argued that since there are three blues and only one of them can be used, middle blue should be selected, since it is the 'average' blue. Instead, the Diversity Algorithm would pick light blue, since it is used more than the others. When possible, the algorithm will try to maximize the number of pixels that are 'correct' (i.e. exactly what was asked for), rather than trying to minimize the total error of the picture. This way, additional colors smooth out gradations, rather than changing the overall color balance of the picture.

Suppose that a small yellow circle is added to the picture described above. If the problem is still 'display this picture using only two colors', then it cannot be resolved in any satisfactory method. There are no two colors that will adequately display red, yellow, and blue simultaneously . No matter what colors are used, one of the three major colors is lost. As this is now a no-win scenario, it is no longer very interesting. It doesn't matter what colors are picked, since it will look bad regardless. If, however, the problem is changed, and three colors can now be selected, it is intuitively obvious that yellow, red, and one of the blues should be selected.

So, the question is, "what is being maximized when colors are selected in this manner?" Certainly, since the blue regions are so much larger than the red and yellow regions, any rule based on the number of pixels satisfied by the color choice is irrelevant. What is being maximized is the *diversity* of the colors. By picking colors that are as *unlike* each other as possible, we wind up covering the 'inhabited' portion of the RGB color space as quickly as possible.

As a general rule, this tends to bring out the major details (such as objects) in the picture first, since the details are likely to involve contrasting colors. As more colors are picked, gaps in the RGB space are filled in. This smoothes out the color gradations, and brings out lesser detail (such as texture).

Section F.2: The Original Diversity Algorithm

The algorithm operates as follows:

1. Run a histogram on the entire picture to determine 'pixel counts' for each desired color in the colormap. Important point: throw away any colors that have a 'pixel count' of 0. These colors are never actually used in the image, and it's important that we not waste valuable colorcells allocating useless colors.

2. Pick the color with the highest pixel count. This is the 'overall' color of the picture.

3. Run through the list of un-picked colors, and find the one with the greatest 'distance' from the first color. This is the color that is most diverse from the 'overall' color.

   Note: For speed, use the 'Manhattan' distance formula:

$$d = |r1 - r2| + |g1 - g2| + |b1 - b2|$$

rather than the slower and more accurate 'geometric' distance formula:

$$d = [\,(r1 - r2)^2 + (g1 - g2)^2 + (b1 - b2)^2\,]^{1/2}$$

r1,*g1*,*b1* are the RGB components of one color, and *r2*,*g2*,*b2* are the RGB components of another color. *d* is the 'distance' between the two colors.

4. For each color remaining in the 'unpicked' list, compute the distance from it to each of the colors in the 'picked' list. Find the color in the unpicked list that is furthest from all of the colors in the picked list. Pick this color. Repeat until all colors have been picked.

Section F.3: The Modified Diversity Algorithm

Tom Lane of of the Independent JPEG Group came up with a couple of improvements to the Diversity Algorithm, resulting in the Modified Diversity Algorithm, which is what *xv* currently uses. He rightly pointed out that, on displays with an intermediate number of colors (~64), too much emphasis was being placed on getting 'different' colors, and not enough emphasis was placed on getting the 'correct' colors.

His idea was to modify the sorting criteria slightly, to better balance the allocation between diverse colors and 'popular' colors (colors with high 'pixel counts'). His solution to the problem was to alternate between picking colors based on diversity and based on popularity.

In the Modified Diversity Algorithm, as implemented in *xv*, the first color picked is the most-popular color. The second color picked is the color furthest away from the first color. The third through tenth colors picked are all picked using the normal Diversity Algorithm. The *eleventh* color picked is picked on popularity, (the un-picked color with the highest 'pixel count' is chosen). The twelfth color is once again picked on diversity. The thirteenth color is chosen on popularity, and so on, alternating, until all the colors have been picked.

It should be pointed out that there's a fair amount of subjectivity here. Tom originally had the algorithm pick colors alternately based on diversity and popularity right from the first color. (The first color picked on popularity, the second on diversity, the third on popularity, etc.) I felt that this broke the algorithm for displays with very few colors (<16), and proposed the strategy described above. (First color picked on popularity, the next ten colors picked on diversity, remaining colors alternately picked on popularity and diversity.)

Tom's other major modification to the Diversity Algorithm was to rewrite it so that 'diverse' colors are picked in $O(n^2)$ time, instead of $O(n^3)$ time.

For further information, consult the source code. (The function '`SortColors()`' in the file '`xvcolor.c`'.)

Appendix G: Adding Other Image Formats to XV

This appendix is split up into two sections, one for reading a new file format, and the other for writing a new file format. Note that you do not necessarily have to read *and* write a new file format. For example, *xv* can write PostScript files, but it can't read them.

The following instructions were written as I added PBM/PGM/PPM capability to the program, so they're likely to be fairly accurate. For example purposes, I'll be talking about the PBM/PGM/PPM code specifically. (See the file `xvpbm.c` for full details.)

Section G.1: Writing Code for Reading a New File Format

Note: Despite the wide variety of displays and file formats *xv* can deal with, internally it only manipulates 8-bit colormapped images. If you're loading an 8-bit colormapped image, such as a GIF image, no problem. If you're loading an 8-or-less-bits format that doesn't have a colormap (such as an 8-bit greyscale image, or a 1-bit B/W bitmap) your `Load()` routine will have to generate an appropriate colormap. And if you're loading a 24 bit RGB file, you'll have to compress it down to 8 bits by calling `Conv24to8()`.

Make a copy of `xvpm.c`, calling it something appropriate. I'm adding PBM capabilities, so I think `xvpbm.c` is a fine file name.

Edit the `Makefile` and/or the `Imakefile` so that your new module will be compiled. In the `Makefile`, add "`xvpbm.o`" to the "`OBJS = ...`" macro definition. In the `Imakefile`, add "`xvpbm.o`" to the end of the "`OBJS1 = ...`" macro definition.

Edit the new module.

You'll need to `#include "xv.h"`, of course.

The module should have one externally callable function that does the work of loading up the file. The function is called with two arguments, a filename and the number of colors available on this display, like so:

```
/*******************************************/
int LoadPBM(fname,nc)
    char *fname;   int nc;
/*******************************************/
```

The file name will be the complete file name (absolute, not relative to any directory). Note: if *xv* is reading from *stdin*, don't worry about it. *stdin* is always automatically copied to a temporary file. Your `Load()` routine is guaranteed that it will be reading from a real file, not a stream. This lets you use routines such as `fseek()`, and such.

The number of colors argument is either going to be 2^n, where n is the number of bitplanes on your display, or 'ncols', if specified on the command line. In either case, this number will only come into play if you have to do a 24-to-8 bit conversion. More on that later.

The `Load()` function returns ‘0’ on success, non-zero on failure.

This function is expected to load up the following global variables:

`byte *pic;`
: this is a wide*high array of bytes, one byte per pixel, starting at the top-left corner, and proceeding in scan-line order. There is no padding of any sort at the end of a scan line. The `Load()` function is expected to `malloc()` the memory for this image.

`int pWIDE, pHIGH;`
: these variables specify the size of the image that has been loaded, in pixels.

`byte r[256], g[256], b[256];`
: the desired colormap. As specified above, ‘pic’ is an 8-bits per pixel image. A given pixel value in pic maps to an RGB color through these arrays. In each array, a value of 0 means ‘off’, and a value of 255 means ‘fully on’. Note: the arrays do not have to be completely filled. Only RGB entries for pixels that actually exist in the ‘pic’ need to be set. For example, if the pic is known to be a B/W bitmap with pixel values of 0 and 1, you’d only have to set entries ‘0’ and ‘1’ of the r,g,b arrays.

`char *formatStr;`
: a short character string describing the format and size of the image. For example, “320x200 PBM”.

The function should also call ‘`SetISTR(ISTR_FORMAT, fmt, args)`’ to set the “Format:” string in the *xv info* window. It should call the function as soon as possible (i.e., once it knows the format of the picture, but before it has tried to load/parse all of the image data.) Note that the “Format:” string in the *xv info* window should be set to a somewhat more verbose version of `formatStr`. See the source code for examples.

The `Load()` function should also call ‘`SetDirRButt(F_FORMAT, ...)`’ to set the default format (in the *xv save* window) to the format of the loaded file. This, of course, is only relevant if you will also be able write files in your new format. If you aren’t planning to have a `Write()` function for this format, you won’t *have* a listing for this format in the *xv save* window.

Section G.1.1: Error Handling

Non-fatal errors in your `Load()` routine should be handled by calling `SetISTR(ISTR_WARNING, fmt, args...)`, and returning a non-zero value. The error string will be displayed in the *xv controls* and *xv info* windows.

Non-fatal errors are considered to be errors that only affect the success of loading this one image, and not the continued success of running *xv*. For instance, “can’t open file”, “premature EOF”, “garbage in file”, etc. are all non-fatal errors. On the other hand, not being able to allocate memory (unsuccessful returns from `malloc()`) is considered a *fatal* error.

Fatal errors should be handled by calling ‘`FatalError(error_string)`’. This function prints the string to *stderr*, and exits the program with an error code.

Section G.1.2: Loading 24-bit RGB Formats

If (as in the case of PPM files) your file format has 24 bits of information per pixel, you'll have to get it down to 8 bits and a colormap for *xv* to make any use of it. Conveniently, a function '`Conv24to8(pic24, w, h, nc)`' is provided, so don't worry about it.

To use it, you'll have to load your picture into a `pWIDE*pHIGH*3` array of bytes. (You'll be expected to `malloc()` this array.) This array begins at the top left corner, and proceeds in scan-line order. The first byte of the array is the red component of pixel0, followed by the green component of pixel0, followed by the blue component of pixel0, followed by the red component of pixel1, etc... There is no padding of any kind.

Once you've got this image loaded, call `Conv24to8()` with a pointer to the 24bit image, the width and height of the image, and the number of colors to 'shoot for' in the resulting 8-bit picture. This is the same parameter that was passed in to your `Load()` routine, so just pass it along.

If successful, `Conv24to8()` will return '0'. It will have created and generated the `pic` array, and filled in the `pWIDE` and `pHIGH` variables, and the `r[]`, `g[]`, `b[]` arrays. You should now `free()` the memory associated with the 24-bit version of your image and leave your `Load()` function.

Read the source in `xvpbm.c` for further info on writing the `Load()` routine.

Once you have a working `Load()` routine, you'll want to hook it up to the *xv* source.

Edit `xv.h` and add two function prototypes for any global functions you've written (presumably just `LoadPBM()` in this case). You'll need to add a full function prototype (with parameter types) in the `#ifdef __STDC__` section (near the bottom), and a function reference (just the return type) in the `#else /* non-ANSI */` section at the bottom.

Edit `xv.c`:

- Add a filetype `#define` near the top. Find the following section:

  ```
  /* file types that can be read */
  #define UNKNOWN 0
  #define GIF     1
  #define PM      2
  ```

  and add one more to the list, in this case: "`#define PBM 3`"

  Note: I only added one filetype to this list, despite the fact that I'm really adding three (or six, really) different file formats to the program (PBM, PGM, and PPM, in 'raw' and 'ascii' variations). This is because all of these file formats are related, and are handled by the same `Load()` function.

- Now tell the `openPic()` routine about your `Load()` routine:

  find the following (in openPic()):

  ```
  filetype = UNKNOWN;
  if (strncmp(magicno,"GIF87a",6)==0 ||
      strncmp(magicno,"GIF89a",6)==0) filetype = GIF;
  ```

```
else if (strncmp(magicno,"VIEW",4)==0 ||
         strncmp(magicno,"WEIV",4)==0) filetype = PM;
```

Add another 'else' case that will set `filetype` if the file appears to be in your format:

```
else if (magicno[0] == 'P' && magicno[1]>='1' &&
         magicno[1]<='6') filetype = PBM;
```

And add another case to the switch statement (a few lines further down)

```
switch (filetype) {
case GIF: i = LoadGIF(filename,ncols);  break;
case PM:  i = LoadPM(filename,ncols);   break;
}
```

add:

```
case PBM: i = LoadPBM(filename,ncols); break;
```

That should do it. Consult the files `xvpm.c` or `xvpbm.c` for further information. Remember: do as I mean, not as I say.

Section G.2: Adding Code for Writing a New File Format

Note: Despite the wide variety of displays and file formats *xv* deals with, internally it only manipulates 8-bit colormapped images. As a result, writing out 24-bit RGB images is a horrible waste (unless your format does some clever file compression), and is to be avoided if your file format can handle colortable images.

If you haven't already done so (if/when you created the `Load()` function):

- Make a copy of `xvpm.c`, calling it something appropriate. I'm adding PBM capabilities, so I think `xvpbm.c` is a fine file name.

- Edit the `Makefile` and/or the `Imakefile` so that your new module will be compiled. Add '`xvpbm.o`' to the `OBJS` macro in the `Makefile`, and to the `OBJS1` macro in the `Imakefile`.

Edit the new module.

You'll need to `#include "xv.h"`, of course.

The module should have one externally callable function that does the work of writing the file. The function is called with a virtual plethora of arguments. At a minimum, you'll be given a `FILE *` to an already open-for-writing stream, a pointer to an 8-bits per pixel image, the width and height of that image, pointers to 256-entry red, green, and blue colormaps, the number of colors actually used in the colormaps, and the 'color style' from the *xv save* window.

You may pass more parameters, since you're going to be adding the call to this function later on. For example, in my PBM code, I pass one more parameter, '`raw`' (whether to save the file as 'raw' or 'ascii') to handle two very similar formats. (Rather than having to write `WritePBMRaw()` and `WritePBMAscii()` functions.)

Your function definition should look something like this:

```
/*******************************************/
int WritePBM(fp,pic,w,h,rmap,gmap,bmap,
             numcols,colorstyle,raw)
    FILE *fp;
    byte *pic;
    int   w,h;
    byte *rmap, *gmap, *bmap;
    int   numcols, colorstyle, raw;
/*******************************************/
```

Write the function as you deem appropriate.

Some Notes:

- your function should return '0' if successful, non-zero if not
- don't close 'fp'
- pic is a w*h byte array, starting at top-left, and proceeding in normal scan-line order
- colorstyle can (currently) take on three values:

F_FULLCOLOR: This could mean either 24-bit RGB, or an 8-bit colormap or any other color format. r[pix],g[pix],b[pix] specify the color of pixel 'pix'.

F_GREYSCALE: preferably 8 bits. Two caveats: you *must* use the colormap to determine what grey value to write. For all you know, pixel value '0' in pic could map to white, '1' could map to black, and '2' could map to a half-intensity grey. You cannot make the assumption that pixel values of '0' are black, and pixel values of '255' are white.

The other note: unless the original picture was a greyscale, (which *shouldn't* be tested for), the colormap is going to have actual colors in it. You'll want to map RGB colors into greyscale values using 'the standard formula' (roughly .33R + .5G +.17B). The following code shows how to quickly write a raw greyscale image:

```
if (colorstyle == F_GREYSCALE) {
  byte rgb[256];
  for (i=0; i<numcols; i++)
    rgb[i] = MONO(rmap[i],gmap[i],bmap[i]);

  for (i=0, p=pic; i<w*h; i++, p++)
    putc(rgb[*p],fp);
}
```

F_BWDITHER: The stippling algorithm will have already been performed by the time your function is called. pic will be an image consisting of the pixel values '1' (white) and '0' (white). pic will still be organized in the same way (i.e., one byte per pixel).

Note: for F_FULLCOLOR or F_GREYSCALE images, you will be guaranteed that all pixel values in pic will be in the range [0 - numcols-1] (inclusive).

That done, edit 'xv.h' and add a pair of function declarations for your new function (one full ANSI-style prototype, and one that just declares the return type). Copy the declarations for

'WritePM()'.

Also find the block:

```
#define F_GIF      0
#define F_PM       1
```

and add another line (or two, in this case)

```
#define F_PBMRAW   2
#define F_PBMASCII 3
```

These numbers must be contiguous, as they are used as indices into the `formatRB` array.

Edit 'xvdir.c'. This is the module that controls the *xv save* window.

Add another format type to the 'formatRB' button list:

In the function 'CreateDirW()', find the block that (starts like):

```
formatRB = RBCreate(NULL,dirW,26,y,"GIF",infofg,infobg);
RBCreate(formatRB,dirW,26,y+18,"PM",infofg,infobg);
```

copy the last 'RBCreate' call in the list, add '18' to the 'y+**' argument, and stick in an appropriate format type name. In this case, I'm adding two formats (PBM raw and PBM ascii) so I'll add these two lines:

```
RBCreate(formatRB, dirW, 26, y+36,
         "PBM (raw)", infofg, infobg);
RBCreate(formatRB, dirW, 26, y+54,
         "PBM (ascii)", infofg, infobg);
```

Note: The `RBCreate()` calls must be done in the same order as the `F_GIF`, `F_PM`, etc. macros were defined in `xv.h`.

In the function `DoSave()`, find the following block:

```
switch (fmt) {
case F_GIF:
  rv = WriteGIF(fp,thepic,w, h, r, g, b,nc,col); break;
case F_PM:
  rv = WritePM (fp,thepic,w, h, r, g, b,nc,col); break;
}
```

and add cases for your function(s), like so:

```
case F_PBMRAW:
  rv = WritePBM(fp,thepic,w, h, r, g, b,nc,col,1); break;
case F_PBMASCII:
  rv = WritePBM(fp,thepic,w, h, r, g, b,nc,col,0); break;
```

That should do it!

Section G.2.1: Writing Complex Formats

If your format requires some additional information to specify how the file should be saved (such as the ‘quality’ setting in JPEG, or position/size parameters in PostScript), then your task is somewhat more difficult. You’ll have to create some sort of pop-up dialog box to get the additional information that you want.

This is not recommended for anyone who doesn’t understand Xlib programming.

The more adventurous types who wish to pursue this should take a look at the `xvjpeg.c` code, which implements an extremely simple pop-up dialog. A considerably more complicated dialog box is implemented in `xvps.c`. In addition to writing a module like these for your format, you’ll also have to add the appropriate hooks to the `DoSave()` function (in `xvdir.c`) and the `HandleEvent()` function (in `xvevent.c`). ‘`grep PS *.c`’ will be helpful in finding places where the `xvps.c` module is called.