xv

Interactive Image Display
for the X Window System

Written by John Bradley
Version 3.10

Thank you for acquiring a copy of *xv*, a pretty nifty X program. I hope you enjoy using it, as I've enjoyed writing it.

The latest version of *xv* (or at least a pointer to it) is available via anonymous ftp on `ftp.cis.upenn.edu`, in the directory pub/xv. If you're not sure if you have the latest version, or you are missing the source or documentation for *xv*, PLEASE pick up the latest version of the xv distribution. Do *not* send mail unless absolutely necessary (i.e., you don't have ftp capability).

xv Licensing Information

xv is shareware for personal use only.

You may use *xv* for your own amusement, and if you find it nifty, useful, generally cool, or of some value to you, your registration fee would be greatly appreciated. $25 is the standard registration fee, though of course, larger amounts are quite welcome. Folks who donate $40 or more can receive a printed, bound copy of the *xv* manual for no extra charge. If you want one, just ask. *Be sure to specify the version of xv that you are using!*

Commercial, government, and institutional users *must* register their copies of *xv*.

This does *not* mean that you are required to register *xv* just because you play with it on the workstation in your office. This falls under the heading of 'personal use'. If you are a systems administrator, you can put *xv* up in a public directory for your users' amusement. Again, 'personal use', albeit plural.

On the other hand, if you use *xv* in the course of doing your work, whatever your 'work' may happen to be, you *must* register your copy of *xv*. (Note: If you are a student, and you use *xv* to do classwork or research, you should get your professor/teacher/advisor to purchase an appropriate number of copies.)

xv licenses are $25 each. You should purchase one license per workstation, or one per *xv* user, whichever is the smaller number. *xv* is *not* sold on a 'number of concurrent users' basis. If *xv* was some $1000 program, yes, that would be a reasonable request, but at $25, it's not. Also, given that *xv* is completely unlocked, there is no way to enforce any 'number of concurrent users' limits, so it isn't sold that way.

Printed and bound copies of the 120-ish page *xv* manual are available for $15 each. Note that manuals are *only* sold with, at minimum, an equal number of licenses. (e.g. if you purchase 5 licenses, you can also purchase *up to* 5 copies of the manual)

The source code to the program can be had (as a compressed 'tar' file split over a couple 3.5" MS-DOS formatted floppies) for $15, for those who don't have ftp capabilities.

Orders outside the US and Canada must add an additional $5 per manual ordered to cover the additional shipping charges.

Checks, money orders, and purchase orders are accepted. Credit cards are not. All forms of payment must be payable in US Funds. Checks must be payable through a US bank (or a US branch of a non-US bank). Purchase orders for less than $50, while still accepted, are not encouraged.

All payments should be payable to 'John Bradley', and mailed to:

John Bradley
1053 Floyd Terrace
Bryn Mawr, PA 19010
USA

Site Licenses

If you are planning to purchase 10 or more licenses, site licenses are available, at a substantial discount. Site licenses let you run *xv* on any and all computing equipment at the site, for any purpose whatsoever. The site license covers the current version of *xv*, and any versions released within one year of the licensing date. You are also allowed to duplicate and distribute an unlimited number of copies of the *xv* manual, but only for use within the site. Covered versions of the software may be run in perpetuity.

Also, it should be noted that a 'site' can be defined as anything you'd like. It can be a physical location (a room, building, location, etc.), an organization (a workgroup, department, division, etc.) or any other logical grouping ("the seventeen technical writers scattered about our company", etc.).

The site license cost will be based on your estimate of the number of *xv* users or workstations at your site, whichever is the smaller number.

If you are interested in obtaining a site license, please contact the author via electronic mail or FAX (see below for details). Send information regarding your site (the name or definition of the 'site', a physical address, a fax number, and an estimate of the number of users or workstations), and we'll get a site license out to you for your examination.

Copyright Notice

The author may be contacted via:

US Mail: John Bradley
1053 Floyd Terrace
Bryn Mawr, PA 19010

FAX: (610) 520-2042

Electronic Mail regarding *xv* should be sent to one of these three addresses:

`xv@devo.dccs.upenn.edu` - general *xv* questions
`xvbiz@devo.dccs.upenn.edu` - all *xv* licensing questions
`xvtech@devo.dccs.upenn.edu` - bug reports, technical questions

Please do *not* send electronic mail directly to the author, as he gets more than enough as it is.

Table of Contents

Section 0: Release Notes

The following is a short synopsis of the major features that have been added to *xv* since Version 3.00. See the `CHANGELOG` file in the *xv* source directory for complete details, and up-to-the-minute information.

- Image cut and paste capabilities.
- Minimal image editing (line drawing, text annotation, and smudging).
- Additional choices in the **Algorithms** menu. Also, algorithms now operate on the selected region of the image, if any.
- Image padding.
- Added a **Print** command.
- Additional file formats: XPM files and 24-bit Targa files can both be read and written, as can a variety of the Amiga ILBM formats (commonly called 'IFF' files). *xv* can read (but not write) the XWD (X Window Dump) format. Also, *xv* can now read 16-bit greyscale 'pgmraw' files, however, the low-order 8-bits will be discarded.
- Much improved **Grab** command. It is now possible to grab pull-down menus in other programs.
- Much improved selection rectangle handling.
- Image 'panning' when zoomed in.
- Improved 24-to-8-bit algorithm.
- Filename completion in the *load* and *save* windows.
- Changed command-line handling. All the 'flag' arguments ('-perfect', '-smooth', etc.) are forced 'on' if you start them with a '-', and are forced 'off' if you start them with a '+'. (They used to toggle the state, which could lead to unpredictable results.)
- Support for 'gzip'ed files.
- Sniffs out the 'best' visual type to use, rather than simply using the default visual.
- New versions of Sam Leffler's *libtiff* TIFF i/o library (version 3.3) and the Independent JPEG Group's JPEG i/o library (version 5a) have been merged in.
- When viewing 24-bit images (in 24-bit mode) on an 8-bit display, *xv* now uses a 4*8*4 standard RGB colormap, rather than the 6*6*6 colormap it used in 3.00. This colormap uses fewer colors, and produces better looking results, on average.

Of course, there have been plenty of bug fixes and minor enhancements. It would probably be safe to say that there's some exciting new bugs, too!

Section 1: Overview

xv is an interactive image manipulation program for the X Window System. It can operate on images in the GIF, JPEG, TIFF, PBM, PGM, PPM, XPM, X11 bitmap, Sun Rasterfile, Targa, RLE, RGB, BMP, PCX, FITS, and PM formats on all known types of X displays. It can generate PostScript files, and if you have *ghostscript*[1] (version 2.6 or above) installed on your machine, it can also display them.

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, compress, rotate or flip the image
- crop or pad images
- view files as ASCII text or hexadecimal data
- 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
- run a number of image-processing algorithms on any rectangular portion of the image
- edit an image's colormap
- reduce the number of colors in an image
- dither in color and b/w
- crop off solid borders automatically
- convert image formats

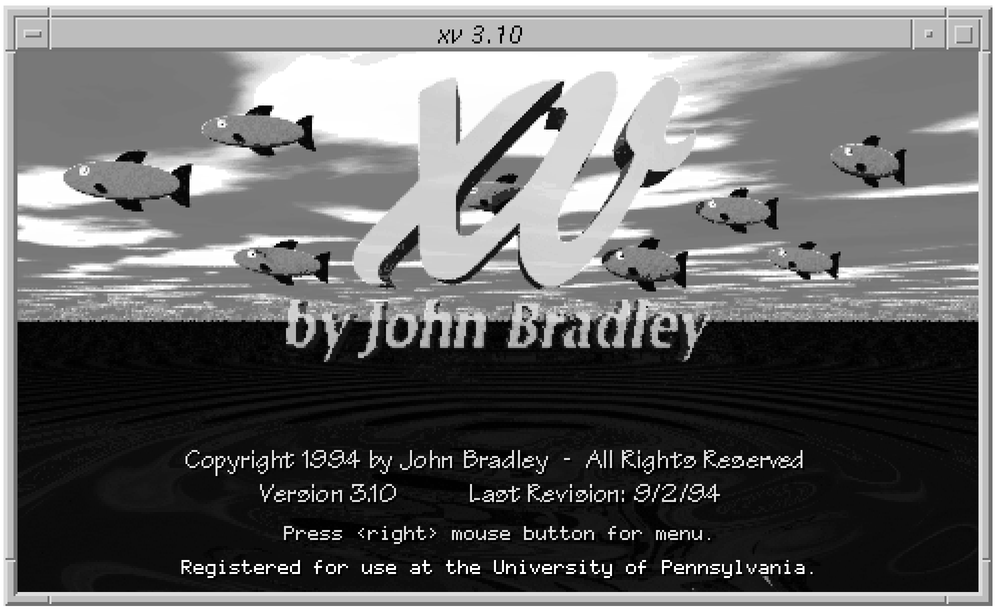
Oddly enough, I'm still having a horrible time tracking down some minor bug in the *Automatic Checkbook Balancing Module*, and once again it fails to make it into the official *xv* distribution.

[1] The *ghostscript* distribution is available via anonymous ftp from `prep.ai.mit.edu`.

Section 2: The Image Window

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 Middle mouse button inside this window will display pixel information in the following format:

```
196, 137 = 191,121,209  #bf79d1  (287 42 81 HSV)  [ 0, 0]
```

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; a 320x200 image remains a 320x200 image, regardless of what size it is displayed on the screen.

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 next item is a six-digit value expressing the RGB values of the selected pixel in hex, in the #rrggbb format. This is the same format used to specify colors to other X programs, which may be useful.

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 & HSV Colorspaces" for more information about what these numbers mean.

Finally, the last pair of numbers is the distance (dx, dy) between the current mouse position and where the mouse was originally clicked.

Note: If you actually want to measure some pixels, it will probably help to zoom in on a the relevant portion of the image, to the point that you can see the individual pixels. See “Section 2.4: Zooming” below.

Note: This also sets the ‘current color’ value used by various painting tools. (See “Section 2.6: Image Editing”.)

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 mouse button in an *xterm* window.

Section 2.2: Selection Rectangle

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

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

Double-clicking the Left button inside the image window will create a selection rectangle the size of the currently displayed area of the image, or remove the previous selection as appropriate.

Once a selection rectangle has been drawn, it can be fine-tuned by clicking and dragging any of the handles that appear on its edge. If you press the **<Shift>** key while dragging a handle, the selection rectangle is kept square. If you press the **<Ctrl>** key while dragging a handle, the selection rectangle maintains the same aspect ratio it originally had.

You can move the selection rectangle by Left-clicking inside it and dragging it around. Pressing the **<Shift>** key while doing so will constrain the motion to the horizontal and vertical axes.

You can determine how large the selection rectangle is (in image coordinates) by bringing up the *xv info* window. Do this selecting **Image Info** from the **Windows** menu *xv controls* window or by pressing the **i** key inside any open *xv* window.

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

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

it means that the current selection 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 selection rectangle precisely, you can use the arrow keys on your keyboard. First, make the *xv info* window visible as described above. Second, use the mouse to draw a rough approximation of the selection rectangle that you want. You can now use the arrow keys to move the selection rectangle around the image. Once you’ve gotten the top and left sides of the selection rectangle precisely where you want them, you can move the bottom-right corner of the selection rectangle 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.

Section 2.3: Cropping

Once you have a selection rectangle that you can live with, you can proceed with the actual cropping operation. Click the **Crop** button in the *xv controls* window, or press 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 press the **a** key in any open *xv* window. This will expand the image so that it will once again have the correct aspect ratio. (Also, see the '`-nodecor`' option, in "Section 12.9: Miscellaneous Options".)

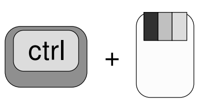
You can crop a cropped image by repeating the same steps (drawing a new selection rectangle and issuing the **Crop** command), ad infinitum.

You can return to the original, uncropped image by using the **UnCrop** command. Simply click the **UnCrop** button or press 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.

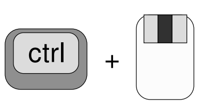
Note that if you try to make the selection rectangle too small in either width or height (under 5 screen pixels), it'll just turn itself off. If you want to crop a very small portion of an image, you'll probably have to do it in two passes. First, crop to a smallish rectangle, expand that image, then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

You can also fine-crop the image by pressing the **<Ctrl>** key and one of the arrow keys. This will crop off one pixel from the edge of the image, based on the arrow key pressed. The **Up** key moves the bottom edge up one pixel, the **Down** key moves the top edge down one pixel, and so on.

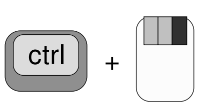
Section 2.4: Zooming



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 appear, centered around the cursor position, showing you what portion of the image will be expanded. Move the rectangle as you see fit, release the mouse button, and the region inside the rectangle will be redrawn at twice its previous 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.



While zoomed-in, you can **Pan** the image around by holding down the **<Ctrl>** key, clicking the Middle mouse button in the image window, and dragging it around. A rectangle will appear around the mouse, indicating where the currently-displayed portion of the image will be moved to.



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 2.5: Multi-Page Documents

xv now has the ability to display multi-page image files, though currently this is only implemented for PostScript files (using the *ghostscript* interpreter). If you are viewing a multi-page document, a "page *n* of *m*" string will be added to the *xv image* window's title bar.

You can walk through the document by pressing the **<PageUp>** and **<PageDown>** keys on your keyboard (they may be labeled 'Prev' and 'Next' instead) while the keyboard focus is on the *xv image* window. Pressing **<shift-Up>** and **<shift-Down>** may also work.

You can also jump directly to any given page by typing '**p**' into the *xv image* window. This will pop up a dialog window which will ask you what page you'd like to go to.

Section 2.6: Image Editing

xv now features a number of rudimentary image-editing tools. While it isn't about to compete with Adobe's *PhotoShop* anytime soon, you may still find what it does to be of some use. You now have the ability to do pixel-editing, freehand 'scribbling', line drawing, and pixel-level 'smudging'. Also, *xv* now supports cut, paste, copy, and clear of rectangular selections, as described in "Section 3.8: Image Cut and Paste Commands". Also, *xv* now has a 'text annotation' tool, described in "Section 3.10: Text Annotation".

The major weakness: the various drawing tools listed above *can only draw in colors that already exist in the image*. At this time, these tools (and *xv*) are primarily focused on the 'image editing' job, rather than on the 'arbitrary paint program' job. Needless to say, there's still a way to paint in whatever color you'd like. It's just a matter of getting the desired color(s) into the picture. The best way is probably to use the **Pad** command (see "Section 3.9: The Pad Command").

Section 2.6.1: Pixel Editing and Line Drawing

All of the following drawing commands use the concept of the 'current color'. The current color is set by using the Middle mouse button in the image window to 'pick up' the desired color.

You can edit individual pixels by doing a shift-middle-click with the mouse. (Hold a <**Shift**> key, and Middle-click somewhere in the image window.) The selected pixel will change to the desired color. It will probably help to zoom in on the relevant section of the image first. Anyway, if you drag the mouse around, you will be able to draw arbitrary 'scribbles' all over the image. This probably isn't desirable, so be careful!

You can also draw straight lines by holding down the <**Ctrl**> key along with the shift-middle combination.

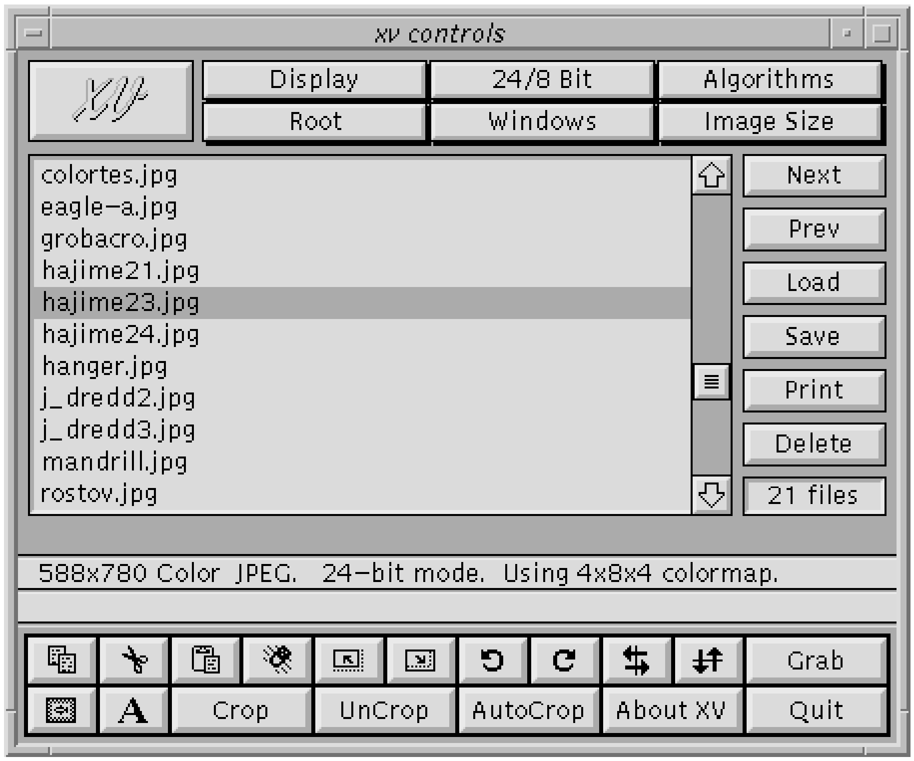
Section 2.6.2: Pixel Smudging

You can 'smudge' specific areas of the image by doing a shift-right-click with the mouse, in the image window. The clicked-on pixel will be replaced with the average color of the 9 pixels in the 3x3 square centered on said pixel. This can be used to manual edit out 'salt and pepper' noise. It can also be used to soften edges. As with the pixel editing tool, you can drag the 'smudger' around.

Note: This only works in '24-bit' mode. If you are viewing an 8-bit image, it will be automatically converted up to a 24-bit version of itself.

Note: While this tool is similar to the **Blur** and **DeSpeckle** algorithms, those algorithms operate on *every* pixel in a rectangular region, whereas this tool operates only on those pixels that you click or drag on.

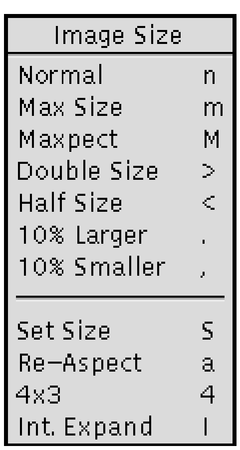
Section 3: The Control Window



The *xv controls* window is the central point of control for the program, which just goes to show how appropriately it was named. It contains 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 pressing 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



The resizing commands have all been moved to the **Image Size** menu. 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.

Note: The keyboard equivalents for each command are listed on the right side of the menu. These commands may be entered in the *xv image* window, the *xv controls* window, the *xv info* window, and the *xv color editor* window. They may *not* be entered in the *xv visual schnauzer* windows, the *xv load* window, or the *xv save* window. The same goes for all other commands listed in the *xv controls* window.

`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 2.3: Cropping" for more information.

Also, if the image is larger than the size of your screen, it will be shrunk (preserving the aspect ratio) 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 1280x823 image. (1400/900 = 1280/823)

`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 title bar, you'll find that the title bar is now off the top of the screen. To get the title bar back, simply shrink the image to anything smaller than the size of the screen. The window will be moved so that the title bar 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.

`Double Size >`

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).

`Half Size <`

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 2.3: 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 **Double Size** this image, you'll get a 264x184 image, not the 265x185 image that you started with.

`10% Larger .`

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.

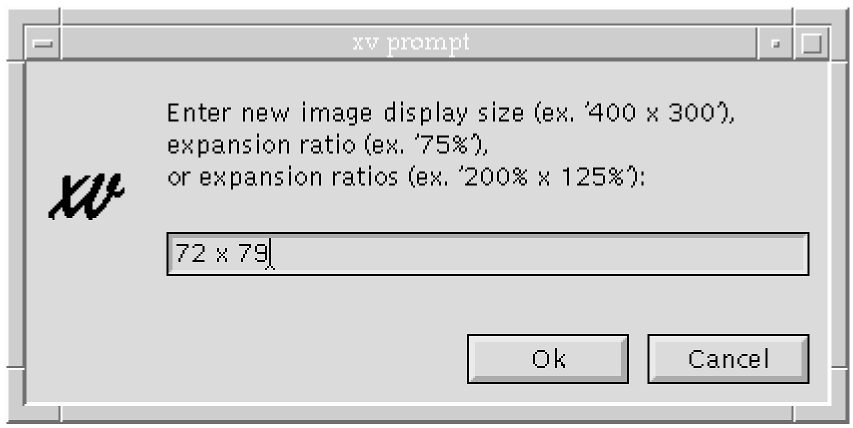
10% Smaller ,

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% Larger** and **10% Smaller** 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% Larger** and the image will be 352x220. If you issue the **10% Smaller** command now, the image will be made (352 - 35.2)x(220 - 22), or 316x198.

Set Size S

Lets you specify the exact size, or exact expansion, to display the image. Pops open a dialog box where you can type a string of the form “*width* x *height*”, “*expansion%*”, or “*horiz-expansion%* x *vert-expansion%*”. The spaces between numbers aren’t necessary, but the ‘x’ and ‘%’ characters are.



Re-Aspect a

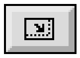
Applies the ‘default aspect ratio’ to the image. This is done automatically when the image is first loaded. Normally, the default aspect ratio is ‘1:1’, but certain GIF files may have an aspect ratio encoded in them. You can also set the default aspect ratio via a command-line argument or an X resource. See “Section 12.3: Image Sizing Options” 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.

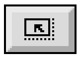
4x3 4

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).

Resizes the image to the nearest integral expansion or compression ratio. For example, if an image is currently being displayed at “162.43% x 231%”, the **IntExpnd** command will show the image at a “200% x 200%” expansion ratio. Likewise, if an image is being shown at a “37% x 70%” expansion ratio, **IntExpnd** will resize it to “33% x 50%”, the nearest integer compression ratios.

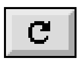


An alternate way of issuing the **10% Larger** command.



An alternate way of issuing the **10% Smaller** command.

Section 3.2: Rotate/Flip Commands



Rotates the *entire* image (not the selection) 90° clockwise.

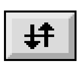
Keyboard equivalent: **t**

Rotates the *entire* image (not the selection) 90° counter-clockwise.

Keyboard Equivalent: **T**

Flips the image (or the selected area of the image) horizontally (around the vertical center-line of the image).

Keyboard Equivalent: **h**



Flips the image (or the selected area of the image) vertically (around the horizontal center-line of the image).

Keyboard Equivalent: **v**

Section 3.3: The Display Menu

This menu controls how the image is displayed on your screen. None of these commands *actually* modify the image itself, only how it is presented.

Section 3.3.1: Dithering Commands

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. This command is normally disabled, and is only enabled *after* you have issued a **Dithered** or **Smooth** command..

Dithered d

Regenerates the displayed image by dithering with the available colors in an attempt to approximate the original image. This has a useful effect *only* 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, which can help eliminate visible banding, and such. Note: If you are running *xv* on a 1-bit display the **Dither** command will be disabled, as the image will always be dithered for display.

Smooth s

Smoothes out distortion caused by integer round-off when an image is expanded or shrunk. This is generally a desirable effect, however it can be fairly time-consuming on large images, so by default it is not done automatically. See "Section 12: Modifying *xv* Behavior" for more details. Note that **Smooth** *only* has a useful effect if the image has been resized. If the image is being displayed at its normal 1:1 expansion ratio, then the **Smooth** command will not have a useful effect.

Note: if you are currently in '24-bit mode' (see "Section 3.5: The 24/8 Bit Menu" for more info), the **Dithered** command is disabled, **Raw** displays the image (dithered on an 8-bit display), and **Smooth** displays a smoothed version of the image (dithered on an 8-bit display).

Section 3.3.2: Color Allocation Commands

Read/Write Colors

When turned on, forces *xv* to use read/write color cells (ignored and disabled in **Use Std. Colormap** mode, below).. Normally, *xv* allocates read-only color cells, which allows it to share colors with other programs. If you use read/write color cells, no other program can use the colormap entries that *xv* is using, and vice-versa. The major advantage is that using read/write color cells allows the **Apply** function in the *xv color editor* window to operate much faster, and allows the **Auto-Apply while dragging** feature to be used at all. Also note that this command is only enabled if you are running *xv* on a *PseudoColor* display. See "Appendix E: Color Allocation in *xv*" for more information on display modes.

Normal Colors

xv's normal color allocation mode. For any given picture, *xv* figures out what colors should be allocated, and tries to allocate them (read-only, or read/write, as determined by the **Read/Write Colors** setting). If any color allocation fails, *xv* will try a few other tricks, and generally just map the remaining colors (the ones it didn't get) into the *closest* colors that it *did* get.

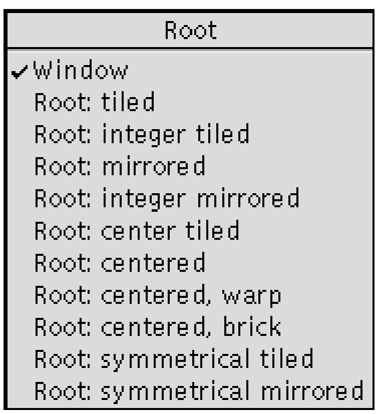
Perfect Colors

When **Perfect Colors** is turned on, *xv* proceeds as in the **Normal Colors** case. If any color allocation request fails, all colors are freed, and *xv* creates itself a private colormap, and tries all over again. It is assumed that having a private colormap will provide more colors than allocating out of the already partially-used system default colormap.

Like **Perfect Colors**, but it doesn't even *try* to allocate out of the system colormap. Instead, it starts off by creating its own colormap, and allocating from there. Slightly faster than **Perfect Colors** mode. Also useful, as certain X servers (AIX 3.1 running on an RS6000, for instance) *never* report an allocation error for read-only color cells. They just return the closest color found in the system colormap. Generally nice behavior, but it prevents **Perfect Colors** mode from ever allocating a colormap...

An entirely different color allocation mode. Instead of picking the (generally unique) colors that each image requires, this mode forces all images to be displayed (dithered) using the same set of (standard) colors. The downside is that the images don't look as nice as they do in the other modes. The upside is that you can display many images simultaneously (by running more than one copy of *xv*) without instantly running out of colors. The setting of **Read/Write Colors** is ignored while in this mode. Also, this mode is the only one available when you are displaying images in 24-bit mode.

Section 3.4: Root Display Modes



While *xv* normally displays an image in its own window, it is also possible for it to display images on the *root window* (a.k.a. 'the desktop'). You would normally use this feature from the command-line, as part of some sort of X startup script, to put up a background pattern or image of your choice. See "Section 12.6: Root Window Options" for further information on the relevant command-line options.

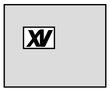
You can also specify root display options interactively, while the program is running, by using the **Root** menu, as shown.

One note regarding root display: 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, image editing, pan and zoom operations, etc. 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). There's a good reason for this. If you *could* get rid of the window, there would be no way to ever get it back (since it won't see keypresses or mouse clicks in the root window).

One other note: If you are running *xv* on certain 24-bit displays, where the 'default' visual type is an 8-bit PseudoColor, but *xv* in its cleverness has decided to use a 24-bit TrueColor mode, you will *not* be able to switch the display to a root mode. This is because *xv* requires the visual used in the 'window' mode to be the same as the visual used for the 'root' modes. In the case of these types of displays, it is not possible to put a 24-bit TrueColor image on the root window. You can get this to work by using the '`-visual default`' command-line option, which will force *xv* to use the 'default' visual for both the 'window' and 'root' modes.

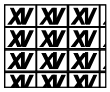
Also note: This is only necessary to get this menu to work on such displays. If you use the '`-root`' or '`-rmode`' command-line options, *xv* will automatically use patented "DoTheRightThing" technology...

Window



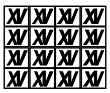
Displays the image in a window. If you were previously in a ‘root’ mode, the root window will also be cleared.

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.

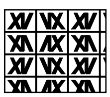
Root: integer tiled



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’.

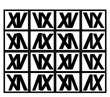
Note: using any of the ‘resizing’ commands (such as **Normal**, **Double Size**, etc.) will lose the ‘integer’-ness of the image, and you’ll have to re-select this mode to ‘integer-ify’ the image again.

Root: mirrored



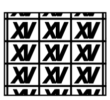
Tiles the original image with versions that have been horizontally flipped, vertically flipped, and both horizontally and vertically flipped. This gets rid of the sharp dividing lines where tiled images meet. The effect is quite interesting.

Root: integer mirrored



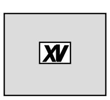
Like **Root: Mirrored**, but also does the integer-ization described under the **Root: Integer Tiled** entry.

Root: center tiled



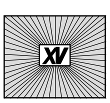
Like **Root: Tiled**, but it positions the images so that one of them is centered on the screen, and the rest are tiled off in all directions. Visually pleasing without the image size distortion associated with **Root: Integer Tiled**.

Root: centered



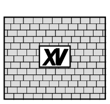
Displays a single image centered in the root window, surrounded by black, or your choice of any other solid color. (See “Section 12: Modifying *xv* Behavior” for more information.)

Root: centered, warp



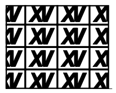
Displays a single image centered in the root window, surrounded by a black and white ‘warp’ pattern, which produces some mildly visually pleasing Moiré effects. The colors can also be chosen by the user. (See “Section 12: Modifying *xv* Behavior” for details.)

Root: centered, brick



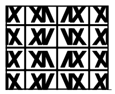
Displays a single image centered in the root window, surrounded by a black and white ‘brick’ pattern. Again, the colors can be set by the user.

Root: symmetrical tiled



Tiles images on the root window such that the images are symmetric around the horizontal and vertical center lines of the screen.

Root: symmetrical mirrored

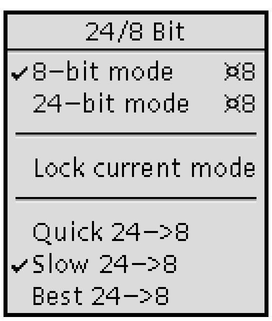


Like the **Root: symmetrical tiled** mode, but the images are also mirrored.

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.

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 run '`xv -clear`', which will clear the root window, release any allocated colors, and exit. Alternately, *xsetroot* or any other X program that puts things in the root window should be able to do the trick as well.

Section 3.5: The 24/8 Bit Menu



xv has a whopping grand total of *two* internal image formats: 8-bit colormapped, and 24-bit RGB. Every image you load is converted to one of these two formats, as part of the image loading procedure, before you ever get to see the image.

There are a few crucial differences between these two modes. In 8-bit mode, you have a colormap, and thus color-editing changes (see "Section 5: The Color Editor") can happen much more quickly. On the downside, most of the image editing functions only actually operate on 24-bit images. (Such as the **Algorithms** menu, described later in this section.) If you use these functions, *xv* will convert the 8-bit image into a 24-bit image, operate on the 24-bit data, and convert the result back to an 8-bit image. As a result, artifacts can be introduced from the image conversions. As such, if you're going to be doing a lot of this sort of thing, switch to 24-bit mode first.

But I digress...

This menu lets you see which mode *xv* is currently operating in, and lets you change modes. You can also force *xv* to remain in the current mode, and select how the program will convert 24-bit images into 8-bit images.

8-bit mode ⌘8

Forces the program into **8-bit mode** when selected. If you are currently working on a 24-bit image, it will be converted into an 8-bit image using the selected conversion algorithm (see below), and the 24-bit data will be thrown away.

24-bit mode ⌘8

Forces the program into **24-bit mode** when selected. If you currently working on an 8-bit image, it will be converted into a 24-bit image and the 8-bit image will be thrown away. Note that if you are working on a 24-bit image, switch to **8-bit mode**, and switch back to **24-bit mode**, your 24-bit data will have been lost in the conversions. A dialog box will pop up to alert you of this potential problem.

Lock current mode

Normally, *xv* will switch between 8 and 24-bit modes based on the image type (if you load a 24-bit image, it'll switch to **24-bit mode**, otherwise it will use **8-bit mode**). Turning this option on will force *xv* to remain in the current mode. One reason that you might wish to this would be to lock *xv* into **8-bit mode** so that 24-bit images are shown dithered with the 'Slow' algorithm (see below), which produces better looking images on 8-bit displays. (Just don't try to save the image afterwards!)

Quick 24->8

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 24->8

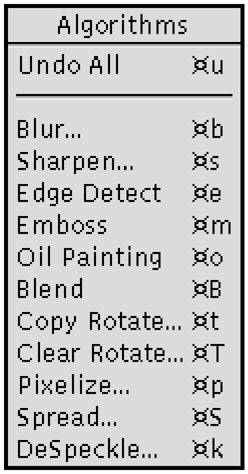
The default algorithm. Takes about twice as long as the Fast algorithm. Uses the median-cut algorithm to pick a set of 256 colors, and then dithers with these colors. It can be selected via the '-slow24' command-line option or X resource.

Best 24->8

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.

Note that none of the three 24->8 algorithm choices immediately *does* anything. They merely select which algorithm will be used the next time *xv* feels a need to convert a 24-bit image into an 8-bit image.

Section 3.6: Image Algorithms



xv now has the ability to run a number of standard image-processing algorithms on the current image, or any rectangular portion of the current image. If there is a current selection rectangle (See “Section 2.2: Selection Rectangle”), the algorithm will only affect the selected area. Otherwise, the algorithms operate on the entire image.

Algorithms are chosen via the **Algorithms** menu, and are executed immediately. Algorithms are cumulative, in that if you run an algorithm on an image, and then run a second algorithm, the second algorithm operates on the modified image produced by the first algorithm. And so on.

See “Appendix H: Adding Algorithms to *xv*” for information on how you can add additional algorithms to this menu.

Also, it should be noted that the algorithms operate on 24-bit images. If you are currently operating on an 8-bit image, and you run an algorithm, the image will be converted up to 24-bits, the algorithm will be run, and the result will be converted back to 8-bits using the currently selected 24->8 algorithm. As such, if you’re going to be doing a lot of algorithm-ing, you may find it faster to temporarily switch to **24-bit mode**. Likewise, if you intend to run multiple algorithms on the same image (say, a **Blur** followed by an **Emboss**), you should *definitely* switch to **24-bit mode**, to maintain full 24-bit precision on the intermediate images (i.e., to prevent noise from being added to the image by any intermediate 24->8 conversions).

Undo All ¤u

The (normally dimmed-out) **Undo All** selection undoes any and all algorithms that have been run on the current image. It restores the image to the state it was in when the first algorithm was executed, and it also puts *xv* back into the 8/24-bit mode it *was* in.

Keyboard Equivalent: **<Meta> u**

Blur... ¤b

Runs a convolution over each plane (red, green, blue) of the image, using a $n*n$ convolution mask consisting of all 1’s. It has the effect of, for each pixel in the image, replacing it with the average value of all the pixels in the $n*n$ rectangle centered around the pixel in question. When you select this command, a dialog box is opened up to request the value for ‘*n*’. Values must be greater than or equal to ‘3’, and they must also be odd (e.g. 3,7,5,9,11). Larger values blur the image to a greater extent.

Warning! This algorithm, due to its n^2 nature, can get very slow as *n* is increased. Values above ‘15’ aren’t particularly recommended.

Note: This command is a reasonable way to turn dithered black-and-white images back into some semblance of the greyscale that they probably originally started as.

Keyboard Equivalent: **<Meta> b**

Sharpen... ¤s

Runs an edge-sharpening algorithm on the image, which attempts to maximize contrast between adjacent pixels. Prompts for a ‘strength’ value of 0% through 99%.

Keyboard Equivalent: **<Meta> s**

Edge Detect ⌘e

Runs a convolution using a pair of convolutions, one which detects horizontal edges, and one which detects vertical edges. The convolution is done separately for each plane (red, green, blue) of the image. It is only done for pixels that have the 3x3 mask fully contained within the image, to avoid pesky edge conditions. One note: since it runs an edge detection separately for each plane of the image, the results are colorful. You'll get red edges when there are edges in the red plane, yellow edges when there are edges in the red and green planes, and so on. If you'd like a traditional grey edge detection (based on the overall intensity of each pixel), you should use the **Saturation** dial in the *xv color editor* (See "Section 5.3.3: The Saturation Control") to completely desaturate all the colors in the image (turning it grey) first. Then, the results will also be grey.

Keyboard Equivalent: **<Meta> e**

Emboss ⌘m

Runs an algorithm that produces nifty 'embossed' images by using a variation of the edge detection algorithm. It produces greyscale (8-bit) images where most of the image is shown as a medium gray. 'Leading' edges (edges found on the top and left sides) are shown as a lighter gray, and 'trailing' edges (bottom and right edges) are shown as a darker gray. The image wind up looking like pseudo-3-d, sort of like the Motif toolkit. You can then use the **White** dial in the *xv color editor* (See "Section 5.3.2: The White Remapping Control") to colorize the image as you see fit

Keyboard Equivalent: **<Meta> m**

Oil Painting ⌘o

Does an 'oil transfer', as described in the book "Beyond Photography", by Holzman. It is a sort of localized smearing. The basic algorithm is to take a rectangle (7x7 in this case) centered around the current pixel, compute a histogram of these (49) pixels, and replace the current pixel with the 'most popular' pixel as determined by the histogram.

Keyboard Equivalent: **<Meta> o**

Blend ⌘B

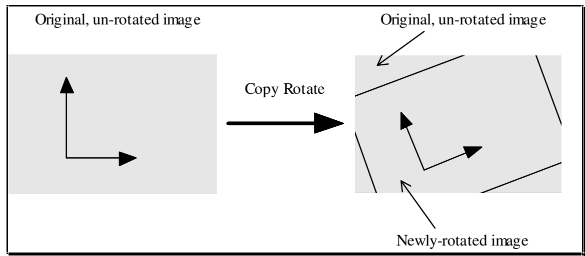
Blends a rectangular selection out of existence by replacing the selected area with a radial gradient fill. Each pixel on the edge of the selection retains its original color, the pixel in the center is replaced by the average of all the edge pixels, and all remaining pixels are replaced by the weighted average of the center pixel and the edge pixel (along the line that runs through the center pixel and the pixel being computed).

This is much easier to see than it is to explain.

Keyboard Equivalent: **<Meta> B**

Copy Rotate... ¤t

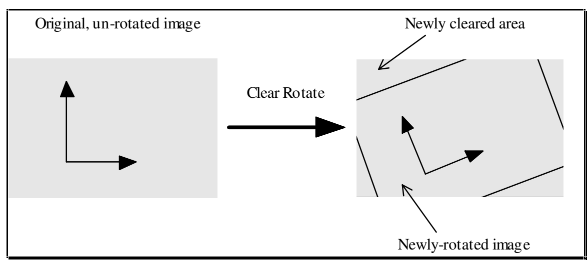
Does arbitrary (smooth) rotation of the entire image (or the selected region). You will be prompted to enter a rotation amount, in degrees. Positive values rotate the image counter-clockwise, negative values rotate the image clockwise. The **Copy Rotate** function makes a copy of the selection, rotates it, and pastes it back on top of the original, like so:



Keyboard Equivalent: **<Meta> t**

Clear Rotate... ¤T

Also does arbitrary smooth rotation, as with the **Copy Rotate** command, but this version clears the original image (using the 'current color', see "Section 2.1: Displaying Pixel Values") before pasting the rotated version, like so:



Keyboard Equivalent: **<Meta> T**

Pixelize... ¤p

Runs a 'pixelization' algorithm, suitable for obscuring sensitive and/or naughty parts of an image. Pops open a dialog box which prompts for either a single value '*m*', or a pair of values '*m*x*n*'. Divides the image (or the selected region) up into *m*-by-*m* squares (if a single value was entered) or *m*-by-*n* rectangles (if a pair of values were entered). Each area is filled with the average color of all pixels contained within the area.

Keyboard Equivalent: **<Meta> p**

Spread... ¤S

Runs a 'random spread' algorithm on the selected area of the image (or the entire image). Pops up a dialog box which prompts for either a single value '*m*', or a pair of values '*m*x*n*'. In the case of the single value, each pixel in the image is swapped with some other random pixel located within a distance of *m* pixels from the first pixel. If two values are entered, each pixel is swapped with another pixel located within $\pm m$, $\pm n$ of the original pixel. This can produce an interesting 'de-rez' effect. Whether it's useful or not is for you to decide.

Keyboard Equivalent: **<Meta> S**

Runs a despeckling algorithm, also known as a *median filter* among image-processing circles. This algorithm is good at removing minor noise from an image, such as you'll normally find on scanned-in faxes and the like. It may also prove useful for de-dithering images, turning black-and-white dithered images into greyscale images, and dithered color images into undithered color images. Note that the **Blur** algorithm can also be used in the same way, and may do a better job of un-dithering. However, the **DeSpeckle** algorithm generally leaves edges alone, unlike the **Blur** algorithm, which blurs everything uniformly.

This algorithm runs through the selected area of the image, and for each pixel in the image, looks at the *m*-by-*m* square centered around the pixel. It replaces the pixel with the median value of the pixels found in the square. As with the **Blur** algorithm, you will be prompted for a value of '*m*'. The value must not be less than '3', and it must be odd. Larger values have a greater effect, though '3' should be sufficient for most purposes.

Keyboard Equivalent: **<Meta> k**

Section 3.7: Cropping Commands

Crops the image to the current selection rectangle. This command is only available when a cropping rectangle has been drawn on the image. See "Section 2.3: Cropping" for further information.

Keyboard Equivalent: **c**

Returns the image to its normal, uncropped state. This command is only available after the image has been cropped. See "Section 2.3: Cropping" for further information.

Keyboard Equivalent: **u**

Crops off any constant-color 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. In an attempt to get around this problem, if you **AutoCrop** while in **24-bit Mode**, it will crop off portions that change by a little bit, not just portions that are exactly the same. Not that it works all that well.

Keyboard Equivalent: **C**

Also, you can 'fine-crop' an image, by holding the **<Ctrl>** key and pressing the arrow keys on your keyboard. Each press of the keys will remove one row (or column) of pixels from the appropriate edge. For example, pressing **<Ctrl><Up>** will crop off the bottom row of the image. Likewise, **<Ctrl><Down>** will crop off the top row, **<Ctrl><Left>** will crop off the right edge, and **<Ctrl><Right>** will crop off the left edge. It may sound backwards, but it does the Right Thing.

This ‘fine-cropping’ will work regardless of whether or not there is a selection rectangle currently drawn. If you accidentally crop too much off the image, you can use the **UnCrop** command described above to get the original image back.

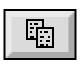
Note that cropping never actually changes the image, only what portions are displayed. The original image can always be restored using the **UnCrop** command. Also, note that the ‘cropping’ commands and the ‘zoom’ commands (See “Section 2.4: Zooming”) are related, in that both affect the ‘visible portion’ of the image. The cropping commands maintain the same expansion factors by making the window larger or smaller, while the zooming commands maintain the same window size, and stretch the relevant portion of the image to fit. As such, the **UnCrop** command will also undo an zooming that you may have done.

Section 3.8: Image Cut and Paste Commands

A new feature added to Version 3.10 is the ability to cut and paste arbitrary rectangular portions of images. You can cut (and copy) and paste within a single image, between two different images, or even between two running copies of *xv*. You can even cut or copy a portion of an image, exit *xv*, go out for dinner, come back, restart *xv*, and paste the previously-copied portion into the current image.

Copied image data is normally stored within the X Server’s memory (in the ‘`XV_CLIPBOARD`’ property), and remains there until some other data is copied, or until the X Server is reset (i.e., when you log off). This allows you to transfer image data between different copies of *xv*, and the *xv*’s can even be running on different machines.

If there is not enough server memory available to hold the copied image data (this can happen if you copy a large amount of data, and you’re using an X Terminal, as opposed to a workstation), *xv* will detect this, and fall back to using a file (‘`.xvclip`’) in your home directory. Needless to say, this precludes transferring data between *xv*’s running on different machines, but it’s better than nothing.

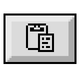


Copy. First, you must select a region to copy, via the selection rectangle mechanism (See “Section 2.2: Selection Rectangle”). Then, use this command to copy the selected data to the ‘clipboard’.

Keyboard Equivalent: **<Meta> c**

Cut. Operates just like the **Copy** command, except that it also clears the selection to the ‘current color’, which is set using the pixel measurement tool. See “Section 2.1: Displaying Pixel Values” for further information.

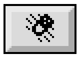
Keyboard Equivalent: **<Meta> x**



Paste. If there is no selection rectangle currently drawn, the first time you issue the **Paste** command, it will create a selection rectangle the size and shape of the image data that will be pasted. You may move and resize this rectangle as you see fit (See “Section 2.2: Selection Rectangle”). Then, when you have the selection appropriately adjusted, hit **Paste** a second time to actually paste the clipboard contents onto the image.

On the other hand, if you have a selection rectangle drawn when you hit **Paste** the first time, the image will be pasted into the selected area immediately.

Keyboard Equivalent: **<Meta> v**

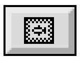


Clear. Clears the selected region to the 'current color', which is set using the pixel measurement tool. See "Section 2.1: Displaying Pixel Values".

Keyboard Equivalent: **<Meta> d**

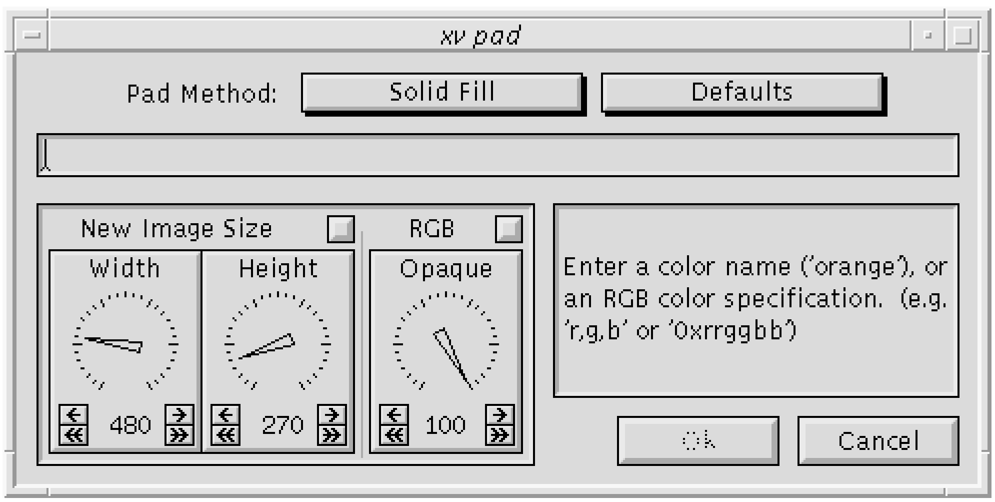
Section 3.9: The Pad Command

The **Pad** command is the opposite of the **Crop** command. Sort of. It lets you add a border of a specified size to the edges of the image. It also lets you resize images to some desired size without manually expanding or cropping the image. But it goes further than that. There are a whole slew of interesting effects available, as described below:



Pad. Brings up the dialog box shown below.

Keyboard Equivalent: **P**



Section 3.9.1: Pad Methods

There are three primary **Pad** methods, as shown to the left. These methods are accessible via the 'Pad Method' menu, which says 'Solid Fill' in the dialog box shown above. In all cases, the **Pad** command operates in the same basic way: A new image of the desired size is created, it is filled as specified, and the original image is pasted onto this new image, centered. If the new image is smaller than the original image, the original image will be cropped. Otherwise, the area outside the original image will have the new background.

Note: The new, padded image will have the same 8/24-bit mode as the original. As such, you *probably* want to switch to **24-bit Mode** *before* issuing the **Pad** command, particularly if you intend to modify the image further. (See "Section 3.5: The 24/8 Bit Menu".)

The 'Pad Method' menu gives you three different ways of filling the background:

The background is filled with a solid color. The color may be specified by name, (as specified in the X11 '`rgb.txt`' file), by RGB values in the range 0-255, separated by commas (e.g. '0,255,128'), or by RGB values in hexadecimal (e.g. '0x00ff80'). In each case, the string you enter is parsed, and the background is filled solid with the specified color.

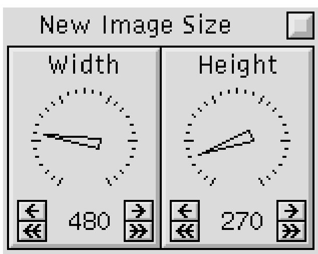
Run 'bggen'

The external program *bggen* is executed to generate the background. This program is normally distributed with *xv*, and *xv* assumes that the executable is located somewhere in your search path (such as '`/usr/local/bin/bggen`'). *bggen* can generate background gradients, at a specified angle. The gradient can be the entire size of the background, or it can repeat multiple times. See "Section 11: External Programs" for a complete description of the *bggen* command.

Load Image

An image file in any of the *xv*-supported formats may be loaded and used as the background. In this case, the background size (and therefore the size of the padded image) is equal to the size of the loaded background image. If you want a different size, you can load the background image (as a regular image), resize it as you see fit, save it, and use the newly saved version as the background for the original image.

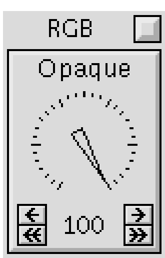
Section 3.9.2: Image Size Dials



These dials let you specify the size of the new, padded image in the "Solid" and "Run 'bggen'" modes. In the "Load Image" mode, the dials are disabled, as the new image size is dictated by the size of the loaded image.

If the image size dials are enabled, the small button above them resets them to the size of the currently-loaded image. Note that padding with the 'new' size equal to the current size won't actually *do* anything, unless you change the "Opaque" dial, described below.

Section 3.9.3: Opaque-ness



This dial sets the 'opaque-itude' of the foreground (the original image). Normally, this is set to '100%', which simply means that everywhere the original image and the background overlap, the original image is what you'll see. If you set the dial to '0%', then wherever the original image and the background overlap, the background is what you'll see. In any case, at points where the foreground image does *not* overlap the background, you'll see the background, unmodified. (Practical upshot: at '0%' you will *only* see the background.)

Of course, most of the interesting effects happen when the dial is set to intermediate values, where the foreground image will appear semi-transparent.

There are four modes the 'Opaque' dial can operate in. They are selected by clicking the button located above the 'Opaque' dial.

RGB

The two images are blended together in the most obvious way. Each RGB color component in the new image is computed by taking a weighted average of the RGB components from the appropriate pixels in the foreground and background images. Like so:

$R_{new} = (opaque\% \;*\; R_{fg}) + (\,(100 - opaque\%) \;*\; R_{bg})$ *(same for G and B)*

INT On overlapping pixels, only the *Intensity* components (the ‘V’ in the HSV colorspace, see “Appendix D: RGB & HSV Colorspaces”) of the pixels are blended, using the same weighted average calculation shown above. If you set the ‘Opaque’ value to ‘0%’, brightness of overlapping pixels will be completely determined by the ‘background’ pixel. It’s possible to get some interesting ‘masking’ effects and whatnot by playing around with black&white backgrounds.

SAT Similar to **INT**, except that it is the *Saturation* component (the ‘S’ in the HSV colorspace, see “Appendix D: RGB & HSV Colorspaces”) of the two pixels that will be blended together, again using the weighted average calculation. The effects possible with this one are considerably more subtle, and not necessarily interesting. It is mainly included for completeness.

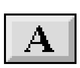
HUE In this case, the colors of the two overlapping pixels are blended. You can get some interesting effects by setting ‘Opaque’ to ‘0%’, and loading a colorful rainbow background. In this case, at all overlapping pixels, it will be as if the foreground image were a greyscale transparency, as the brightness and saturation will be controlled by the foreground, and the color will be controlled by the background.

Technical Note: The **HUE** case does *not* simply do a weighted average of the two hue components. Instead, it plots the two points on the edge of a standard color wheel, draws a straight line between them, and picks a point along this line based on the ‘Opaque’ value (i.e., the larger the value, the closer it is to the ‘foreground’ endpoint of the line). As a result, the **HUE** setting can affect *both* the hue and saturation components of the computed pixel. While this makes this ‘Opaque’ mode somewhat unlike the others, once again it is a fine example of xv’s trademark “Do The Right Thing” technology.

Section 3.10: Text Annotation

Starting in *xv* version 3.10, it is now possible to do *text annotation*, or what we in the business call “putting words on the picture”. There is one thing to note, before you go diving in: *xv*’s text annotation ability is Really Lame. I mean, *Big Time* Lame. It’s awkward to use (to say the least), and not very powerful. Frankly, it’s an embarrassment. But *maybe* it’ll be useful to someone.

Now that your expectations are suitably low, here’s how it works:



Text Annotation. First, if you have a selection rectangle drawn, get rid of it. Click this button to bring up a dialog box, where you can enter a single-line string. After you enter the string and press **<Return>** (or the **Ok** button in the dialog box), you will be presented with an appropriately-sized selection rectangle. You may reposition and resize this rectangle as you see fit. When you have the rectangle *just so*, use the **Paste** command to put the text on the image. The text will be pasted in the *current color* (See “Section 2.1: Displaying Pixel Values”). The text will be stretched to fit the selection rectangle.

Keyboard Equivalent: **A**

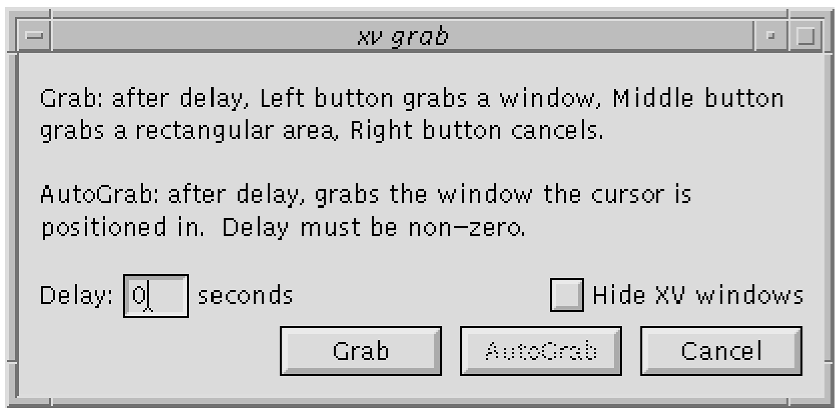
Note: if you want to change the color of the pasted text, pick a new color (via the ‘eyedropper’ described in “Section 2.1: Displaying Pixel Values”), and issue the **Text Annotation** command again. (It is not sufficient to simply **Paste** again.) Note that if you have a selection rectangle drawn, the **Text Annotation** command will not change the position or shape of the rectangle to reflect the new text, which lets you easily re-**Paste** over your last attempt.

Section 3.11: The Grab Command

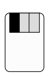
The **Grab** command in *xv* version 3.10 has been extensively improved. It is now possible to do *delayed grabs*, which can give you time to shuffle some windows around before the grab takes place. You can also do *delayed auto-grabs*, which gives you a way to grab pull-down menus and other actions that take place while a mouse-button is being held down.

Grab command. Clicking this button brings up the *xv grab* dialog box shown (and described) below.

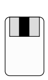
Keyboard Equivalent: **<Ctrl> g**



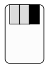
A normal grab operation (issued by hitting the **Grab** button in the *xv grab* dialog box) operates as follows:



You can grab the entire contents of a window (including its frame) by clicking the Left mouse button in a window. If you click the Left button somewhere on the root window, the entire screen will be loaded into *xv*. Note: if you Left-click somewhere inside a window whose contents are drawn in a different visual than the frame (as happens on many 24-bit graphics systems, where the default visual is an 8-bit PseudoColor, but the window contents may be drawn in 24-bit TrueColor), the window frame will *not* be included in the grabbed image. (It is not possible to grab data from two different visuals simultaneously.)



You can grab an arbitrary region of the screen by clicking the Middle mouse button and dragging a rectangle in exactly the same way you draw a cropping rectangle. When you release the mouse button, the contents of this rectangle will be read from the screen and loaded into *xv*. Note: the image will be grabbed with respect to the visual of the outermost window that completely encloses the grabbed rectangle. (Practical upshot: on 24-bit systems, if you try to grab the contents of a 24-bit window, plus some of the (8-bit) root window, window frames, etc., you will *not* get what you probably wanted. Sorry.)



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

A *delayed grab* is very similar to a normal grab, except that *xv* will wait for the specfied number of seconds, and then beep once to signify the start of the grab. You can then press the Left, Middle, or Right mouse buttons, as described above. To do a delayed grab, simply enter a non-zero number in the 'Delay' field, and press the **Grab** button in the *xv grab* dialog box.

A *delayed auto-grab* is another matter entirely. When you issue this command (by entering a non-zero number in the 'Delay' field, and pressing the **AutoGrab** button), *xv* will sleep for the specified number of seconds, and then *automatically* grab the full contents of whatever top-level window the mouse is positioned inside of. This makes it possible to grab things like pull-down menus that cannot be captured using the normal grab method. For example, to grab a pull-down menu in another program, set the 'Delay' field to something small (say, '5' seconds), hit the **AutoGrab** button, go to your other program, and open up the pull-down menu. Continue holding the menu open until *xv* beeps once (signifying the start of the grab), and then beeps twice (signifying the end of the grab). The entire top-level window containing your menu will be copied into *xv*. If you get more than you want, you can easily crop off the rest, using the cropping commands. (See "Section 2.3: Cropping")

Note: You may not be able to **Grab** all of the *xv* windows, at least not windows owned by the *xv* doing the grabbing. If you run into this minor inconvenience, simply start a second copy of *xv* and use it to grab the windows from the first *xv* (or vice-versa).

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

You can use the **Grab** command, in conjunction with *xv's* **Zoom** and **UnZoom** commands, as a reasonable, albeit overpowered and none-too-quick, 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 *xv color 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 3.12: 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, as shown below. 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 3.12.1: 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 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 **<PageUp>** and **<PageDown>** keys scroll the list up or down a page at a time. These keys may be called **<Previous>** and **<Next>** on your keyboard. You can probably also page up and down by typing **<shift>-<Up>** and **<shift>-<Down>**.
- Pressing the **<Home>** key will jump to the beginning of the list. Pressing the **<End>** key will jump to the bottom of the list. If you don't have **<Home>** and **<End>** keys on your keyboard, you may be able to emulate them by holding **<shift>** and typing the **<PageUp>** and **<PageDown>** keys.

Section 3.12.2: 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 appropriate error message.

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.

Keyboard Equivalent: **<Space>**

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.

Keyboard Equivalent: **<backspace>**

This command lets you 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.

Keyboard Equivalent: **<Ctrl> d**

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: **<Enter>**)

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**)

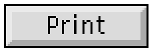
Cancel, which lets you get out of the **Delete** command without actually deleting anything. (Keyboard Equivalent: **<Esc>**

Opens the *xv load* window, which lets you load images that weren't specified on the original command-line. See "Section 8: The Load Window" for more details. Note: this function has been mostly taken over by the *visual schnauzer* (See "Section 6: The Visual Schnauzer") but remains for largely historical reasons. (It *is* considerably faster than the *schnauzer*, however...)

Keyboard Equivalent: **<Ctrl> l**

Opens the *xv save* window, which lets you save the current image (or current selected area) in a variety of image formats. See "Section 9: The Save Window" for more details.

Keyboard Equivalent: **<Ctrl> s**



Prints the current image to a PostScript[2] device.. This command takes a somewhat Unix-centric approach to printing, but it may work on other operating systems. When you issue this command, it opens a dialog box which prompts you for the 'print' command to use. Normally, this is just '`lpr`', but if you'd like to send the output to another printer, you could use something like '`lpr -Pelsewhere`'. (And right after you do that, you should change the name of that printer!) You change the default command the appears in this window by using the '`xv.print`' X resource. See "Section 12.10.5: Other Resources".

Anyway, enter some command that will have the desired effect. *xv* will open up the *xv postscript* window, where you can fine-tune the size, position, and rotation of your image. See "Section 10: The PostScript Window" for more information. When you press the **Ok** button, *xv* will generate the appropriate PostScript, and feed it as '`stdin`' to the command you entered earlier.

Keyboard Equivalent: **<Ctrl> p**

Section 3.12.3: 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 (i.e., 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, 3-d 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>`.

xv also has a 'polling mode', enabled by the '`-poll`' option. When it is turned on, *xv* will attempt to recognize when the currently displayed file changes on disk. (What with UNIX being a multi-tasking OS, it's perfectly possible to have another process modify a file while *xv* is displaying it.) When the current file changes, *xv* will reload it.

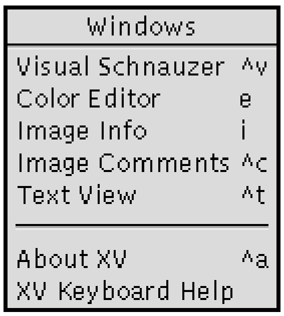
You can use this feature to get *xv* to monitor all sorts of things. For example, if you have one of those programs that automatically goes out and *ftps* the latest version of the US weather map, (and you do, in the `unsupt` directory), then you can have *xv* automatically reload the map whenever a new one is downloaded.

---

[2] It probably goes without saying that 'PostScript' is a registered trademark of Adobe Systems, Inc.

You could even use *xv* as a sort of frame buffer, to allow otherwise non-X programs to display graphics. Just have your program draw on its own internal 'frame buffer' (just an appropriately sized hunk of memory), and periodically write it out to a file in some *xv*-supported format. The PBM/PGM/PPM formats are trivial to write. See the documentation in the `doc` subdirectory of the *xv* distribution. Anyhow, periodically write the image to a file, and have *xv* display the file with the '`-poll`' option turned on. Voila! An instant output-only frame buffer for X workstations, albeit a fairly slow one.

Section 3.13: The Windows Menu



As an obvious result of *xv*'s patented "Keep-Adding-Code-Until-It-Breaks" Technology, there are considerably more commands, features, cool gizmos, and dubious design decisions than will comfortably fit in the *xv controls* window.

Luckily, this is nothing that bolting-on a couple more top-level windows won't fix, which is the method that has been used.

This is where the **Windows** menu comes into play, though in practice you'll probably just use the keyboard equivalents to open the other windows. I know *I* do.

Visual Schnauzer ^v

Opens up a *xv visual schnauzer* window (you can have up to four of them). The *visual schnauzer* lets you manipulate your image file collection in a nifty Point'n'Click, Drag'n'Drop, What You See Is What You Get, Graphical User Interface. Despite all that, it's actually a useful thing. See "Section 6: The Visual Schnauzer" for more info.

Keyboard Equivalent: **<Ctrl> v**

Color Editor e

Opens and closes the *xv color editor* window. The *color editor* provides you with an interesting collection of color manipulation doodads, including a colormap editor, hue remapping controls, arbitrary R, G, B, and I gamma correction curves, and so on. See "Section 5: The Color Editor" for more info.

Keyboard Equivalent: **e**

Image Info i

Opens and closes the *xv info* window. This window gives you detailed information about the current image, the current selection, the success/failure of the color allocation code, and such. See "Section 4: The Info Window" for more info.

Keyboard Equivalent: **I**

Image Comments ^c

Opens the *xv image comments* window. This lets you view any comments that may be encoded in the current image. It does *not* let you add or edit any comments, as that's a project for a later release. Note that, currently, the vast majority of images do *not* have comments in them. My fault, no doubt. See "Section 7.2: The Comment Window" for more details.

Keyboard Equivalent: **<Ctrl> c**

`Text View    ^t`

Opens the *xv text viewer* window. Displays the contents of the currently selected file (i.e., the file selected in the filename list in the *xv controls* window), as ASCII text, or as a hexadecimal dump. This happens automatically when *xv* tries to load a file in a format it doesn't recognize. See "Section 7: The TextView Window" for more info.

Keyboard Equivalent: **<Ctrl> t**

`About XV    ^a`

Opens the *xv text viewer* window, and displays version information, copyright notice, licensing information, the author's email addresses, and such. Basically, it's the text of the first two (or so) pages of this manual.

Keyboard Equivalent: **<Ctrl> a**

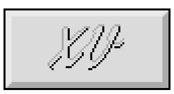
`XV Keyboard Help`

Opens the *xv text viewer* window, and displays a summary of all the keyboard equivalents and mouse button permutations.

Section 3.14: Other Commands

`About XV`

Does exactly the same thing as the **About XV** command described in "Section 3.13: The Windows Menu": displays version number, licensing info, etc.



'Reloads' and displays the *xv logo* image, which not only contains the version number and revision date of this copy, it *also* has some nifty 3-D fish.[3]

The logo image also displays who, if anyone, this particular copy of *xv* is licensed to. If it still says "UNREGISTERED COPY", you should probably do something about that.

Keyboard Equivalent: <none whatsoever>

`Quit`

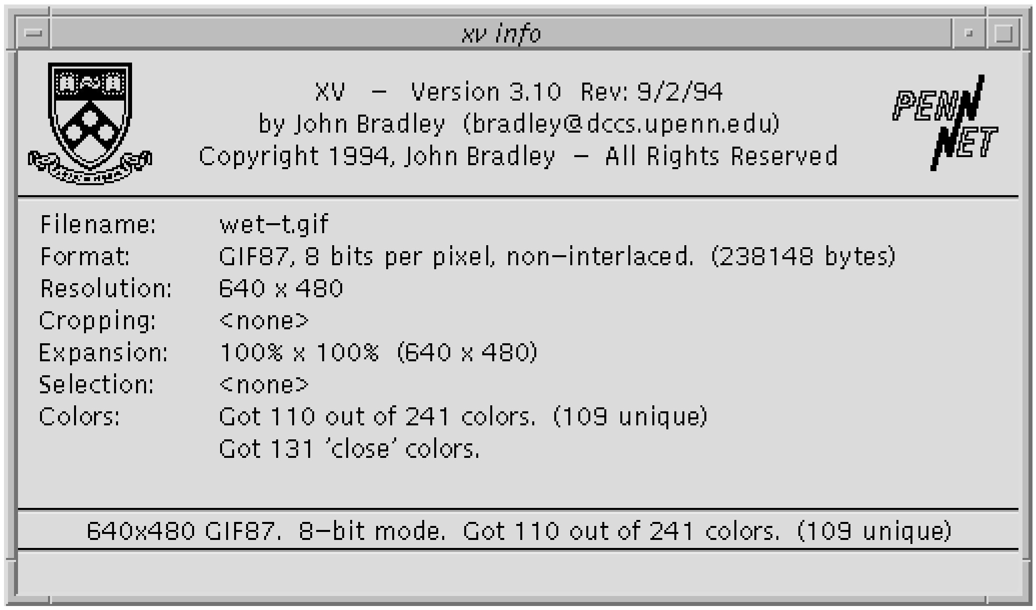
Quits *xv*. Quite possibly the most useful command of them all.

Keyboard Equivalents: **q** and **<Ctrl> q**

---

[3] Through the magic of *povray*, just in case you're interested.

Section 4: The Info Window



Section 4.1: Overview

This window displays information about the current image, color allocation, expansion, cropping, and any error messages. It can be opened by issuing the **Image Info** command. (Select **Image Info** from the **Windows** menu in the *xv controls* window, or type **i** in any open *xv* window.) You can close the window by using the **Image Info** command while the window is open. You can also close the window by clicking anywhere inside of it.

The top portion of the window displays the program name, version number, and revision date. It also shows the University of Pennsylvania shield, the PennNet 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 (i.e., you're looking at the default image, or a grabbed 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/selection activity. If you are looking at the entire (uncropped) image, this field will show "<none>". If you are viewing a cropped portion of an image, this field will display something like "247x128 rectangle starting at 132,421". See "Section 2.3: Cropping" for more details.

The “Expansion” field gives you information about how the image is displayed. It can display something like “158.00% x 137.00% (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 original image (which, in this case, had a size of 320x200).

The “Selection” field displays the size and position of the currently-drawn selection rectangle. If there isn’t a current selection, this field will show “<none>“.

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. (e.g., if ‘white’ (255,255,255) was listed twice in the image’s colormap, it would only allocate one X colorcell, but would still count as 2 successful color allocations).

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 four fields are shown (Cropping, Expansion, Selection, and Colors).

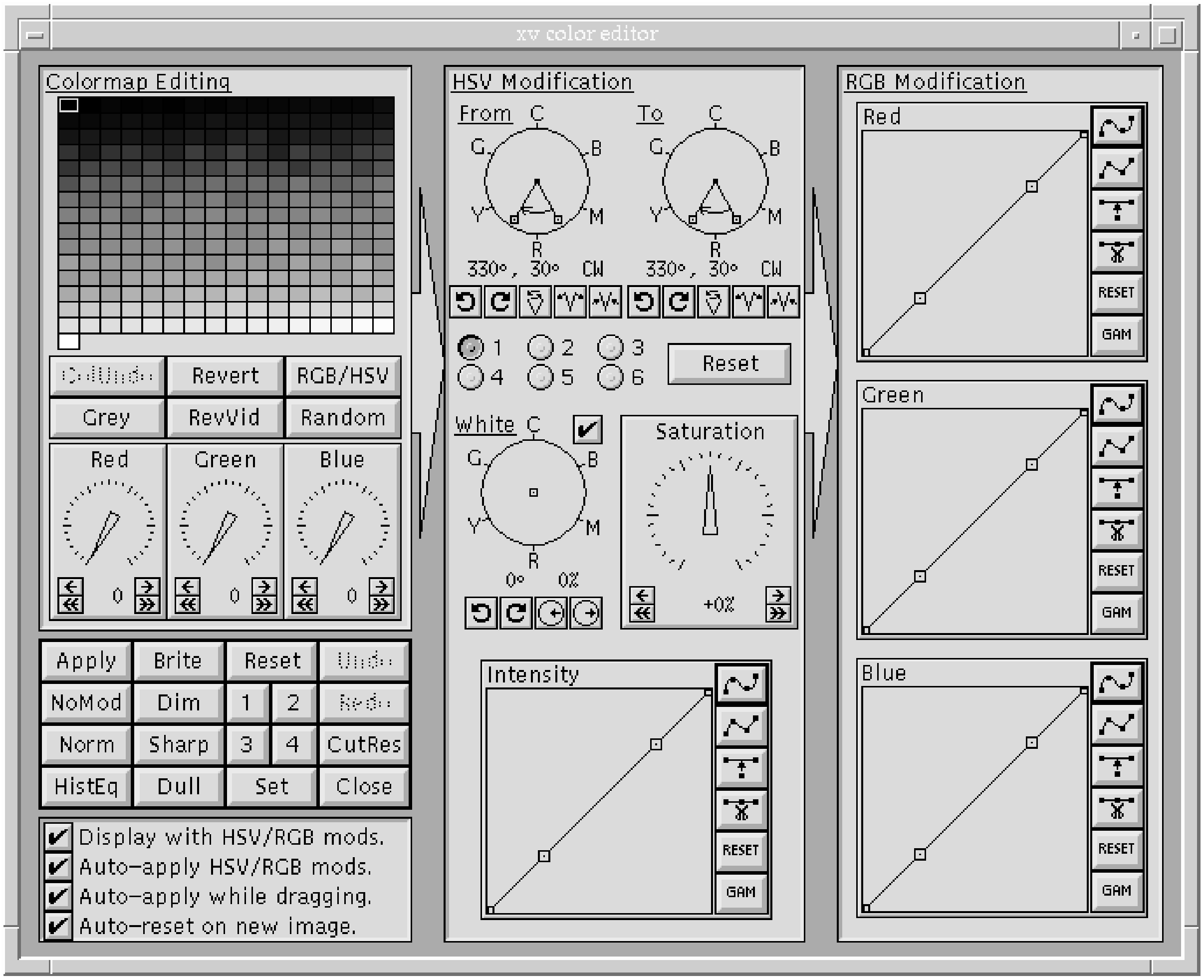
Section 4.3: Status Lines

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. Note that the *Colormap Editing* tool is *not* available when you are in **24-bit mode**.

- 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.

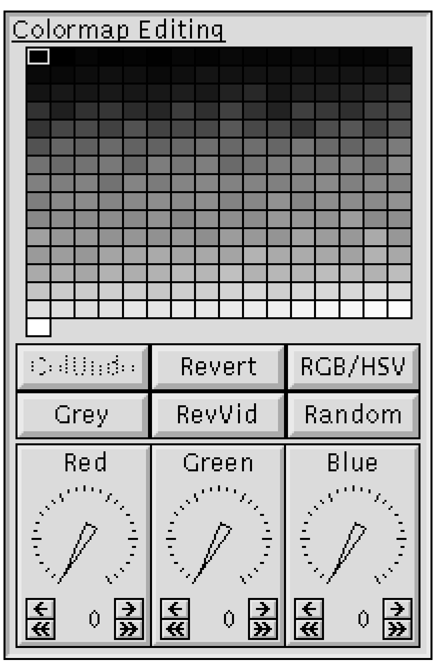
The major use of the *RGB Modification* tool is to correct for the differing color response curves of various color monitors, printers, and scanners. This is the tool to use when the image is too red, for instance.

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.

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.

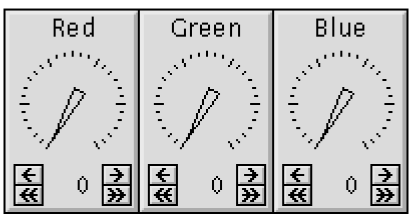
To group color cells together, do the following:

1. Hold down the **<shift>** key.
2. Left click on one color cell that you would like to be in the group
3. 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.)
4. 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

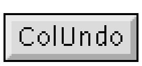


At the bottom of the *Colormap Editing* tool are three dials that let you set the color of the current color cell (or group of cells). By default, the dials control the Red, Green, and Blue components of the RGB colorspace, but they can also control the Hue, Saturation, and Value components of the HSV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



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.

Undoes all color changes. Returns the colormap to its original state. Destroys any groups that you may have created.

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.

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.

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.)

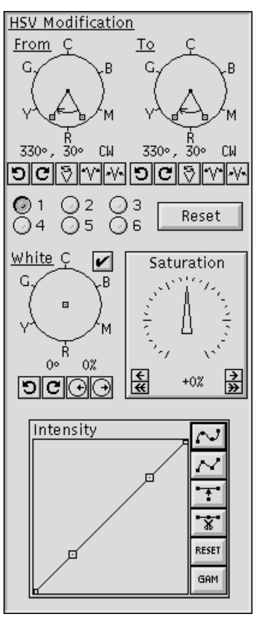
In RGB mode, each color component is separately 'inverted'. For example, Yellow (which is composed of full red, full green, and no blue) would turn to Blue (no red, no green, full blue).

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.

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 either the *HSV Modification* tool or the *RGB Modification* tool. 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. It confuses me, it'll probably confuse you, too.

Section 5.3: The HSV Modification Tools



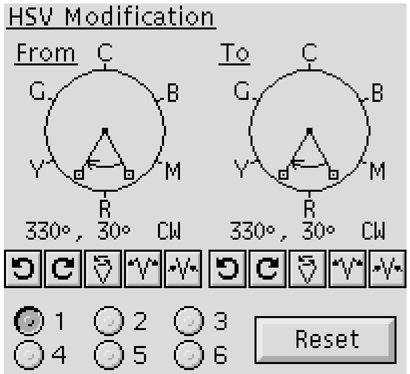
There are four separate controls in the HSV Modification tool.

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 also dial control that lets you saturate/desaturate the colors of the image.

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:

1. As a simple example of the sort of things you can do with the hue remapping control, we'll change the color of the 3-D fish in the default (xv logo) image without changing any other colors in the image. Since the fish are drawn using many different shades of pink, you would not want to do this with the *Colormap Editing* tool. It would take forever, or at least most of your lunch break.

2. Get the default image up on the screen by running *xv* without specifying any filenames on the command line. Open up the *xv color editor* window via the **Color Editor** command.

3. Next, click the mouse in the image window and drag it around. You'll find that pixels in the fish bodies have a Hue component value of '0' (which corresponds to pure red).

4. To remap this hue, simply adjust the *From* dial so that its range includes this Hue value. (It probably already does, by default.) Then twiddle the *To* dial to remap the hues. If you find more than the fish changing color, you can shrink the *From* range so it covers a smaller range of hues. This may help.

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 numbered mappings.

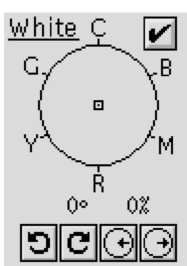
An example of multiple hue remappings:

1. 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.
2. Copy this range to the *To* range by pressing the **Reset** button.
3. 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.
4. You’ve just built yourself what is effectively a tint control.
5. Now, suppose, you’d like to adjust the sky color of your (tint-modified) image, without affecting anything else. Clicking on the sky in the image window reveals that the background still has an (original) hue of around 240. To modify this hue without affecting anything else, we’ll need a second hue remapping.
6. 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 since both the sky and the ocean are blueish, it may not be possible to change the color of one without affecting the color of the other.

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 gives you 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.

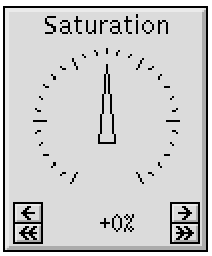
The current Hue and Saturation values provided by the control are displayed below the dial. The first number is the Hue component, in degrees, and the second is the Saturation component, as a percentage.

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 value gets down to 0%.

Example:

1. Press the **Grey** button in the **Colormap Editing** tool. This turns all the colors in the image into shades of grey.
2. 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



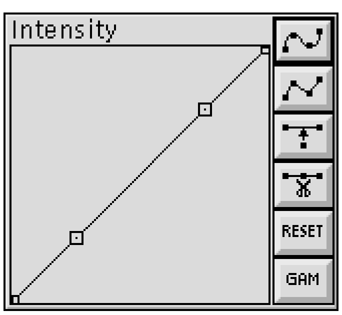
The saturation control lets you globally increase or decrease the color saturation of the image. In effect, it is much like the ‘color’ control on most color televisions.

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%’, at which point every color is fully saturated. Likewise, as values decrease, the colors become desaturated. At ‘-100%’, every color will be completely desaturated (i.e., a shade of grey). Note that this control is applied after 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. You could get around this problem by saving the (now grey) image, and reloading, or you could simply use the **Grey** button in the **Colormap Editor** instead.

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 the 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  button. The function will change to a series of line segments that connect the handles. Press the  button to go back to 'spline' mode.

The next two buttons let you add or delete handles. The  button will insert a handle into the largest 'gap' in the function. The  button will remove a handle from the smallest 'gap' in the function. You can have as few 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.

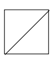
The  button puts everything back on a straight line connecting bottom-left to top-right (a 1:1 function). It does not change the number of handles, nor does it change the *x*-positions of the handles.

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

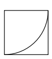
$$y = 255 \cdot (i \div 255)^{1/\gamma}$$

where *i* is the input value (0-255), γ is the gamma value, and *y* is the computed result (0-255).

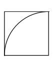
Gamma values (for our purposes) are floating point numbers that can range between -1000 and 1000, non-inclusive.



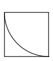
A gamma value of '1.00' results in the normal 1:1 straight line.



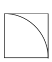
Gamma values of less than 1.00 but greater than 0.00 result in 'exponential' curves which will dim the image.



Gamma values greater than 1.00 result in 'logarithmic' curves, which will brighten the image. Try it and see.



Gamma values between 0 and -1 result in reversed 'exponential' curves.



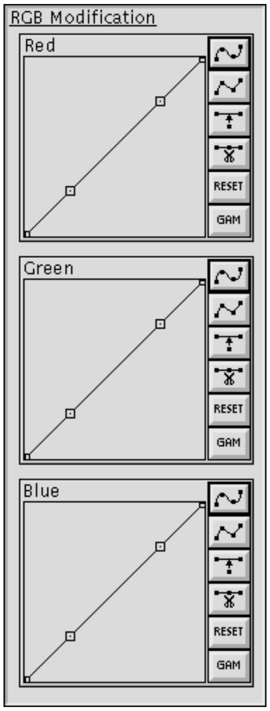
Gamma values less than -1 result in reversed 'logarithmic' curves.

There is a shortcut for the  button. Type **g** while the mouse is inside the graph window.

Also, touching any of the handles after a  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



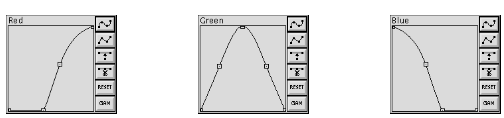
The *RGB Modification* tool is a collection of three graph windows, each of which operate on one of the components of the RGB colorspace. This tool lets you perform global color-correction on the image by boosting or cutting the values of one or more of the RGB color components. You can use this to correct 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:

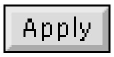
1. Adjust the *Red* graph so that there is a strong red presence on the right side of the graph, and none on the left, or in the middle.
2. Adjust the *Green* graph so that there is a strong green presence in the middle of the graph, and none on the left or right.
3. Adjust the *Blue* graph so that there is a strong blue presence on the left side of the graph, and none on the left, or in the middle.
4. 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!

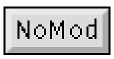
Section 5.5: The Color Editor Controls

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.



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.

Keyboard Equivalent: **<Meta> a**



Displays the image without any HSV or RGB Modifications. Also turns the '**Display with HSV/RGB mods**' checkbox off.

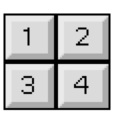
Undoes the last change to the HSV or RGB controls.

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.

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.

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.

Keyboard Equivalents: **R**, **<Meta> r**, and **<Meta> 0**



Pressing any of these buttons recalls a preset (a complete set of values for the HSV and RGB controls). By default, the presets are:

1. Binary intensity. Every color in the image is either at full brightness, or black. Gives images a neat 'neon' sort of look, much like the *Saturday Night Live* credits of the late-70's.

2. Binary colors. The image will be shown using only the 8 binary combinations of red, green, and blue. (e.g. black, blue, green, cyan, red, magenta, yellow, white)

3. Temperature pseudo-color. (For use on greyscale images) Maps intensity values 0-255 into a 'temperature' color scheme where blue is 'coldest' and red is 'hottest'.

4. Map pseudo-color. (For use on greyscale images) Maps intensity values 0-255 into something akin to the standard 'elevation map' color scheme (blue, green, yellow, brown)

Of course, you can replace these defaults with your own. See "Section 12: Modifying xv Behavior" for more details.

Keyboard Equivalents: **<Meta> 1**, **<Meta> 2**, **<Meta> 3**, and **<Meta> 4**

Set

Used in conjunction with the **Reset**, **1**, **2**, **3**, or **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**, or **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 12: 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 12: Modifying xv Behavior" for more details.

Close

This button closes the *xv color editor* 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.

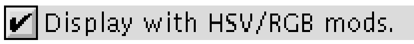
Norm

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.

Keyboard Equivalent: **N**

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.

Keyboard Equivalent: **H**



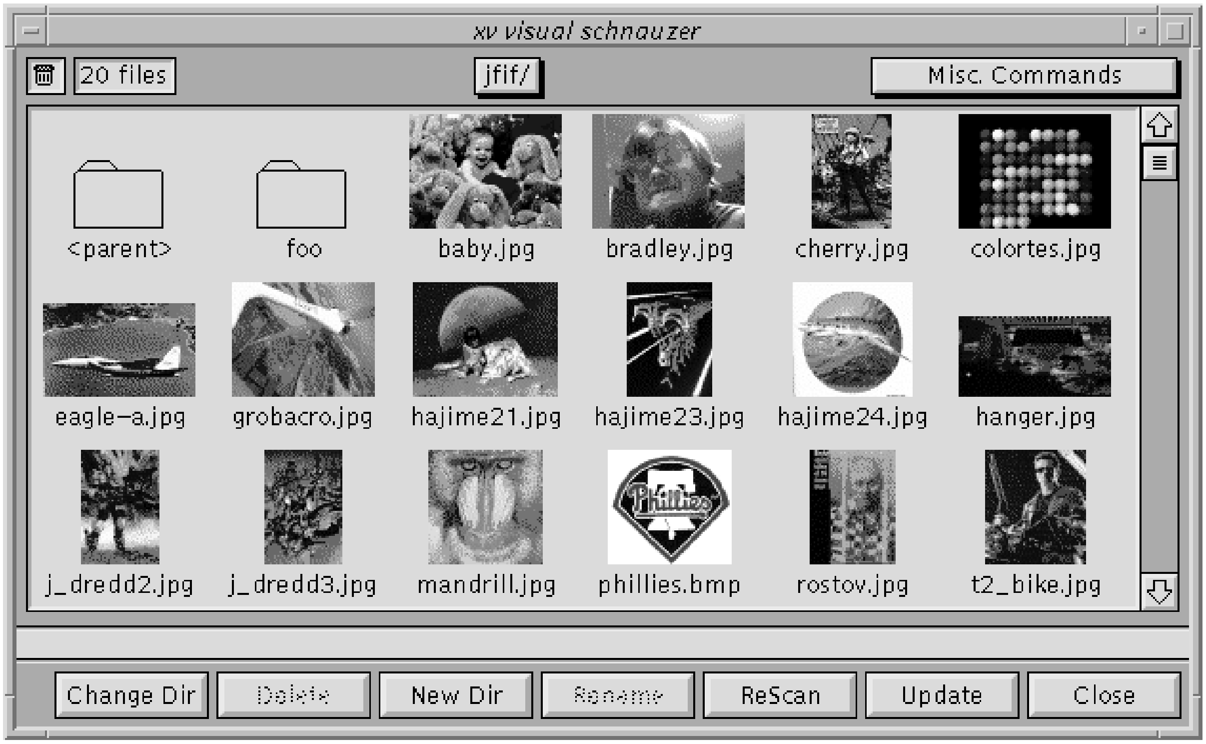
The '**Display with HSV/RGB mods**' checkbox tells you whether or you're looking at a modified image (checked) or the 'raw', unmodified image (unchecked). The **Apply** and **NoMod** buttons change the setting of this checkbox, and you can also change the checkbox directly by clicking on it.

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, if 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-apply while dragging**' checkbox controls whether or not the image colors are changed automatically as you manipulate the various *xv color editor* dials and graphs. This button is normally turned on, but for it to have any effect, you must be in '**Read/Write Colors**' mode. See "Section 3.3.2: Color Allocation Commands" and the '`-rw`' mode described in "Section 12.4: Color Allocation Options" for more information.

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 Visual Schnauzer



Section 6.1: What's a Visual Schnauzer?

Hint: it's not a smallish seeing-eye dog of Germanic descent.

Probably the most exciting new feature in Version 3.00 was the *Visual Schnauzer*, and while it hasn't changed dramatically in Version 3.10, it's still plenty exciting. The *Visual Schnauzer* gives you a visual interface to the UNIX file system. It lets you view tiny 'thumbnail' representations of your image files, directories, and other files. It lets you create directories, rename files, move and copy files to different directories, delete files, view image files, view text files, and just generally nose around.

In short, it's cool, and it should be of some use for those of us who have large numbers of GIF and JPEG files with non-descriptive MS-DOS-*style* '8+3' filenames.

Note: Unlike most *xv* windows, schnauzer windows are resizable. You can decide how many (or few) icons you'd like displayed by changing the size of the schnauzer window. It also should be pointed out that there are more command buttons available than are shown at the default schnauzer size. If you make the window larger, you will be given additional command buttons. *All* the commands are always available from the **Misc. Commands** menu.

Section 6.2: Operating the Schnauzer

Click on the **Visual Schnauzer** button in the *xv controls* window, or type a **<Ctrl> v** inside any active *xv* window. This will open a schnauzer window. The window will display the contents of the directory that you were in when you started *xv*. (Though this behavior, like so many things, can be overridden. See the description of the '`-dir`' command-line option in "Section 12.9: Miscellaneous Options".)

The first time you open an image directory with the schnauzer, you'll find that your image files are *not* displayed as spiffy little thumbnail icons, but instead are shown as generic icons showing the file format used. This is because the icons need to be generated from the image files. Displaying a thumbnail icon would normally require loading the entire image, compressing it down to a small 80x60-*ish* image, and dithering it with some set of colors. This is a very time-consuming operation. To avoid doing this every single time you open a schnauzer window, *xv* lets you do this operation once, and it maintains the results in a hidden subdirectory ('`.xvpics`') for each directory. This subdirectory contains one small (2k) file for each file in the parent directory. The icons are in an 8-bit per pixel format, pre-dithered with a 332 RGB colormap.

So, assuming that there are some image files in the current directory, the first thing you should do is generate some icons. (If there aren't any image files in the current directory, exit *xv* and start it again from a directory that *does* have image files. We'll discuss navigating via the schnauzer later.)

Section 6.2.1: Generating Image Icons

To generate icons for all the images in the current directory, issue the **select All files** command from the **Misc. Commands** menu. All the icons in the current directory should 'light up' in some way to signify that they are selected, with the possible exception of the `<parent>` directory.

Next, issue the **Generate icons** command from the **Misc. Commands** menu. *xv* will begin the process of generating icon files for the selected files. This process can be quite time-consuming, particularly if you have a large number of files in the directory, or the image files are in the JPEG format. During this time, *xv* is effectively 'out-to-lunch', as it will not be paying attention to any X events. There isn't any way to stop it either, though if you *kill* the *xv* process from another window, you will succeed in stopping the icon generation. You will *not* lose any computed icons by doing this, as icon files are written out as they are computed, so it's perfectly safe to stop the generation via this method.

Also, you'll note that *xv* displays a bar-graph 'progress meter' at the top of the schnauzer window while it's doing this, so you can gauge roughly how long it will take. Icons will be displayed as they are generated, in an attempt to make the process as entertaining as possible.

Section 6.2.2: Changing Directories

The name of the current directory is displayed in a button at the top-center of the schnauzer window. You can change to any higher-level directory in the current path by pulling down this menu button. The names of the parent directories will be shown in the pull down menu.

You can also go up one level by double-clicking on the `<parent>` directory icon.

You can go down to any subdirectory by double-clicking on the appropriate subdirectory icon.

The **Change Directory** command gives you a more direct route: you just type the pathname of the directory you'd like to go to.

Section 6.2.3: Scrolling the Schnauzer

You can scroll the schnauzer by operating the vertical scroll bar, which operates exactly as described in "Section 3.12.1: Operating a List Window".

You can also scroll the window a page at a time by using the **<PageUp>** and **<PageDown>** keys on your keyboard (which may be labeled **<Prev>** and **<Next>** on some keyboards). The **<Home>** and **<End>** keys will take you to the beginning and end of the list, respectively. (If your keyboard doesn't have Home and End keys, **<Shift> <PageUp>** and **<Shift> <PageDown>** may also work.) You can scroll the list a line at a time by using the up and down arrow keys. Note, however, that the arrow keys move the indicator that shows the currently selected file. Likewise, the left and right arrow keys move the selection indicator left and right, wrapping at the edges of the window.

Also, if you type a normal, alphabetic character into the schnauzer window, it will attempt to scroll the window so that the first file beginning with that character will be visible. Note that this behavior keeps you from typing normal *xv* keyboard equivalents into this window, so don't even try it.

Section 6.2.4: Selecting Files

Double-click on an image file's icon. The image should be loaded, and the full path name of the image should be added to the bottom of the *xv controls* window's name list. You can also double-click on text files to view them in a TextView window. (See "Section 7: The TextView Window" for more information.)

It is possible to select more than one file at a time. The easiest way to do so is to hold down the **<Shift>** key and click on icons. When the **<Shift>** key is held down, any icons clicked on are simply toggled between being selected, and being *not* selected. All other files remain as they were. If you type **<Shift> <Space>**, *xv* will keep all currently 'lit' files lit, and light (and load) the file *after* the last lit file it finds. This can be useful for building up a big list of files to delete by **<Shift> <Space>**-ing your way through the list, and manually unselecting any files you want to keep.

You can also select files by drawing a rectangle around icons that you want selected. To do so, simply click in the space in between the icons, and drag. A rectangle will be drawn, and all icons that are inside (or partially inside) the rectangle will be selected. Note that if you drag the rectangle off the top or bottom of the window, the window will automatically scroll. Also, if you hold down the **<Shift>** key while you draw a rectangle, any previously selected icons will continue to remain lit.

Finally, you can also select files by using the **select All files** command from the menu (or by typing a **<Ctrl> a**) to select all the icons in the current directory. If you want *most* of the files selected, it might be easiest to select all the files, hold down the **<Shift>** key, and selectively turn off the ones you don't want selected.

If you double-click on a selected file while there are multiple files selected, the clicked-on file will be displayed, and *all* selected filenames will be copied into the *xv controls* window's list.

Section 6.2.5: File Management

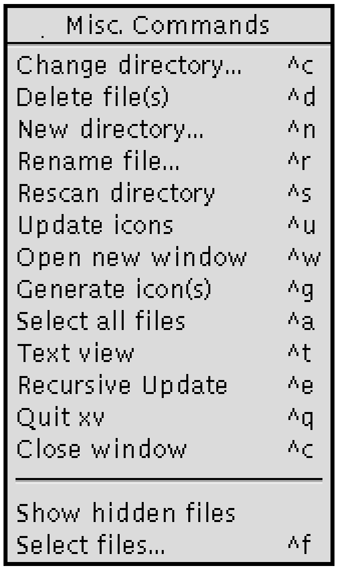
Once you have some files selected, you can move them to other directories by clicking on a lit icon and dragging it into one of the subdirectories. When you press the mouse button, the cursor will change to a 'files' cursor to indicate that you are potentially moving files around. If you move the cursor near the top or bottom edge of the window, the window will scroll up or down accordingly. Necessary, as the subdirectory folders are always shown at the top of the window. When you move the cursor on top of a folder (either the `<parent>` directory, or one of the subdirectories), the folder will light up to acknowledge your selection. If you release the mouse button while a folder is lit, all currently selected files (and folders) will be moved into the lit folder.

Note that, using only one schnauzer window, it's only possible to move files up or down one directory level. If you want to move files to completely unrelated directories, you can easily accomplish this by opening a second schnauzer window (via the **open new Window** command in the menu, or **<Ctrl> w**). Get the second schnauzer window aimed at the desired destination directory, and simply drag the files from the first window into the second window.

Also, when you drag files around, *xv* normally 'moves' the files from one directory to another. If you'd like, you can copy files instead of moving them. Just press (and hold down) the **<Ctrl>** key while dragging the files. The 'files' cursor will change to display 'CPY' as well, to signify that you are copying files instead of moving them.

The remaining file-management commands (renaming files, deleting files, and creating subdirectories) are discussed below.

Section 6.3: The Commands



There are more commands available in the schnauzer than are normally displayed in the buttons at the bottom of the window. The additional buttons will be displayed if you make the window larger. Alternately, all commands are always available via the **Misc. Commands** menu shown to the left.

Note that the keyboard equivalent for all commands that have one is listed in the menu, next to the command. ('^' means **<Ctrl>**)

Also note that the bottom two commands, **Show hidden files**, and **Select files**, do not appear as buttons, ever. They're only accessible via the **Misc. Commands** menu.

Pops open a dialog box that prompts you for a directory name to *cd* to. This can be handy if the directory you want to go to is far away in the directory tree.

Keyboard Equivalent: **<Ctrl> c**

This command can only be issued when one or more files (or directories) are selected. It will pop up a confirmation box to make sure you didn't accidentally issue the command. At the confirmation box, you can type **<Ctrl> d** a second time to say **Ok**, or press **<Esc>** to **Cancel** the deletion. (Of course, you can just click on the appropriate button, as well.) Also, note that if you delete a subdirectory, all files and directories in that subdirectory will be recursively deleted. Finally, note that you can't delete the `<parent>` directory... for your own protection.

Keyboard Equivalent: **<Ctrl> d**

When you issue this command you'll be prompted for a plain filename for the new directory. You can only create subdirectories in the current directory.

Keyboard Equivalent: **<Ctrl> n**

`Rename file...  ^r`

This command can only be used when a single file (or subdirectory) is selected. You'll be prompted for the new name. Also, note that you can't rename the `<parent>` directory, again, for your own good.

Keyboard Equivalent: **<Ctrl> r**

`Rescan directory  ^s`

Rescans the current directory. Since other programs may be creating or deleting files in the current directory, the contents shown in the schnauzer may not always be up-to-date. Use the **ReScan** command to re-read the current directory.

Keyboard Equivalent: **<Ctrl> s**

`Update icons  ^u`

Like the **ReScan** command, but more serious about it. In addition to re-reading the current directory contents, it will also look for files that don't have icons created (or files that have later modification times than their associated icon files) and generate icons for them. After that, it will look for icon files that no longer have associated image files (which can happen if you delete the image files without using the schnauzer), and deletes them. Note that this command is *not* limited to the selected files, unlike the **Generate icon(s)** command listed below.

Keyboard Equivalent: **<Ctrl> u**

`Open new window  ^w`

Opens a new schnauzer window, up to a maximum of 4. The new schnauzer opened will display the same directory that the opening-schnauzer was displaying.

Keyboard Equivalent: **<Ctrl> w**

`Generate icon(s)  ^g`

Unconditionally generates icons for all the currently selected files. As such, you can only issue this command when one or more files are selected.

Keyboard Equivalent: **<Ctrl> g**

`Select all files  ^a`

Selects all files in the current directory.

Keyboard Equivalent: **<Ctrl> a**

`Text view  ^t`

Displays the currently-selected file in a TextView window. This command can only be used when there is exactly one file currently selected. Note that if you double-click on a file in an unrecognized format, *xv* will automatically display it in a TextView window. This command lets you view recognized image files in text mode.

Keyboard Equivalent: **<Ctrl> t**

`Recursive Update  ^e`

Recursively walks its way down a directory tree, starting from the current directory, and does an **Update** command in each subdirectory. It will follow symbolic links, if you have any, and it's smart enough to avoid circular links. It gives you a nifty way to generate icons for an entire hierarchy of image files. It can also take a Good Long Time, depending on your particular situation, so it pops up an annoying “Are You Sure?” dialog box before it starts working its magic.

Keyboard Equivalent: **<Ctrl> e**

Quit xv ^q

Exits the *xv* program.

Keyboard Equivalent: **<Ctrl> q**

Close window ^c

Closes this particular schnauzer window.

Keyboard Equivalent: **<Ctrl> c**

Show hidden files

Toggles whether or not ‘hidden’ files (in Unix, files that have a ‘.’ as their first character) should be displayed.

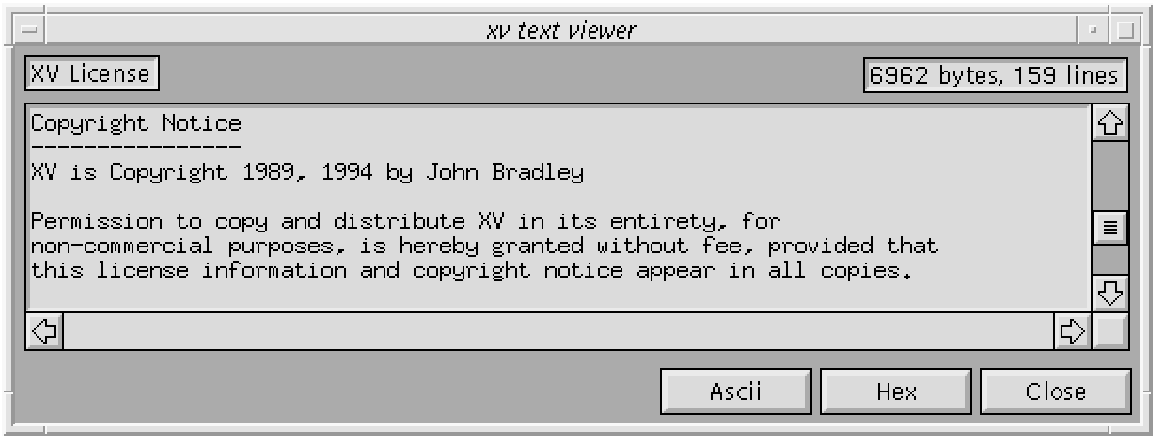
Keyboard Equivalent: <none>

Select files... ^f

Opens a dialog box that lets you select files by name. Wildcards are accepted. It does *not* clear any previously selected files.

Keyboard Equivalent: **<Ctrl> f**

Section 7: The TextView Window



Section 7.1: Overview

xv now has the ability to display arbitrary text data. While there are several different ways to cause a *TextView* window to appear, (such as opening an unrecognized type of file, using the **Text view** command in the *Visual Schnauzer*, or the **TextView**, **Comments**, or **License** commands in the *xv controls* window) all of the windows behave the same.

The TextView window has two primary modes of operation. It can display data as ASCII text, or it can display data in hexadecimal format.

There are a total of two TextView windows available. One is reserved for the **Comments** command. The other is used for all other text data.

As you might suspect, you can scroll the TextView windows by using the mouse in the scroll bars (as described in “Section 3.12.1: Operating a List Window”). Likewise, you can scroll the window around using the arrow keys on your keyboard, the **<PageUp>** and **<PageDown>** keys (sometimes labeled **<Prev>** and **<Next>**) and the **<Home>** and **<End>** keys. It behaves as you would expect.

You can switch modes by pressing the nice friendly **Ascii** and **Hex** buttons. **Close** (or pressing **<Esc>**) will close the TextView window.

Section 7.1.2: ASCII Mode

This is the default mode for the TextView windows. It will display text of any width and length. The only limitation is that all of the text must fit into memory. If it doesn’t, you’ll get an appropriate error message, and the text will not be displayed.

TextView windows are resizable. When you change the size of the main window, the inner text display window will change size appropriately, and display more (or less) data. While the default size of all TextView windows (except for the **Comments** window) is 80 characters wide by 24 lines high, there is nothing magical about these numbers. They’re only chosen out of a sense of tradition.

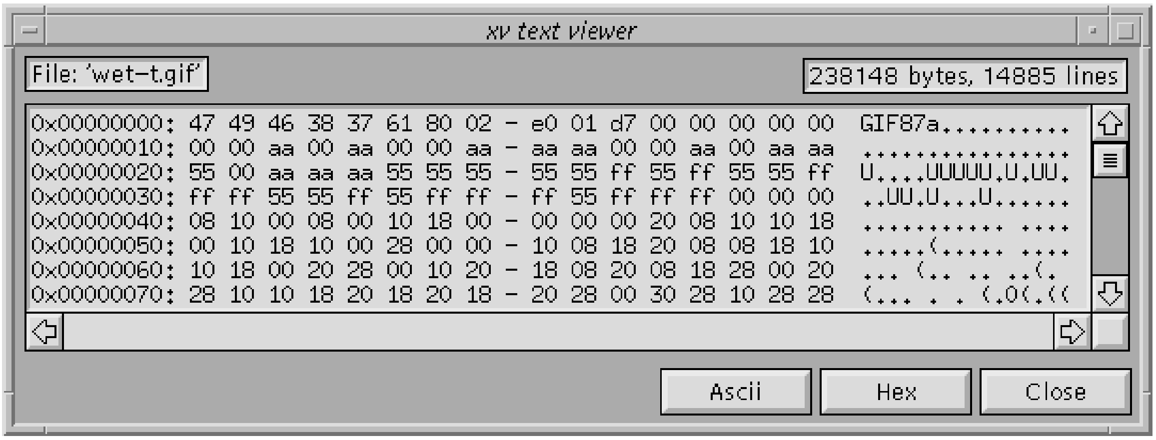
When in ASCII mode, the TAB character (ctrl-I) and the NL character (ctrl-J) are interpreted as is normal on a UNIX system (e.g. tab stops every 8 characters, NL marks end of line). The CR character (ctrl-M) is not displayed.

All other control characters (characters with an ASCII value less than ‘32’) are displayed with a caret (^) and the appropriate upper-case letter. For example, character number 17 (decimal) will be displayed as ‘^Q’.

All characters with an ASCII value greater than 127 are displayed as octal numbers with a leading backslash. As an example, character number 128 (decimal) will display as ‘\200’.

All other characters are displayed with the appropriate standard glyph.

Section 7.1.3: Hex Mode



This mode is useful for displaying binary data. In fact, if you have some binary data to display, you might want to start up *xv* just to display it, as this mode beats the heck out of using the standard UNIX command ‘`od -ha`’.

The data is shown 16 bytes to a line. The first number on each line is the offset (in hex) from the beginning of the file. This is followed by 16 bytes, shown in hex, and then in ASCII. Bytes which have a value less than 0x20 or greater than 0x7f are shown as a ‘.’ in the ASCII section.

While you can resize the window while in **Hex** mode, changing the width will not be of any use, as the output is formatted for an 80 character wide TextView window. Making the window wider will just put unused space on the right side. Making the window narrower will just enable the horizontal scroll bar. Changing the height of the window may prove useful, however.

Section 7.2: The Comment Window

The *xv comments* window (opened by the **Comments** command in the *xv controls* window) operates in exactly the same way as the TextView window. You can even display image comments in **Hex** mode, though I don’t see that as being very useful.

The *xv comments* window displays any comments found in the currently-loaded image file. If there are no comments, the window will be empty. Note that only certain image formats support comment fields (GIF89, JPEG, TIFF, and PBM/PGM/PPM are the most likely formats to have comments).

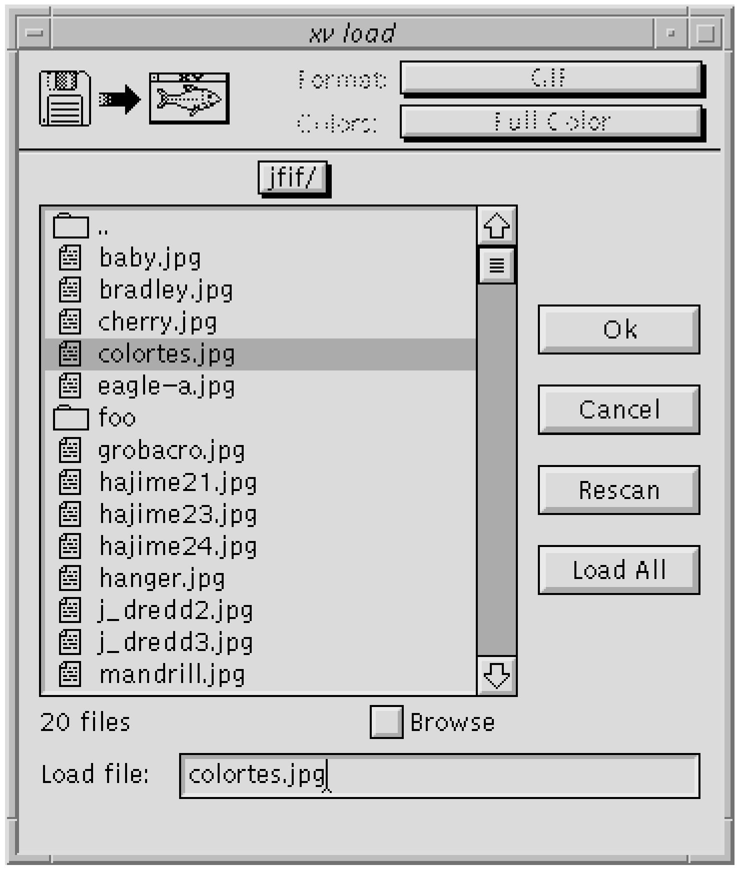
Whenever a new image is loaded, the *xv comments* window is updated to reflect the new image comments, or lack thereof.

Whenever you save an image in a format that supports comments, the comments from the last loaded image will be written out as well. This lets you read a file (say a GIF) which has comments, and write it out in another format (say, JPEG) preserving the comments.

Currently, *xv* does *not* give you a way to directly edit the image comments. Given how few images actually have comments in them, it seemed like a lot of work for very little gain. (Yeah, I know, if something actually let you edit the comments, perhaps people would start entering some...) Whatever. Seems likely enough that this feature will make it into a future version of *xv*. Of course, that's exactly what I said in the 3.00 documentation, a year and a half ago, so you may want to take that statement with a grain of salt...

Until then, if you *really* want to add or edit the comments in an image file, write it out as a PBM/PGM/PPM file. This format has a plain-text header (comments simply begin with a '#' character, which is not part of the comment), and you can edit it very easily with *emacs*, or any other reasonable text editor. Once that's done, you can use *xv* to convert the file back into the desired format.

Section 8: 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*. While it has been made somewhat obsolete by the *Visual Schnauzer*, the *xv load* window is considerably quicker.

The load window shows the contents of the current directory in a scrolling window. The files will be sorted alphabetically, with a small icon for each indicating the type of file (directory, executable, or normal file).

This list window operates in the same way that the one in the *xv controls* window works. (See “Section 3.12.1: 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>**, **<Down>**, **<Home>**, **<End>**, **<PageUp>**, and **<PageDown>** 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 hitting **<Return>**) will make *xv* 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. (Though a better way would probably involve using the **Visual Schnauzer**.)

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 a 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. 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:

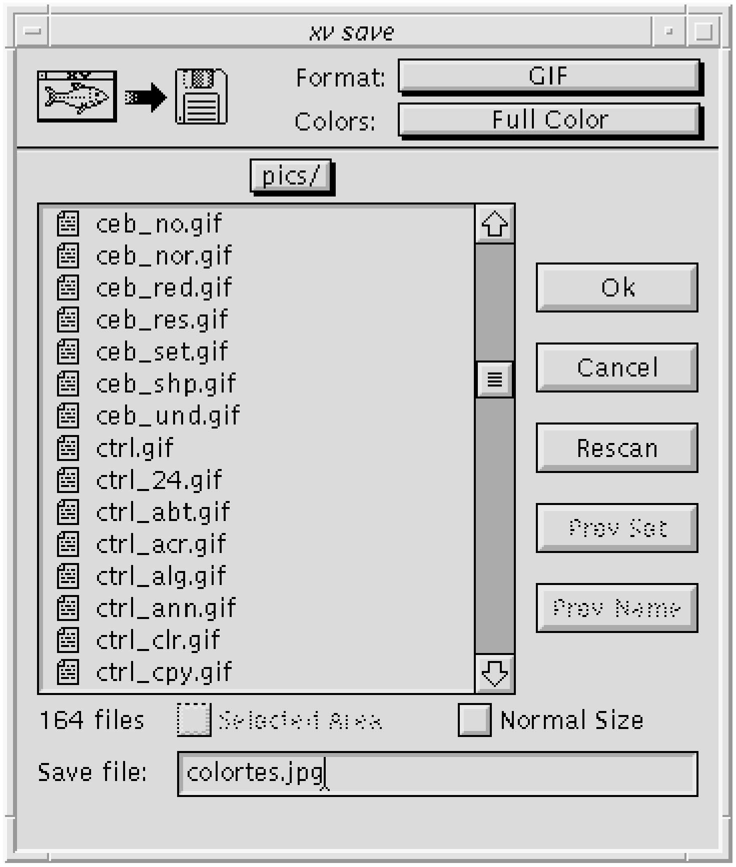
<Ctrl> f	moves the cursor forward one character
<Ctrl> b	moves the cursor backward one character
<Ctrl> a	moves the cursor to the beginning of the line
<Ctrl> e	moves the cursor to the end of the line
<Ctrl> d	deletes the character to the right of the cursor
<Ctrl> u	clears the entire line
<Ctrl> k	clears from the cursor position to the end of the line.
<Tab>	Filename completion. Note: it can only complete 'simple' filenames (no '/' chars)

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.

The **Load All** button simply copies the names of all 'plain files' found in the current directory into the *xv controls* filename list, where you have the **Next** and **Prev** buttons to help you look at a list of files.

Section 9: The Save Window



Damned if it doesn't look suspiciously like the *xv load* 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!* (i.e. At the current size, with the current color modification, rotation, cropping, etc. applied.) The only exceptions to this rule are if you are displaying images on a 1-bit B/W display, or displaying 24-bit images on a non-24-bit display. The fact that such images have to be dithered in order to be displayed doesn't count as 'modification', and the file won't be saved 'as displayed'. As such, you can manipulate and save color images on a 1-bit display, and 24-bit images on *any* type of display, even if you can't really see them 'as they are'.

For the most part, the *xv save* window operates exactly like the *xv load* window. (See "Section 8: 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. Likewise, the **Format** and **Colors** selections will reflect the currently loaded image. This behavior can be annoying if you are using *xv* to do file format conversion, or are routinely typing the same filename (a piped command, for instance).

The **Prev Set** button restores the **Format** and **Colors** choices to the settings that were used the last time a file was saved during this particular run of *xv*.

Likewise, the **Prev Name** button restores the filename string to the value it had the last time a file was saved.

At the top of the window are a pair of pull-down menus, **Format** and **Colors**. These menus let you choose the image file format and color settings that will be used when you save the image. Normally, they display the format that the current image is already in. If you change 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.

Normally, when you save an image, it will be saved at the current expansion (i.e., 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 particularly relevant when you're viewing images larger than your screen. By default, *xv* will automatically shrink images so that they 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, most notably the **Smooth** and **Dither** commands, 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.

If you have a selection rectangle drawn on the image, this control can be used. When enabled, only the contents of the selection will be saved. This can be very useful if you want to break parts of an image out into their own files, as was done *extensively* to put together this documentation. (i.e. I **Grab**'bed an *xv* window, such as this *save* window, moved the selection rectangle around so that it contained various parts of the image (such as the 'Selected Area' button to the left), and saved these parts in their own separate files, for use with Microsoft Word.)

One important note: If your selection goes 'off-screen' (beyond the boundaries of the *xv image* window), as can happen when you zoom-in to fine-tune the position of the selection rectangle, your selection will be *cropped* to the window when you save it, *unless* you also turn on the **Normal** checkbox. You can not save expanded areas of the image that are not visible in the *xv image* window, as such areas don't really exist (they aren't computed).

Section 9.1: Color Choices

At the top of the *xv save* window there is the **Colors** menu, which contains a list of possible color styles to use when saving the image.. 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 choices in the **Colors** menu 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.

Section 9.2: Format Notes

GIF

While *xv* can read both the GIF87a and GIF89a variants of the GIF format, it will normally write GIF87a. *xv* will only write a GIF89a file if there are image comments to be saved (comment blocks being a GIF89a extension).. This is in keeping with the GIF89 specification, which states that if you don't need any of the features added in GIF89, you should continue to write GIF87, for greater compatibility with old GIF87-only readers.

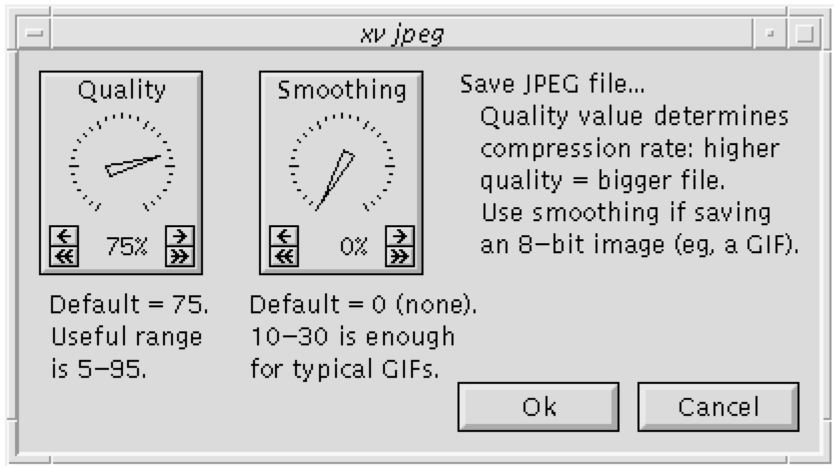
Since GIF only supports one format (up to 8 bits per pixel, with a colormap), there will be no file size difference between a **Full Color** and a **Greyscale** image. A **B/W Dithered** image, on the other hand, will be considerably smaller.

If you are currently in **24-bit mode**, and you are saving in any color mode other than **B/W Dithered**, the currently selected 24->8 conversion algorithm will be used to generate an 8-bit version of the current image, and that image will be written. (See "Section 3.5: The 24/8 Bit Menu" for more info.)

JPEG

One minor clarification: 'JPEG' is an image compression algorithm, *not* an image file format. 'JFIF' is a file format created by the fine folks at the Independent JPEG Group for the storage and interchange of 'JPEG'-ed image data. As such, it is the de-facto standard for 'JPEG' images. So, technically, when folks talk about 'JPEG' files, what they probably mean is 'JFIF' files, but it's not *necessarily* so. Therefore, it is possible to run across 'JPEG' files that *xv* can't read, because they don't adhere to the 'JFIF' file format standard. Just thought you'd like to know.

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 do so, a **Full Color** JPEG file will be saved.



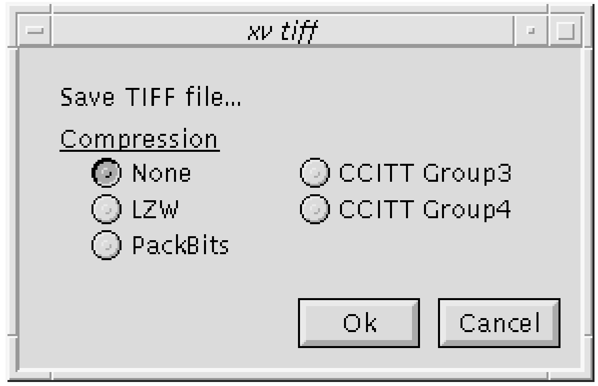
When you save in the JPEG format, the dialog box shown above will pop up and ask you for a quality setting and a smoothing value. '75%' is the default quality value, and really, it's a fine choice. You shouldn't have to change it under normal circumstances. The lower you set the quality, the higher the compression ratio will be (i.e., the JPEG file will be smaller). Note that setting the quality setting to '100%' will *not* result in 'lossless JPEG' (i.e., the original and the JPEG'd images will *not* be exactly the same, just very close).

The smoothing value is used to 'blur' images before saving them. It's often a good idea to blur GIF (and other 8-bit color) images before saving them, as you'll get better compression that way, and it also may partially undo the dithering that been done to many 8-bit images. On the downside, you'll also get somewhat blurred images. Something you have to decide for yourself.

Note: The JPEG support in *xv* is optional. While it is normally enabled, it is possible that it may not be enabled on your system (due to problems compiling the JPEG library, or something). If this is the case, you won't have a **JPEG** selection in the **Format** menu. Please feel free to complain to whomever built the binary that you're using.

TIFF

Full Color and **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, the dialog box shown above will pop up and ask you which type of image compression it should use. **None**, **LZW**, and **PackBits** compression types are available for use with all the Color modes. In addition, there are two **B/W Dithered**-only algorithms, **CCITT Group3** and **CCITT Group4**.

Note: The TIFF support in *xv* is optional. While it is normally enabled, it is possible that it may not be enabled on your system (due to problems compiling the TIFF library, or something). If this is the case, you won’t have a **TIFF** selection in the **Format** menu. Please feel free to complain to whomever built the binary that you’re using.

PostScript

Full Color and **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. (Section 10: The PostScript Window”, for details.)

PBM/PGM/PPM (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, as it produces considerably smaller image files.

PBM/PGM/PPM (ascii)

Like **PBM/PGM/PPM (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/PGM/PPM (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*-created PBM files (both **raw** and **ascii** variants) 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.

XPM

New to *xv* version 3.10, you can now load and save images in the XPM (X PixMap) format. Such files are best suited to small-ish images with a small number of colors, such as multi-color icons, and the like. One nifty feature of this format is that images can be created and edited using a text editor. All color choices work as expected. One warning: *xv* will not write out an XPM file with more than 256 different colors. If you are viewing a 24-bit image and save it as an XPM file, *xv* will first compute an 8-bit version of the image using the current 24->8 conversion algorithm (see "Section 3.5: The 24/8 Bit Menu"), and then save that.

BMP

xv will write a number of different types of BMP files depending on the 8/24 bit mode that you're in, the number of colors in the image, and the current 'Colors' choice.

If you are currently in **8-bit Mode**, and you select **Full Color**, **Reduced Color,** or **Greyscale**, *xv* will write out an uncompressed 4- or 8-bits per pixel BMP file, based on the number of different colors in the current image.

If you are in **24-bit Mode** and you select **Full Color**, the program will write out an uncompressed 24-bits per pixel image.

If you are in **24-bit Mode** and you select **Greyscale**, an uncompressed 8-bit per pixel BMP file will be written.

If you select **B/W Dither**, a 1-bit per pixel BMP file will be written.

Sun Rasterfile

Full Color and **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.

IRIS RGB

If you select **Full Color** or **Reduced Color**, the program will write a 24-bit image. Otherwise, it will write out an 8-bit image.

Targa (24-bit)

Another new format added to *xv* version 3.10. *xv* can now read and write 24-bit uncompressed Targa files. As such files are about the only reason folks still use the Targa format, this shouldn't be a problem. The color choices will do what you'd expect, but given that *xv* only reads/writes 24-bit Targa files, saving anything but **Full Color** would be a waste.

FITS

Saves images in the FITS format, a greyscale-only format used primarily by astronomers.

PM

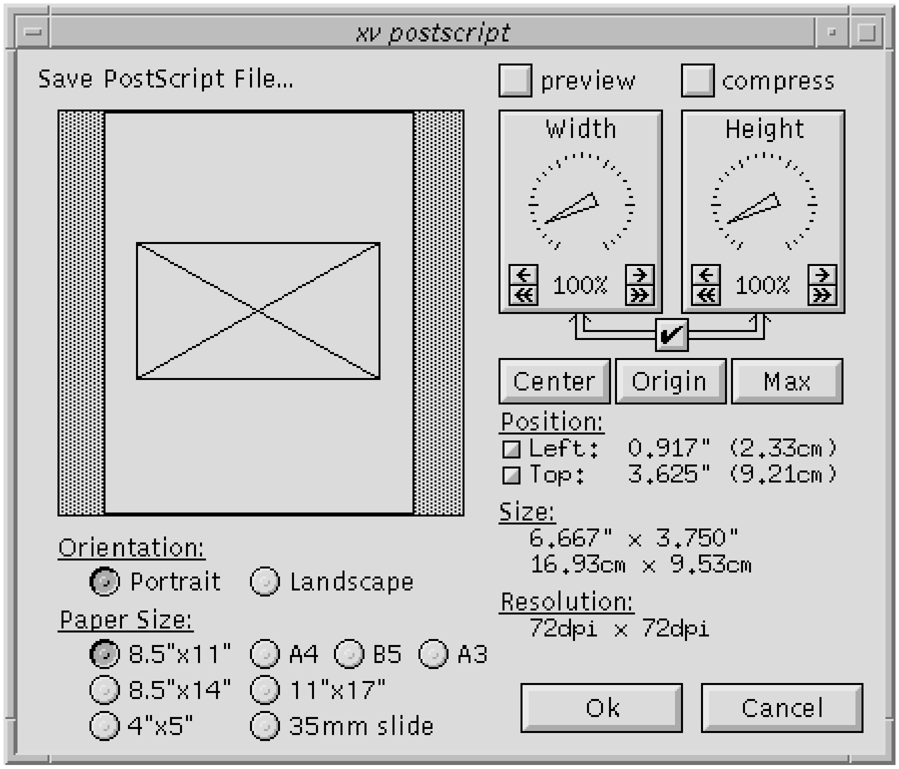
Note: This format is was developed at Penn, and is rarely seen outside of Penn. If you don't know anything about it, trust me, you don't want to.

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.

Filename List

This is not an 'image file format' at all, which is why it appears separate from the rest of the formats in the **Format** menu. Instead of saving the current image, what this does is save a list of all the filenames in the *xv* controls window file list. This file list can be used in conjunction with the '`-flist`' option. See "Section 12: Modifying xv Behavior" for more details.

Section 10: The PostScript Window

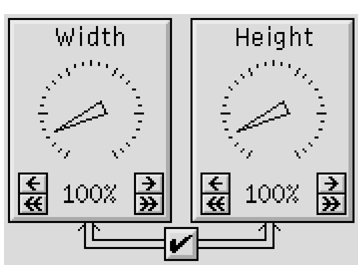


The *xv postscript* window lets you describe how your image should look when printed. You can set the paper size and the image size, position the image on the paper, and print in 'portrait' or 'landscape' mode.

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.

One note on the ‘Resolution’ setting: I get a fair number of complaints about the default of ‘100% = 72 dpi’. Many folks wonder why they can’t set the resolution to 300 dpi, given that most PostScript printers are 300 dpi. The answer, of course, is that they can. While you are constrained to integral percentage scaling factors, all the *good* resolutions (72, 75, 100, 150, 300, and 600 dpi) are attainable with appropriate settings of the scaling factors (100, 96, 72, 48, 24, and 12%, respectively).

Now, one other thing to note is that, *unless* you are printing a black-and-white (not greyscale) image, you almost certainly do *not* want to print at 300 dpi (or whatever your printer’s resolution is). At this time, all but the most enormous and expensive PostScript printers (> $20K) are fundamentally *bitonal*. That is, they can either put a dot at a certain position on a page, or *not* put a dot there. They have no ability (or nearly none) to adjust the size or darkness of the dot. As such, they cannot put an arbitrary 8-bit shade of gray (or 24-bit RGB color) at each 300 (or 600) dpi pixel position. They approximate grays (and colors) by dithering over a larger area. For example, if you print at 75 dpi onto a 300 dpi printer, each arbitrarily colored pixel in your source image will be represented by a 4x4 grid of bitonal dots on the page. This is why you should print non-bitonal images at low resolutions, such as 72 dpi. If you try to print them at higher resolutions, you will lose shades of gray (or different colors).

Located below the ‘page’ rectangle are a set of radio buttons that let you specify the current paper size (8.5”x11”, 8.5”x14”, 11”x17”, A3, A4, B5, 4”x5”, and 35mm), and orientation (Portrait and Landscape).

This button will center the image on the page.

This button will align the bottom-left of the image with the bottom-left of the page. (Which is the origin in the standard PostScript coordinate system.) This may be useful if you are generating Encapsulated PostScript files for use in some desktop-publishing package.

This 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).

This 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.

Specifies whether or not to generate compressed 8-bit per pixel PostScript. This is particularly handy if you’re generating color PostScript, and you are currently in **8-bit Mode**, 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 compression may make up for the increased execution time. Also note that the '**compress**' checkbox is not available when you are in **24-bit Mode**.

Click the **Ok** button when you're done. 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 11: External Programs

Section 11.1: bggen

Syntax: **bggen** [-h *high*] [-w *wide*] [-b *bits*] [-g *geom*] [-d *disp*] [-a] [-r *rot*] [-G *rptgeom*]
color1 [*color2* ... *colorN*]

bggen generates background gradients for use with *xv*. It does not directly display anything itself. Instead, it generates an appropriately sized PPM file, which *xv* can display. It is also used by the *xv*'s **Pad** command, for a similar purpose.

By default, *bggen* will generate a screen-sized PPM file with a vertical color gradient. The top of the image will be in *color1*, the bottom of the image will be in *colorN*, and intermediate positions will have an appropriate intermediate color. You can list as many colors as you'd like. If you only list one color, the entire image will be that single color.

Color specifications can be in one of three formats:

- A color name (e.g. 'lightblue') known to your X server. Use the *showrgb* command to get a (gigantic) list of known colors.
- An X11-style hexadecimal RGB color specification (e.g. '#00ff00'), where the first two hex digits specify the red value, the middle two specify the green value, and the last two specify the blue value. Note that the '#' character is treated specially in some shells, and to be on the safe side you should either protect it by putting a '\' character in front of it, or wrap the whole color specification in single-quotes.
- Three integers, in the 0-255 range, separated by spaces (e.g. 0 255 0). This is mainly for backward compatibility, as it was the only format understood by previous versions of *bggen*.

Options:

-d *disp*	Specifies the name of the X display that the program should talk to. *bggen* needs to connect to an X display to determine the screen size (which is used as the default image size), and to parse color names. It is not *necessary* for *bggen* to connect to an X display. If it can't, you will have to specify an image size (see below), and you'll have to use the numeric format to enter color specifications.
-g *geom*	Specifies the size of the generated image. *geom* should be in the format: *wide***x***high* (e.g. '1024x768'). If you don't specify a size, the image will default to the size of your display, assuming it was able to connect to your display.
-w *wide* -h *high*	Another way to specify the size of the generated image. Provided for backward compatibility.
-G *rptgeom*	Sets the size of the gradient. Normally, the gradient will be the size of the image. However, if you wish, you make the gradient smaller than the image; the gradient will repeat as often as necessary to fill the image
-r *rot*	Lets you rotate the gradient. Normally, the gradient is aligned vertically, with the first color at the top, and the last color at the bottom. Using this option, you can rotate the whole mess by an arbitrary amount. *rot* should be specified in degrees. Positive values rotate the gradient counter-clockwise.

-a	If supplied, *bggen* will generate a PPM 'ascii' file, rather than the default PPM 'raw' format.
-b *bits*	Specifies the number of significant bits to use in the RGB color specifications in the output image. It must range between 1 and 8, inclusive. Use values less than 8 (the default) to limit color use by increasing the color granularity. Since *xv* can do this sort of thing better, this is of dubious value.

Section 11.2: xcmap

This is a little X program I wrote years ago to help me debug *xv*'s color allocation code. It displays the contents of the default colormap as (typically) a 16x16 grid of colored rectangles. Run it, and then run something that allocates colors (such as *xv*), and you'll see the color cells changing color. It can give you an indication of how full your colortable is, and whatnot. It only works on displays where the default visual is a PseudoColor visual, but that's nearly all of them, so it shouldn't be a problem.

There are two standard X options: '-d *display*' to set the X display, and '-g *geom*', to set the default window size and position.

You can click in the window to examine the RGB values of various color cells. I have not found a useful purpose for this feature, but there must have been one at one time...

Type 'q' in the window to exit the program.

Section 11.3: xvpictoppm

This program is a standard Unix filter that reads the 'thumbnail' files created by the *visual schnauzer* (from `stdin`) and writes standard PPM versions of same (to `stdout`). The thumbnail files are small (80x60 maximum) 8-bit versions of your image files and are used as the icons in the *visual schnauzer*. These files are stored in my own weird format (one optimized for my purposes), but many folks have asked for a way to turn them into standard image files, for use on their WWW pages, or whatever.

Anyway, assuming you've done an **Update** or **Generate Icons** command in the *visual schnauzer*, you'll find the thumbnail version of an image stored with the same name in the '`.xvpics`' subdirectory. For example, if you have a file '`/pics/gif/foobie.gif`', you would find the thumbnail version of it in the file '`/pics/gif/.xvpics/foobie.gif`'.

Section 11.4: cjpeg and djpeg

These two programs are distributed as part of the Independent JPEG Group's software, located in the `jpeg` subdirectory of the *xv* source. They perform command-line conversion between JPEG and the PPM/PGM, GIF, BMP and Targa formats. Since *xv* has absolutely no command-line or 'batch' conversion abilities, you may find these useful if you want to convert a lot of images, or automatically convert images from a script. If this is the case, you should probably get Jef Poskanzer's most-excellent *pbmplus* package, which you can get via anonymous ftp on `ftp.x.org` and elsewhere..

Section 11.5: vdcomp

Uncompresses compressed PDS and VICAR images. Beyond that, I know nothing.

Section 12: 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`'.

Section 12.1: Command Line Options Overview

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

```
xv [-] [-/+24] [-/+2xlimit] [-/+4x3] [-/+8] [-/+acrop] [-aspect w:h] [-best24]
   [-bg color] [-black color] [-bw width] [-/+cecmap] [-cegeometry geom]
   [-/+cemap] [-cgamma rval gval bval] [-cgeometry geom] [-/+clear] [-/+close]
   [-/+cmap] [-cmtgeometry geom] [-/+cmtmap] [-crop x y w h] [-cursor char#]
   [-DEBUG level] [-dir directory] [-display disp] [-/+dither] [-drift dx dy]
   [-expand exp | hexp:vexp] [-fg color] [-/+fixed] [-flist fname]
   [-gamma val] [-geometry geom] [-grabdelay seconds] [-gsdev str]
   [-gsgeom geom] [-gsres int] [-help] [-/+hflip] [-hi color] [-/+hist]
   [-/+hsv] [-icgeometry geom] [-/+iconic] [-igeometry geom] [-/+imap]
   [-/+lbrowse] [-lo color] [-/+loadclear] [-/+max] [-/+maxpect] [-mfn font]
   [-/+mono] [-name str] [-ncols #] [-/+ninstall] [-/+nodecor] [-/+nofreecols]
   [-/+nolimits] [-/+nopos] [-/+noqcheck] [-/+noresetroot] [-/+norm]
   [-/+nostat] [-/+owncmap] [-/+perfect] [-/+poll] [-preset #] [-quick24]
   [-/+quit] [-/+random] [-/+raw] [-rbg color] [-rfg color] [-/+rgb] [-RM]
   [-rmode #] [-/+root] [-rotate deg] [-/+rv] [-/+rw] [-slow24] [-/+smooth]
   [-/+stdcmap] [-tgeometry geom] [-/+vflip] [-/+viewonly] [-visual type]
   [-/+vsdisable] [-vsgeometry geom] [-/+vsmap] [-/+vsperfect] [-wait seconds]
   [-white color] [-/+wloop] [filename ...]
```

Yes, as a matter of fact, I *am* a little embarrassed about it.

One big change starting in *xv* 3.10: In the past, most of the command-line options were *toggles*, in that they would normally turn some feature on, but if you had an X resource that also turned that feature on, then they would turn that feature *off*. Now, they are absolute settings. For instance, '`-rv`' turns on reverse video mode, regardless of the setting of the '`xv.reverse`' resource, and '`+rv`' turns *off* reverse video mode, regardless of the setting of the '`xv.reverse`' resource. (Thanks Peder!)

Section 12.2: General Options

Note: In the following sections, the part of the option name shown in **boldface** is the shortest allowable abbreviation of the option in question.

-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 12.3: Image Sizing Options

-ge`ometry` *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)

±fi`xed` — 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 preserve the original aspect ratio of the picture. (`+fixed` turns off this feature, forcing the image to be exactly the size specified by the `-geometry` option.)

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)

-e`xpand` *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.

You can also specify a pair of expansion values, separated by a colon, to set different horizontal and vertical expansion ratios. For instance, '3:2' would expand images 3x horizontally, and 2x vertically.

(Resource name: `expand`. Type: string)

-as`pect` *w:h*

Lets you set an initial aspect ratio, and also sets the value used by the **Aspect** command. 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 12.4: Color Allocation Options

-nc`ols` *nc*

Sets the maximum number of colors that *xv* will use. By default, this is '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.)

The other major use of this option is to limit the number of colors used when putting up an image that's going to be hanging around for a while. (i.e., an image in the root window) You may want to limit the number of colors used for such images so that other programs will still have some colorcells available for their own use.

(Resource name: `ncols`. Type: integer)

±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 major reason to 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. (`+rw` forces *xv* to use read-only color, the default)

(Resource name: `rwColor`. Type: boolean)

±pe`rfect`

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)

±o`wncmap`

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)

±st`dcmap` — Puts *xv* into **Use Std. Colormap** mode. All images will be shown dithered using the same set of colors. This lets you run multiple copies of *xv* to display multiple images simultaneously, and still have enough colors to go around.

(Resource name: `useStdCmap`. Type: boolean)

±cec`map` — 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 not install the colormap in the color editor window, as this often makes the color editor window unreadable. Note, however that the *Colormap Editor* tool will be appear somewhat misleading. (This option only applies when the '`-perfect`' or '`-owncmap`' options create their own colormaps.)

(Resource name: `ceditColorMap`. Type boolean)

±ni`nstall` — 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 12.5: 8/24-Bit Options

See "Section 3.5: The 24/8 Bit Menu" for further information about the following options.

±8 — Locks *xv* into **8-**bit **Mode**.

(Resource name: `force8`. Type: boolean)

±24 — Locks *xv* into **24-bit Mode**.

(Resource name: `force24`. Type: boolean)

The following three 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.), and you have *xv* locked into **8-bit Mode**, or you save 24-bit image data into an 8-bit graphics file format (such as GIF). They have no effect whatsoever on how GIF pictures or 8-bit greyscale images are displayed.

-quic`k24` — 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)

-sl`ow24` — 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.

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.

(Resource name: `slow24`. Type: boolean)

-be`st24`

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.

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.

(Resource name: `best24`. Type: boolean)

-noq`check`

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.

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.)

(Resource name: `noqcheck`. Type: boolean)

Section 12.6: 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 (keypresses 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.

±roo`t`

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. Defaults to style '0' if `-rmode` is not specified.

(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
8: symmetrical tiling
9: symmetrical mirrored tiling

The default mode is '0'. See "Section 3.4: Root Display Modes" for a description of the different display modes. Also, if you specify a '`-rmode`' option on the command line, it is not necessary to also specify the '`-root`' option.

(Resource name: `rootMode`. Type: integer)

±noresetroot — Lets you turn off the clearing of the root window that happens when you switch from a 'root' display mode 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. In this case, 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, whether you're in a 'root' mode or not..

(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.

If you aren't using a 'root' mode, this option will make *xv* exit as soon as the user clicks any mouse button in the image window. This is useful if you are calling *xv* from some other program to display an image.

(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 12.7: Window Options

xv currently consists has several top-level windows, plus one window for the actual image. These windows (the *xv controls* window, the *xv info* window, the *xv color editor* window, the *xv comments* window, the *xv text viewer* window, and the *xv visual schnauzer*) 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)

±cmtmap

Maps the *xv comments* window.

(Resource name: `commentMap`. Type: boolean)

-cmtg`eometry` *geom* — Sets the initial geometry of the *xv comments* window.

(Resource name: `commentGeometry`. Type: string)

-tg`eometry` *geom* — Sets the initial geometry for any *TextView* windows (other than the *xv comments* window).

(Resource name: `textviewGeometry`. Type: string)

±vsm`ap` — Maps an *xv visual schnauzer* window.

(Resource name: `vsMap`. Type: boolean)

-vsg`eometry` *geom* — Sets the initial geometry of the *xv visual schnauzer* windows.

(Resource name: `vsGeometry`. Type: string)

±nop`os` — 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 12.8: Image Manipulation Options

±dit`her` — 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), or in conjunction with the '`-ncols`' option.

(Resource name: `autoDither`. Type: boolean)

±sm`ooth` — 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)

±ra`w` — Forces *xv* to display the image in **Raw** mode. Mainly used to override the `autoDither` or `autoSmooth` resources. Can also be used to turn off the automatic dithering and smoothing that occurs when you are using **Use Std. Colormap** mode or when an image is shrunk to fit the screen.

(Resource name: `autoRaw`. Type: boolean)

-cr`op` *x y w h* — Tells *xv* to automatically crop the specified region of the image. The rectangle is specified in *image* coordinates, which remain constant (regardless of any expansion/compression of the *displayed* image). This is useful if you want to view a series of images, and you only want to see one common area of the images. For example, you may have the GIF weather maps of the United States, but only want to display your general region of the country.

(Resource name: <none>)

±acrop

When specified, tells *xv* to automatically issue an **AutoCrop** command whenever an image is first displayed.

(Resource name: `autoCrop`. Type: boolean)

±4x3

Automatically issues a **4x3** command whenever an image is loaded.

(Resource name: `auto4x3`. Type: boolean)

±hflip

Automatically issues a 'horizontal flip' command whenever an image is loaded.

(Resource name: `autoHFlip`. Type: boolean)

±vflip

Automatically issues a 'vertical flip' command whenever an image is loaded.

(Resource name: `autoVFlip`. Type: boolean)

-rotate *deg*

Automatically rotates the image by the specified amount whenever an image is loaded. *deg* can be 0, ±90, ±180, or ±270. Positive values rotate the image clockwise, negative values rotate the image counter-clockwise.

(Resource name: `autoRotate`. Type: integer)

±norm

Automatically issues a **Norm** command (to normalize the contrast of an image) whenever an image is loaded.

(Resource name: `autoNorm`. Type: boolean)

±hist

Automatically issues a **HistEq** command (to do histogram equalization) whenever an image is loaded.

(Resource name: `autoHist`. Type: boolean)

-gamma *val*

Sets the *Intensity* graph (in the *xv color editor* window) to the gamma function of the specified value.

(Resource name: <none>)

-cgamma *rv gv bv*

Sets the *Red, Green,* and *Blue* graphs in the *xv color editor* window to the gamma functions of the specified values.

(Resource name: <none>)

-preset *preset*

Makes the specified *preset* (in the *xv color editor*) the default. It does this by swapping the specified preset (**1**, **2**, **3**, or **4**) with the settings associated with the **Reset** button.

(Resource name: `defaultPreset`. Type: integer)

Section 12.9: Miscellaneous Options

±mono

Forces the image to be displayed in 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 greyscale Sun 3/60s.) The `-mono` option corrects this behavior.

(Resource name: `mono`. 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.

-wh`ite` *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)

-bl`ack` *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.)

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

(Resource name: `black`. Type: string)

-wa`it` *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 for 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.

Note: If you are in **Use Std. Colormap** mode, and you use `-wait 0`, the images will *not* be dithered (as they normally are when you are in **Use Std. Colormap** mode). It's assumed that if you said '`-wait 0`' that you want the images displayed at maximum speed. You can still turn the dithering on if you desire by using the `-dither` option.

(Resource name: <none>)

±wl`oop` 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>)

±ra`ndom` Makes *xv* display multiple image files in a random order. Useful for breaking up the monotony of having slide-shows always display in the same order. Also, if you also use the `-quit` option, you can have *xv* display a single, random file from a list of files. This may be useful if you'd like *xv* to pick a random 'background image' from some set of files.

(Resource name: <none>)

±loa`dclear`

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 SA*`', and voila! Just like the evening news!

(Resource name: `clearOnLoad` Type: boolean)

±nof`reecols`

Whenever you load a new image, *xv* normally frees the colors it was using for the previous image, and allocates new colors for the new image. This can cause 'rainbow' effects on PseudoColor displays as the colors are changed. You can avoid this problem entirely by using the `-nofreecols` option, which suppresses the normal freeing of old colors. This is most useful when doing slide-shows. Note, however that there will be fewer colors available for 'later' images. These images will wind up being displayed with whatever colors were allocated for the earlier images. As such, they may or may not look that hot... (And allow me to reiterate: *xv* is *not* an image animator, despite options like these that let it do so, albeit poorly.)

(Resource name: <none>)

±rg`b`

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 (*false*))

±hs`v`

Specifies that, by default, the colormap editing dials in the *xv color editor* window should be in HSV mode.

(Resource name: `hsvMode`. Type: boolean (*true*))

±lb`rowse`

Turns on the **Browse** checkbox in the *xv load* window. This keeps the window from being automatically closed whenever you successfully load an image.

(Resource name: `loadBrowse`. Type: boolean)

±nos`tat`

Speeds up the performance of the *xv load* and *xv save* windows. (Which are really the same window...) It keeps *xv* from doing a `stat()` system call for 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)

-vis`ual` *vistype*

Normally, *xv* uses the 'best' visual model it can get. It looks for a 24 or 32-bit deep *TrueColor*, or *DirectColor* visual. If it can get one, that's what it uses. Otherwise, it will fall back to using the 'default' visual provided by your X server. You can override this bit of cleverness by explicitly selecting a visual to use. Valid types are *StaticGray*, *StaticColor*, *TrueColor*, *GrayScale*, *PseudoColor*, *DirectColor*, and *default.* All of these modes are not necessarily provided on any given X display. Run *xdpyinfo* on your display to find out what visual types are supported. You can also specify a specific visual by using its numeric visual ID, in the case that you have multiple instances of a given visual type available (*xv* will pick the 'deepest' one by default)

(Resource name: `visual`. Type: string)

-cu`rsor` *curs*

Specifies an alternate cursor to use in the image window (instead of the normal 'arrow' 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)

±2x`limit`

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)

±nol`imits`

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)

±clo`se` — If specified, iconifying the *xv image* window will automatically close all the other *xv* windows. De-iconifying the *xv image* window will re-open the other *xv* windows.

(Resource name: `autoClose`. Type: boolean)

±ico`nic` — Starts *xv* with its image window iconified.

(Resource name: `iconic`. Type: boolean)

-icg`eometry` *geom* — Specifies the screen position of the icon (when you use the `-iconic` option).

(Resource name: `iconGeometry`. Type: string)

-dir *directory* — Specifies an initial directory for *xv* to switch to when run. Also specifies the default directory used for the *visual schnauzer* and the *xv load* and *xv save* windows.

(Resource name: `searchDirectory`. Type: string)

-fl`ist` *fname* — Tells *xv* to read a file *fname* that consists of a list of names of image files to load, one per line. This file is in the same format generated by the **File list** checkbox in the *xv save* window. You can use this to get around shell 'command-length' limitations (which can hit you if you try '`xv *`' in a directory with a thousand or two files), or you could have *find* (or whatever) generate this file based on some type of criteria (age, size, etc.)

(Resource name: `fileList`. Type: string)

-dr`ift` *dx dy* — A kludge. In order to do certain operations, *xv* needs to be able to precisely position the contents of an image window on the screen. Unfortunately, window managers disagree wildly on exactly how the "where's the window" information will be presented to the program. The practical upshot is that, despite a sizable effort to the contrary, *xv* may very will have its image window 'drift' on the screen as you resize it. This option lets you specify correction factors to cancel out the drift. If the window drifts down and to the right, use negative values to cancel the drifting. If the window drifts up and to the left, use positive values to cancel the drifting.

(Resource name: `driftKludge`. Type: string)

-mf`n` *font* — Lets you specify the mono-spaced font used in the *TextView* windows, and a few other places. Be sure you use a mono-spaced font, or you may well get 'interesting' effects.

(Resource name: `monofont`. Type: string)

-na`me` *string* — Lets you change what string is displayed in the titlebar of the image window. Normally, *xv* will display the version number and the filename. If you're calling *xv* from another program, you may prefer to have it print a more descriptive string, or perhaps something like '<click mouse to quit>' if you're also using the `-quit` option.

(Resource name: <none>)

±vie`wonly`

For use when calling *xv* from some other program. Forces all user input to be ignored. This keeps the untrained (or inquisitive) user from nosing around, creating files, or just generally misbehaving. Also note that there's no way for the user to quit the program. It is up to the calling process to manually kill *xv* when it feels that the image has been displayed long enough.

(Resource name: <none>)

±gr`abdelay` *seconds*

Sets the default 'grab delay' in the **Grab** dialog box. See "Section 3.11: The Grab Command".

(Resource name: <none>)

±po`ll`

Turns file polling on. If enabled, *xv* will notice when the currently displayed image file changes (due to some other process rewriting it, or something like that), and it will automatically reload the image file once it appears to have settled down (once the file size stops changing for a few seconds). See "Section 3.12.3: Image Reloading" for further details.

(Resource name: <none>)

±vsp`erfect`

Normally, the *visual schnauzer* uses its own private colormap. This is necessary in order to get a good set of colors to display the image icons, and not steal colors away from the actual image window. However, you may find the colormap install/deinstall very annoying. I do. You can specify this option (`+vsperfect`) to turn the 'perfect' behavior off. If you do so, the *visual schnauzer* windows will steal away a small (64-entry) part of the colormap (unless you are in **Use Std. Colormap** mode, in which case they will share the standard colormap). The downside is that neither the schnauzer *nor* the image will look as good.

(Resource name: `vsPerfect`. Type: boolean)

±vsd`isable`

Completely disables the *visual schnauzer*. This is mainly so, if you have `vsPerfect` turned off, you can disable the schnauzer and keep it from stealing any colors from the image. In the default setting (`vsPerfect` is turned on), this option will have no useful effect.

(Resource name: `vsDisable`. Type: boolean)

-gsd`ev` *str*

Sets the 'device' that the *ghostscript* package will generate output for, which is used whenever you read a PostScript file in *xv*. Currently, the default device is **ppmraw**, which means all PostScript will be converted to either a 24-bit color PPM file, an 8-bit greyscale PGM file, or a 1-bit black-and-white PBM file, as appropriate (based on the particular PostScript file in question). Note that your copy of *ghostscript* must be configured to support this device, and any other values you may wish to set this option to. See the *xv* `Makefile` for further information.

Also note: Be very careful when using these options, as it's pretty easy to have *ghostscript* generate *enormous* data files. For example, for normal 8½" by 11" pages, at 72dpi, a PBM file will require ~60K per page, a PGM file will require ~500K per page, and a PPM file will require ~1.5M per page. If you have it generate images at 300 dpi (see below), these sizes explode to roughly 1, 8, and 24 megabytes per page.

As such, you should forget about viewing color pages at 300 dpi, and you may also want to forget about viewing multi-page PostScript files at 300 dpi.

(Resource name: `gsDevice`. Type: string)

-gsr`es` *res*
Specifies the resolution of the page files generated by *ghostscript*, in dots per inch. Defaults to 72 dpi. You can set it to any value, but be careful about generating enormous intermediate datafiles.

(Resource name: `gsResolution`. Type: integer)

-gsg`eom` *geom*
Sets the page size of the files generated by *ghostscript*. Normally, this defaults to '612x792', which is the size of 8½" by 11" paper, as measured in 72nds of an inch. Note that these numbers are in 72nds of an inch regardless of the resolution (dpi) value set by `gsResolution`.

(Resource name: `gsGeometry`. Type: string)

±nod`ecor`
Turns off certain decorations on the *xv image* window's frame, maybe. Works for *mwm*, at least. There is no way to do this trick with *olwm*, and *twm* doesn't need it. Anyway, if you turn off the titlebar and such on the *xv image* window, it removes the *minimum size* constraint/problem that I talk about in "Section 2.3: Cropping", and elsewhere.

(Resource name: `nodecor` Type: boolean)

-R`M`
If specified, *deletes* all files specified on the *xv* command line when *xv* exits. This can be handy if you're using *xv* as an external viewer, and you'd like to hand it a file, have it displayed, and then have it go away.

(Resource name: `nodecor` Type: boolean)

-D`EBUG` *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 12.10: Color Editor Resources

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

1. Start *xv* and put it in the background by typing '`xv &`'.
2. Type the command '`cat >foo`' in an active *xterm* window
3. Bring the *xv color editor* window up.
4. Issue the **Cut Resources** command.
5. 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.

6. You could type **<ctrl-D>** in the *xterm* to complete the *cat* command, edit this file, and put it in your `.Xdefaults` or `.Xresources` file.

The lines generated by Cut Resources will look like this:

```
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, 2, 3,** or **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 12.10.1: 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`', through '`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 12.10.2: 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 12.10.3: 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 the overall image color saturation.

Section 12.10.4: 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 12.10.5: Other Resources

`autoApply`	A boolean resource that sets the default condition of the **Auto-Apply HSV/RGB Mods** checkbox in the *xv color editor* window.
`displayMods`	A boolean resource that sets the default condition of the **Display With HSV/RGB Mods** checkbox in the *xv color editor* window.
`dragApply`	A boolean resource that sets the default condition of the **Auto-Apply While Dragging** checkbox in the *xv color editor* window.
`autoReset`	A boolean resource that sets the default condition of the **Auto-Reset On New Image** checkbox in the *xv color editor* window.
`saveNormal`	A boolean resource that sets the default condition of **Normal Size** checkbox in the *xv save* window.
`pspreview`	A boolean resource that sets the default condition of **Preview** checkbox in the *xv postscript* window.
`pscompress`	A boolean resource that sets the default condition of **Compress** checkbox in the *xv postscript* window.
`print`	A string resource that sets the default command that shows up in the **Print** dialog box.
`pspaper`	A string resource that sets the default paper size selected in the *xv postscript* window. Valid settings are: `8.5x11, 8.5x14, 11x17, 4x5 35mm a3 a4 b5`
`psorient`	A string resource that sets the default image orientation in the *xv postscript* window. Valid settings are: `portrait` and `landscape`

`psres`	An integer resource that sets the default image resolution in the *xv postscript* window. Valid settings are in the range 10 through 720 dpi.

Section 12.11: Window Classes

xv defines the following 'class' names for its various top-level windows:

`XVroot`	for the *xv image* window
`XVcontrols`	for the *xv controls* window
`XVdir`	for the *xv load* and *xv save* windows
`XVinfo`	for the *xv info* window
`XVcedit`	for the *xv color editor* window
`XVps`	for the *xv postscript* window
`XVjpeg`	for the *xv jpeg* window
`XVtiff`	for the *xv tiff* window
`XVconfirm`	for all the pop-up windows

You may be able to use these class names to do something nifty with your window manager. For instance, with *mwm* you can control which controls you'll get in the window frame, on a per-window basis. For example, to turn off all the *mwm* doodads that normally are tacked onto the *xv image* window, you could put this in your .Xdefaults file:

```
Mwm*XVroot*clientDecoration:  none
```

Section 13: Credits

Thanks go out to the following wonderful folks:

- First and foremost, **John Hagan**, best friend, lead beta-tester, and guy who got me a new job. He's been the driving force behind much of *xv*. Nearly all the major differences between the *xv* that you see today, and the *xgif* of four (five? six?) years ago can be traced to the years of continual harassment because of alleged (and actual) weaknesses 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. For example, he'd been asking for a *Visual Schnauzer* for *years* now...

- **Theresa O'Malley**, friend, sweetie, future wife. About a year and a half ago, in a decision she no doubt regrets, she asked if she could help me with my business. Since then, she's taken over all business aspects of *xv*. When you send mail to `xv@devo.dccs.upenn.edu`, or to `xvbiz@devo.dccs.upenn.edu`, she's the one who (eventually) answers your mail. It's a big job, and you all owe her a thanks, because if *I* had to handle all the *xv* business stuff, I would have had no time whatsoever to work on *xv* version 3.10, and you wouldn't be reading this now, or ever...

- **Filip Fuma**, my former boss, deserves a great deal of thanks for seeing the value of *xv*, and allowing, (if not actually encouraging) me to write it. I also owe him an eternal debt of gratitude for being the genius who said "You know, you should make this into shareware..."

- **Helen Anderson** has provided many fine ideas over the years, and has continued to be amused by *xv* for much of that time, more than I can occasionally say for myself. She also proofread the 2.20 version of this document, much of which remains verbatim.

- **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 would be safe to say that he "started it all." This code, now somewhat modified, is still in use in the module `xvgif.c`.

- **Michael Mauldin** (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.

- **Tom Lane** (tgl@sss.pgh.pa.us), for much advice and assistance on 24->8 bit quantization, color allocation, and, of course, JPEG issues.

  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, thereby providing the rest of us with the new standard graphics format (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!

- **Paul Haeberli** (paul@manray.asd.sgi.com) provided me with nice clean, portable code to read and write IRIS 'rgb' files.

- **Jef Poskanzer** (jef@well.sf.ca.us) is responsible for coming up with several cool/whizzo 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 and the XWD format support.

- **Rick Dyson** (dyson@iowasp.physics.uiowa.edu) has been doing the VMS ports of *xv* for the past couple years now. All you VMS users owe him a great big "Thank you," because you wouldn't have *xv* if it weren't for his efforts. I won't go *near* a VMS system. Rick also gets a "Thanks" from me for doing a good deal of beta-testing, as well.
- **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!
- **Bernie McIlroy** (berniem@microsoft.com) for providing me with information on the BMP format. (Which is included in the `docs` subdirectory, so please don't bother him!)
- **Anthony Datri** (aad@scr.siemens.com) for writing the PDS/VICAR i/o module.
- **Thomas Meyer** (i03a@alf.zfn.uni-bremen.de) for writing the IFF i/o module.
- **Chris Ross** (cross@eng.umd.edu), for writing the XPM i/o module, and in the process, inadvertently making me think about a great number of things...
- **David Robinson** (drtr@mail.ast.cam.ac.uk) for writing the FITS i/o module.

Section 13.1: The Hall of Fame

The following folks have all contributed to the development of *xv* in the form of bug reports, 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
Leon Avery	leon@eatworms.swmed.edu
Tim Ayers	ayers@mermaid.micro.umn.edu
Bill Barabash	barabash@rachel.enet.dec.com
Markus Baur	mbaur@ira.uka.de
Jason Berri	berri@aero.org
Richard Bingle	bingle@cs.purdue.edu
Alan Blanchard	alan@abraxas.mbt.washington.edu
Bob Boag	boag@marshall.wvnet.edu
David Boulware	dgb@landau.phys.washington.edu
Thomas Braeunl	braeunl@informatik.uni-stuttgart.de
Jon Brinkmann	jvb7u@astro.virginia.edu
Andrew Brooks	arb@comp.lancs.ac.uk
David Brooks	dbrooks@osf.org
Kevin Brown	brown@hpbsm15.boi.hp.com
Elaine Chen	eychen@athena.mit.edu
Jeff Coffler	coffler@jeck.amherst.nh.us
Reg Clemens	clemens@plk.af.mil
Paul Close	pdc@lunch.wpd.sgi.com
David A. Clunie	dclunie@flash.us.com
Jan D.	jhd@irfu.se
Anthony Datri	datri@convex.com
Berthold Dettlaff	dettlaff@informatik.uni-stuttgart.de
L. Peter Deutsch	ghost@aladdin.com
Derek Dongray	dongray@genrad.com
Rick Dyson	dyson@iowasp.physics.uiowa.edu
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
Robert Forsman	thoth@raybans.cis.ufl.edu
Peter Glassenbury	pete@cosc.canterbury.ac.nz
Michael Gleicher	mkg@stealth.plk.af.mil
Robert Goodwill	robert@earth.cs.jcu.edu.au
Dave Gregorich	dtg@csula-ps.calstatela.edu
Brian Gregory	bgregory@megatest.com
Ted Grzesik	tedg@apollo.hp.com
Harald Hanche-Olsen	hance@ams.sunysb.edu

Charles Hannum	mycroft@gnu.ai.mit.edu
Dave Heath	heath@cs.jhu.edu
Scott D. Heavner	sdh@po.cwru.edu
Bill Hess	hess@tethys.apl.washington.edu
Dave Hill	ddhill@zk3.dec.com
Tom Hinds	rocker@cs.bu.edu
Ricky KeangPo Ho	kpho@sabina.berkeley.edu
Mark Horstman	mh2620@sarek.sbc.com
Tetsuya Ikeda	tetsuya@is.s.u-tokyo.ac.jp
Yasuhiro Imoto	tomo@sonytek.co.jp
Lester Ingber	ingber@umiacs.umd.edu
Dana Jacobsen	jacobsd@solar.cor2.epa.gov
Roy Johnson	rjohnson@shell.com
Dave Jones	jonesd@kcgl1.eng.ohio-state.edu
Kjetil Jorgensen	jorgens@lise.unit.no
Jonathan Kamens	jik@pit-manager.mit.edu
Vivek Khera	khera@cs.duke.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
Bruce Labow	labow@itd.nrl.navy.mil
Dave Lampe	djl@ptc.timeplex.com
Tom Lane	tom_lane@g.gp.cs.cmu.edu
Peder Langlo	respl@mi.uib.no
Jeremy Lawrence	jeremy@snrc.uow.edu.au
Sam Leffer	sam@sgi.com
Jim Lick	jim@pi-chan.ucsb.edu
Jean Liddle	jliddle@mrcnext.cso.uiuc.edu
Michael Lipscomb	lipscomb@visionl.engr.utk.edu
Eam Lo	eam@netcom.com
Rolf Mayer	rz90@rz.uni-karlsruhe.de
Stephen Mautner	stephen@cs.utexas.edu
Tom McConnel	tmcconne@sedona.intel.com
Craig McGregor	craig@csdvax.csd.unsw.edu.au
Thomas Meyer	i03a@alf.zfn.uni-bremen.de
Peter Miller	pmiller@topaz.bmr.gov.au
Erwan Moysan	erwan.moysan@irisa.fr
Chris Newman	chrisn+@cmu.edu
Lars Bo Nielsen	lbn@hugin.dk
James Nugent	james@ironbark.bcae.oz.au
Arthur Olson	ado@elsie.nci.nih.gov
Machael Pall	pall@rz.uni-karlsruhe.de
Mike Patnode	mikep@sco.com
Nigel Pearson	nigel@socs.uts.edu.au
Joe Peterson	joe@avs.com
Colin Plumb	colin@nyx10.cs.du.edu
Daniel Pommert	daniel@ux1.cso.uiuc.edu
Robert Potter	rpotter@grip.cis.upenn.edu
Werner Randolf	evol@brian.uni-koblenz.de
Eric Raymond	eric@snark.thyrsus.com
Eric Rescorla	erk@eitech.com
Phil Richards	pgr@prg.oxford.ac.uk
Declan A. Rieb	darieb@sandia.gov
David Robinson	drtr@mail.ast.cam.ac.uk
R. P. Rodgers	rodgers@nlm.nih.gov
Chris P. Ross	cross@eng.umd.edu
Eckhard Rueggeberg	eckhard.rueggeberg@ts.go.dlr.de
Arvind Sabharwal	arvind@brutus.ct.gmr.com
Hitoshi Saji	saji@is.s.u-tokyo.ac.jp
Nick Sayer	mrapple@quack.kfu.com
Klaus Schnepper	df40@master.df.op.dlr.de
Steven Schoch	schoch@starnet.com
Bill Silvert	silvert@biome.bio.ns.ca
Ben Simons	ben@lewis.vislab.su.edu.au
Cameron Simpson	cameron@cse.unsw.edu.au
Mark Snitily	mark@zok.sgcs.com
Karsten Spang	krs@kampsax.dk
Greg Spencer	greg@longs.lance.colostate.edu
Matthew Stier	matthew@sunpix.east.sun.com
Andreas Stolcke	stolcke@icsi.Berkeley.edu

Rod Summers	rsummers@ard.fbi.gov
Steve Swales	steve@bat.lle.rochester.edu
Tony Sweeney	sweeney@ingres.com
Matt Thomas	thomas@netrix.lkg.dec.com
Rich Thomson	rthomson@dsd.es.com
Bill Turner	bturner@cv.hp.com
Larry W. Virden	lwv26@cas.org
John Walker	kelvin@autodesk.com
Doug Washburn	washburn@hpmpea2.cup.hp.com
Drew Watson	dwatson@encore.com
Chris Weikart	weikart@prl.dec.com
Michael Weller	eowmob@exp-math.uni-essen.de
Kenny Zalewski	zalewk@rpi.edu
Jamie Zawinski	jwz@lucid.com
Dan Zheme	dan@mordor.webo.dg.com

Section 13.2: The Beta Testers

Eric Demerling	AIX
Rick Dyson	VAX VMS, VMS on Alpha, SunOS
Tom Lane	HP systems
Sam Leffler	SGI systems
Chris Ross	Solaris, SunOS, Ultrix, NetBSD
Michael Weller	Linux
Myself	OSF/1, SunOS

Thanks to these folks we can be pretty sure that *xv* will at least compile cleanly on the vast majority of systems. Any bugs that have slipped through are entirely my fault, as, in all fairness, I haven't given these guys much time to test it. As such, the only platform that I can say *xv* has been seriously tested on is OSF/1 running on a DEC Alpha, as that's been my most recent development environment.

Section 13.3: Miscellaneous Ramblings

And, of course, thanks to everyone else. If you contributed to the developement of *xv* in some way, and I somehow forgot to put you in the big list, my humble apologies. Documentation and careful record-keeping are *not* my strong suits. "Heck," why do you think it takes me a year and a half to come up with a minor new release? Because, while I love to add new features to the code, I dread *documenting* the dumb things. Besides, we all know that writing the documentation is the hardest part of any program. Particularly when the good folks at id Software *insisted* upon releasing DOOM II...

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. (Well, that's not *actually* true. My love of nifty user-interfaces, all the wonderful code I've gotten from the folks listed above, and the fact that *xv* actually serves a useful purpose (albeit "displaying pictures of naked women") are the things that have made *xv* a real success. You folks who've written in have given me a way to *measure* how successful *xv* is.) But I digress. Thanks!

By the way, when I last counted (in October 1992), *xv* was in use at 180 different Universities, and dozens of businesses, goverment agencies, and the like, in 27 countries on 6 of the 7 continents. Since then, I've received messages from hundreds of new sites. And *xv* has been spotted in Antartica, bringing the total to 7 of 7 continents, and allowing me to claim that *xv* is, in fact, truly *global* software. That's probably a good thing. Does anybody know if there's a Unix workstation in the Space Shuttle?... `:-)`

Appendix A: Command Line Options

-	Tells *xv* to read an image from `<stdin>`.
±24	Lock *xv* into **24-bit Mode**.
±2x`limit`	Allow image windows to be twice the size of the screen.
±4`x3`	Issue **4x3** command when an image is loaded.
±8	Lock *xv* into **8-bit Mode**.
±ac`rop`	Issue an **AutoCrop** command when an image is loaded.
-as`pect` *w:h*	Sets the default ratio used by the **Aspect** command.
-be`st24`	Use the 'best' (read: slowest) 24-bit to 8-bit color algorithm.
-bg *color*	Sets the background color.
-bl`ack` *color*	Sets the 'black' color used in B/W dithering.
-bw *width*	Sets the border width of the windows.
±cec`map`	Install image's colormap in the *xv color editor* window.
-ceg`eometry` *geom*	Sets the initial position of the *xv color editor*.
±cem`ap`	Automatically open the *xv color editor* on startup.
-cga`mma` *rv gv bv*	Sets *Red, Green,* and *Blue* graphs to specified gamma values.
-cge`ometry` *geom*	Sets the initial position of the *xv controls* window.
±cle`ar`	Clears out the root window and exits.
±clo`se`	Automatically close *xv* windows when image window is iconified.
±cma`p`	Automatically open the *xv controls* window on startup.
-cmtg`eometry` *geom*	Sets the initial position and size of the *xv comments* window.
±cmtm`ap`	Automatically open the *xv comments* window on startup.
-cr`op` *x y w h*	Automatically do a **Crop** command when an image is loaded.
-cu`rsor` *curs*	Sets the cursor used in the image window.
-D`EBUG` *level*	Displays debugging information.
-dir *directory*	Sets the initial directory for the *visual schnauzer* and *xv load* windows.
-dis`play` *disp*	Specifies which X display to use.
±dit`her`	Automatically do a **Dither** command when an image is loaded.
-dr`ift` *dx dy*	Kludge to keep image window from 'drifting' around screen.
-e`xpand` *exp*	Automatically expand or contract images by the given factor, or factors.
-fg *color*	Sets the foreground color.
±fi`xed`	Sets 'fixed aspect ratio' mode.
-fl`ist` *name*	Loads a list of image filenames from a file.

-ga`mma` *val*	Sets the *Intensity* graph to the given gamma value.
-ge`ometry` *geom*	Specifies initial size and position of image window.
-gr`abdelay` *secs*	Sets time to wait before starting **Grab** command.
-gsd`ev` *str*	Type of file *ghostscript* should produce.
-gsg`eom` *geom*	Size of page *ghostscript* should produce, in 72nds of an inch.
-gsr`es` *res*	Resolution of file *ghostscript* should produce, in dpi.
-he`lp`	Prints a list of valid command-line options.
±hf`lip`	Issue a 'horizontal flip' command when image is loaded.
-hi *color*	Sets the 'highlight' color used by the buttons.
±his`t`	Issue a **HistEq** command when image is loaded.
±hs`v`	Puts the colormap editing dials into HSV mode.
-icg`eometry` *geom*	Specifies position of icon when run in `-iconic` mode.
±ico`nic`	Starts up *xv* with the image window iconified.
-ig`eometry` geom	Specifies initial position of *xv info* window.
±ima`p`	Automatically open *xv info* window on startup.
±lb`rowse`	Keep the *xv load* window open until deliberately closed.
-lo *color*	Sets the 'lowlight' color used by the buttons.
±loa`dclear`	Clear window to avoid 'rainbow' effect on *PseudoColor* displays.
±max	Make the image as large as possible.
±maxp`ect`	Make the image as large as possible, preserving aspect ratio.
-mf`n` *font*	Mono-spaced font used in TextView windows.
±mo`no`	Display all pictures in greyscale.
-na`me` *str*	Set string displayed in image window's titlebar.
-nc`ols` *num*	Specifies maximum number of different colors to use.
±ni`nstall`	Don't 'install' colormaps. Have the WM do it for us.
±nod`ecor`	Remove decorations from *xv image* window frame.
±nof`reecols`	Don't free colors for old image when loading new image.
±nol`imits`	Turn off all 'maximum size' limitations on the image.
±nop`os`	Don't automatically position the *xv* windows.
±noq`check`	Suppress quick-check when doing 24->8 bit algorithms.
±norm	Issue **Norm** command when image is loaded.
±nore`setroot`	Don't clear root when going back to 'window' display mode.
±nos`tat`	Speed up directory changing in load/save windows.
±o`wncmap`	Always use and install a private colormap.

±perfect	Use and install a private colormap if necessary.
±poll	Reload image files if they change.
-preset *set*	Makes preset #*set* the default preset.
-quick24	Use the quickest 24-bit to 8-bit color algorithm.
±quit	Exit after displaying first image.
±random	Show images in random order.
±raw	Issue a **Raw** command when image is loaded.
-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.
-RM	Deletes files listed on command line when *xv* exits.
-rmode *num*	Use specified display mode when using root window.
±root	Display images on root window.
-rotate *deg*	Rotate image when it is loaded.
±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.
±stdcmap	Use Standard Colormap.
-tgeometry *geom*	Initial position and size for TextView window.
±vflip	Automatically do a 'vertical flip' command when image is loaded.
±viewonly	Ignore all user input.
-visual *type*	Use a non-default visual of your X display.
±vsdisable	Disable *Visual Schnauzer*.
-vsgeometry *geom*	Initial size and position for *Visual Schnauzer* windows.
±vsmap	Automatically open a *Visual Schnauzer* window on startup.
±vsperfect	Prevent *Visual Schnauzer* from installing its own private colormap.
-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

Section B.1: Simple Resources

Name	Type	Description
`aspect`	*string*	Sets the default aspect ratio used by the **Aspect** command.
`2xlimit`	*boolean*	Allow image window to be twice the size of the screen.
`auto4x3`	*boolean*	Issue **4x3** command when an image is loaded.
`autoApply`	*boolean*	Sets initial setting of **Auto-Apply HSV/RGB mods** checkbox in *xv color editor* window.
`autoClose`	*boolean*	Automatically close *xv* windows when image window is iconified.
`autoCrop`	*boolean*	Issue an **AutoCrop** command when an image is loaded.
`autoDither`	*boolean*	Automatically do a **Dither** command when an image is loaded.
`autoHFlip`	*boolean*	Issue a 'horizontal flip' command when image is loaded.
`autoHistEq`	*boolean*	Issue a **HistEq** command when image is loaded.
`autoNorm`	*boolean*	Issue **Norm** command when image is loaded.
`autoRaw`	*boolean*	Issue a **Raw** command when image is loaded.
`autoReset`	*boolean*	Sets initial setting of **Auto-reset on new image** checkbox in *xv color editor* window.
`autoRotate`	*integer*	Rotate image when it is loaded.
`autoSmooth`	*boolean*	Automatically **Smooth** image on initial load.
`autoVFlip`	*boolean*	Automatically do a 'vertical flip' command when image is loaded.
`background`	*string*	Sets the background color.
`best24`	*boolean*	Use the 'best' (read: slowest) 24-bit to 8-bit color algorithm.
`black`	*string*	Sets the 'black' color used in B/W dithering.
`borderWidth`	*integer*	Sets the border width of the windows.
`ceditGeometry`	*string*	Sets the initial position of the *xv color editor*.
`ceditMap`	*boolean*	Automatically open the *xv color editor* on startup.
`ceditColorMap`	*boolean*	Install image's colormap in the *xv color editor* window.
`clearOnLoad`	*boolean*	Clear window to avoid 'rainbow' effect on *PseudoColor* displays.
`commentGeometry`	*string*	Sets the initial position and size of the *xv comments* window.
`commentMap`	*boolean*	Automatically open the *xv comments* window on startup.
`ctrlGeometry`	*string*	Sets the initial position of the *xv controls* window.
`ctrlMap`	*boolean*	Automatically open the *xv controls* window on startup.
`cursor`	*integer*	Sets the cursor used in the image window.

`defaultPreset`	*integer*	Makes specified preset number the default preset.
`displayMods`	*boolean*	Sets initial setting of **Display with HSV/RGB mods** checkbox in *xv color editor* window.
`dragApply`	*boolean*	Sets initial setting of **Auto-Apply while dragging** checkbox in *xv color editor* window.
`driftKludge`	*string*	Kludge to keep image window from ‘drifting’ around screen.
`expand`	*string*	Automatically expand or contract images by the given factor.
`fileList`	*string*	Loads a list of image filenames from a file.
`fixed`	*boolean*	Sets ‘fixed aspect ratio’ mode.
`force8`	*boolean*	Lock *xv* into **8-bit Mode**.
`force24`	*boolean*	Lock *xv* into **24-bit Mode**.
`foreground`	*string*	Sets the foreground color.
`geometry`	*string*	Specifies initial size and position of image window.
`gsDevice`	*string*	Type of file *ghostscript* should produce.
`gsGeometry`	*string*	Size of page *ghostscript* should produce, in 72nds of an inch.
`gsResolution`	*integer*	Resolution of file *ghostscript* should produce, in dpi.
`hsvMode`	*boolean*	Puts the colormap editing dials into HSV mode.
`highlight`	*string*	Sets the ‘highlight’ color used by the buttons.
`iconGeometry`	*string*	Specifies position of icon when run in `-iconic` mode.
`iconic`	*boolean*	Starts up *xv* with the image window iconified.
`infoGeometry`	*string*	Specifies initial position of *xv info* window.
`infoMap`	*boolean*	Automatically open *xv info* window on startup.
`loadBrowse`	*boolean*	Keep the *xv load* window open until deliberately closed.
`lowlight`	*string*	Sets the ‘lowlight’ color used by the buttons.
`mono`	*boolean*	Display all pictures in greyscale.
`monofont`	*string*	Mono-spaced font used in TextView windows.
`ncols`	*integer*	Specifies maximum number of different colors to use.
`ninstall`	*boolean*	Don’t ‘install’ colormaps. Have the WM do it for us.
`nodecor`	*boolean*	Don’t put decorations on *xv image* window frame.
`nolimits`	*boolean*	Turn off all ‘maximum size’ limitations on the image.
`nopos`	*boolean*	Don’t automatically position the *xv* windows.
`noqcheck`	*boolean*	Suppress quick-check when doing 24->8 bit algorithms.
`nostat`	*boolean*	Speed up directory changing in load/save windows.
`ownCmap`	*boolean*	Always use and install a private colormap.
`perfect`	*boolean*	Use and install a private colormap if necessary.
`print`	*string*	Sets initial setting of **Orientation** in *xv postscript* window.

`psorient`	*string*	Sets initial setting of **Orientation** in *xv postscript* window. (`landscape portrait`)
`pspaper`	*string*	Sets initial setting of **Paper Size** in *xv postscript* window. (`8.5x11 8.5x14 11x17 4x5 35mm a3 a4 b5`)
`psres`	*integer*	Sets initial setting of **Resolution** in *xv postscript* window.
`pscompress`	*boolean*	Sets initial setting of **compress** button in *xv postscript* window.
`pspreview`	*boolean*	Sets initial setting of **preview** button in *xv postscript* window.
`quick24`	*boolean*	Use the quickest 24-bit to 8-bit color algorithm.
`resetroot`	*boolean*	Clear root when going back to 'window' display mode.
`reverse`	*boolean*	Reverses RGB values when image is loaded.
`rootBackground`	*string*	Root background color, used on some root display modes.
`rootForeground`	*string*	Root foreground color, used on some root display modes.
`rootMode`	*integer*	Use specified display mode when using root window.
`rwColor`	*boolean*	Use read/write color cells for faster color editing.
`saveNormal`	*boolean*	Sets default setting of **Save at normal size** checkbox in *xv save* window.
`searchDirectory`	*string*	Sets the initial directory for the *visual schnauzer* and *xv load* windows.
`textviewGeometry`	*string*	Initial position and size for TextView window.
`useStdCmap`	*boolean*	Use Standard Colormap.
`visual`	*string*	Use a non-default visual of your X display.
`vsDisable`	*boolean*	Disable *Visual Schnauzer*.
`vsGeometry`	*string*	Initial size and position for *Visual Schnauzer* windows.
`vsMap`	*boolean*	Automatically open a *Visual Schnauzer* window on startup.
`vsPerfect`	*boolean*	Prevent *Visual Schnauzer* from installing its own private colormap.
`white`	*string*	Sets the 'white' color used in B/W stippling.

Section B.2: Color Editor Resources:

The *xv color editor* resources take the general form:

`xv`.*state*.*item*: val

where *state* is: '`default`', '`preset1`', '`preset2`', '`preset3`', or '`preset4`'

and *item* is: '`huemap1`' '`huemap2`', '`huemap3`', '`huemap4`', '`huemap5`', '`huemap6`', '`whtmap`', '`satval`', '`igraf`', '`rgraf`', '`ggraf`', or '`bgraf`'

Section B.3: Pad Command Resources

You can change the contents of the 'shortcut' menus in the **Pad** command dialog box using the following resource format:

`xv.pad.`*`menu`*`.`*`slot`*

Where *menu* is one of 'color', 'bggen', 'load', and *slot* is 'val0' .. 'val9', or 'name0' .. 'name9'. The 'val's are string resources that hold the actual 'shortcut text' that is entered when you select this menu choice. The 'name's are optional, and are the short descriptions listed on the menus.

For instance, to add the items 'yellow', and 'blue' to the 'color' menu, stick these lines in your `.Xdefaults` (or `.Xresources`) file:

```
xv.pad.color.name0:   yellow
xv.pad.color.val0:    255 255 0
xv.pad.color.name1:   blue
xv.pad.color.val1:    0 0 255
```

Appendix C: Mouse and Keyboard Usage

Section C.1: Mouse Usage in the Image Window

Button1	draws a selection rectangle
Button2	display pixel values, measure distance, picks drawing color
Button3	opens or closes the *xv controls* window
<Ctrl> Button1	**Zoom In**
<Ctrl> Button2	**Pan** (while zoomed in)
<Ctrl> Button3	**Zoom Out**
<Shift> Button1	draws a square selection rectangle
<Shift> Button2	freehand drawing tool
<Shift> <Ctrl> Button2	line drawing tool
<Shift> Button3	smudging tool

Section C.1.1 Mouse Usage in Selection Rectangle

Button1	moves selection rectangle
<Shift> Button1	moves selection, constrains motion
Button2	'drag-and drop' cut and paste
<Shift> Button2	'drag and drop' cut and paste, constrains motion
<Ctrl> Button2	'drag and drop' copy and paste
<Shift> <Ctrl> Button2	'drag and drop' copy and paste, constrains motion

Section C.2: Normal Keyboard Equivalents

The following keyboard equivalents can be used in most *xv* windows, including the *xv image, xv controls, xv color editor,* and so on, but *not* the *xv visual schnauzer*.

Key	**Description**
<Tab> *or* **<Space>**	**Next**
<Return>	reload currently selected image file
<Delete> *or* **<Backspace>**	**Prev**
<Ctrl> l	**Load**
<Ctrl> s	**Save**
<Ctrl> p	**Print**
<Ctrl> d	**Delete**
<Ctrl> q *or* **q**	**Quit**
<Meta> x	**Cut**
<Meta> c	**Copy**
<Meta> v	**Paste**
<Meta> d	**Clear**
n	reset image to **Normal** size
m	**Max**

M	**Maxpect**
>	**Double Size**
<	**Half Size**
<Period>	**10% Larger**
<Comma>	**10% Smaller**
S	**Set Size**
a	**Re-Aspect**
4	**4x3**
I	**Int. Expand**
t	turn image 90° clockwise
T	turn image 90° counter-clockwise
h	flip image horizontally
v	flip image vertically
P	**Pad**
A	**Text Annotation**
c	**Crop**
u	**UnCrop**
C	**AutoCrop**
r	**Raw**
d	**Dither**
s	**Smooth**
<Meta> 8	toggle between **8-bit** and **24-bit** modes
<Ctrl> v *or* **V**	**Visual Schnauzer**
e	**Color Editor**
i	**Image Info**
<Ctrl> c	**Image Comments**
<Ctrl> t	**Text View**
<Ctrl> g	**Grab**
<Ctrl> a	**About XV**
<Meta> b	**Blur**
<Meta> s	**Sharpen**
<Meta> e	**Edge Detection**
<Meta> m	**Emboss**
<Meta> o	**Oil Paint**
<Meta> B	**Blend**
<Meta> t	**Copy Rotate**
<Meta> T	**Clear Rotate**
<Meta> p	**Pixelize**
<Meta> S	**Spread**
<Meta> r *or* **R** *or* **<Meta> 0**	**Reset** command in *xv color editor*
<Meta> 1	select preset #1 in *xv color editor*
<Meta> 2	select preset #2 in *xv color editor*
<Meta> 3	select preset #3 in *xv color editor*
<Meta> 4	select preset #4 in *xv color editor*
<Meta> a	**Apply** command in *xv color editor*

Section C.2.1: Image Window Keys

The following keys can *only* be used inside the image window.

<Ctrl> <Up>	crops 1 pixel off the bottom of the image
<Ctrl> <Down>	crops 1 pixel off the top of the image
<Ctrl> <Left>	crops 1 pixel off the right side of the image
<Ctrl> <Right>	crops 1 pixel off the left side of the image

If you're viewing a multi-page document:

p	opens a 'go to page #' dialog box
<PageUp>	go to previous page
<PageDown>	go to next page

If a selection rectangle is active:

<Up>	moves selection rectangle up 1 pixel
<Down>	moves selection rectangle down 1 pixel
<Left>	moves selection rectangle left 1 pixel
<Right>	moves selection rectangle right 1 pixel
<Shift> <Up>	shrink rectangle vertically by 1 pixel
<Shift> <Down>	expand rectangle vertically by 1 pixel
<Shift> <Left>	shrink rectangle horizontally by 1 pixel
<Shift> <Right>	shrink rectangle horizontally by 1 pixel

Section C.3: Visual Schnauzer Keys

The following keys can only be used in the *visual schnauzer* windows.

<Ctrl> d	**Delete File(s)**
<Ctrl> n	**New Directory**
<Ctrl> r	**Rename File**
<Ctrl> s	**ReScan Directory**
<Ctrl> w	**Open New Window**
<Ctrl> u	**Update Icons**
<Ctrl> g	**Generate Icons**
<Ctrl> a	**Select All Files**
<Ctrl> t	**Text View**
<Ctrl> q	**Quit XV**
<Ctrl> c	**Change Directory**
<Ctrl> f	**Select Filenames**
<Ctrl> e	**Recursive Update**
<Esc>	**Close Window**
<Return>	load currently selected file(s)
<Space>	load next file
<Shift> <Space>	load next file, keeping previous file(s) selected
<Backspace>	load previous file

Appendix D: RGB & HSV Colorspaces

Both the RGB and HSV Colorspaces define a method 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 saturated colors will be bold and vibrant, 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 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 typical 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. Section 3.3.2: Color Allocation Commands" for further 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, any colors still unallocated are simply mapped into the ‘nearest’ colors that *were* allocated in Phase 1 or Phase 2.

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. It is, however, up to your particular window manager to decide how multiple colormaps are handled..

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.

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, rather than free all the colors and reallocate new *different* colors.

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 '`AllocROColors()`' 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 a completely original algorithm designed 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' can be determined intuitively 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. Additional colors 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 (as *red* will wind up being 'approximated' by 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. This is where it gets somewhat subjective. 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 will be 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. However, if 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 unused 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. Distance is defined by the traditional 'Euclidean' formula:

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

where r1,g1,b1 are the RGB components of one color, and r2,g2,b2 are the RGB components of another color. d is the computed 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 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, and certainly different fine-tunings of the color picking order will make some pictures look better, and other pictures look worse. 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 read PCX files, but it doesn't write them. Likewise, *xv* traditionally could only write PostScript files, but couldn't read them. (And technically, it still doesn't.)

For the purposes of this example, 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 or 24-bit RGB images. If you're loading an 8-bit colormapped image, such as a GIF image, no problem. If you're loading an 8-or-fewer-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.

Make a copy of `xvpbm.c`, calling it something appropriate. For the rest of this appendix, mentally replace the string '`xvpbm.c`' with the name of your new file.

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, and "`xvpbm.c`" to the end of the "`SRCS1 = ...`" 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* to load, and a pointer to a `PICINFO` structure, like so:

```
/*******************************************/
int LoadPBM(fname, pinfo)
char *fname;   PICINFO *pinfo;
/*******************************************/
```

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. The same goes for pipes and compressed files. Your `Load()` routine is guaranteed that it will be reading from a real file that appears to be in your file format, not a stream. This lets you use routines such as `fseek()`, and such.

The `pinfo` argument is a pointer to a `PICINFO` structure. This structure is used to hold the complete set of information associated with the image that will be loaded. When your `Load()` routine is called, the fields in this structure will all be zeroed-out. It is your function's responsibility to load up the structure appropriately, and completely. The structure is defined as:

```
/* info structure filled in by the LoadXXX() image reading routines */
typedef struct { byte *pic;                  /* image data */
                 int   w, h;                 /* size */
                 int   type;                 /* PIC8 or PIC24 */
                 byte  r[256],g[256],b[256]; /* colormap, if PIC8 */
                   int   normw, normh;         /* normal size of image.
                                                  normally == w,h except when
                                                   doing quick load for icons
*/
                 int   frmType;              /* def. Format type to save in */
                 int   colType;              /* def. Color type to save in */
                 char  fullInfo[128];        /* Format: field in info box */
                 char  shrtInfo[128];        /* short format info */
                 char *comment;              /* comment text */

                 int   numpages;             /* # of page files, if >1 */
                 char  pagebname[64];        /* basename of page files */
} PICINFO;
```

The `Load()` function should return '1' on success, '0' on failure.

All other information is communicated using the `PICINFO` structure. The fields should be setup as follows:

`byte *pic;`

This is an array of bytes which holds the returned image. The array is `malloc()`'d by the `Load()` routine. The array should be `w*h` bytes long (for an 8-bit colormapped image) or `w*h*3` bytes long (for a 24-bit RGB image). For an 8-bit image, there is one byte per pixel, which serves as an index into the returned colormap (see below). For a 24-bit image, there are three bytes per pixel, in red, green blue, order. In either case, pixels start at the top-left corner, and proceed in normal scan-line order. There is no padding of any sort at the end of a scan line.

`int w, h;`

These variables specify the width and height (in pixels) of the image that has been loaded.

`int type;`

This variable is used to tell the calling routine whether the loaded image is an 8-bit image or a 24-bit image. It *must* be set equal to `PIC8` or `PIC24`, whichever one is appropriate. These constants are defined in '`xv.h`'.

`byte r[256], g[256], b[256];`

If the returned image is an 8-bit image, you must load up these variables with the colormap for the 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 `pic` need to be set. For example, if `pic` is known to be a B/W bitmap with pixel values of 0 and 1, you would only have to set entries '0' and '1' of the `r,g,b` arrays.

On the other hand, if the returned image is a 24-bit image, the `r,g,b` arrays are ignored, and you do not have to do anything with them.

`int normw, normh;`

These specify the 'normal' size of the image. Normally, they are equal to `w` and `h`, respectively. The only exception is when doing a 'quick' load for icon generation, in which case it may be possible to read a 'reduced' version of the image, sufficient for generating the tiny icon files. In such a case, `w` and `h` would reflect the (reduced) size of the image returned, and `normw` and `normh` would reflect the 'normal' image size, for use in the comments displayed in the *xv visual schnauzer*. Currently only the `LoadJFIF()` function in `xvjpeg.c` actually does this.

```
int frmType;
```

This lets you specify the *Format* type to automatically select when you **Save** a file. As such, this is only relevant if you intend to have *xv* write your image format as well as read it. If you are only writing an image loader, you should set this field to '`-1`'. On the other hand, if you *do* intend to write a `Write()` function for your format, you should edit `xv.h`, find the `F_*` format definitions, and add one for your format. See `xvpcx.c` for an example of a load-only format, or `xvpbm.c` for a load-and-save format.

```
int colType;
```

Used to determine which *Colors* setting should be used by default when you save a file. Since *xv* will use this setting no matter *what* format you're using, you must fill this field in appropriately regardless of whether or not you plan to have a `Write()` function for your format. This field should be set to `F_FULLCOLOR` for any type of color image, `F_GREYSCALE` for greyscale images, and `F_BWDITHER` for black-and-white 1-bit images. If in doubt, `F_FULLCOLOR` is always a safe choice, though it'd be nice if your module 'does the right thing'. (For instance if you read colormapped images, you should check to see if the colormap consists only of shades of grey, and set `F_GREYSCALE` if it does.)

```
char fullInfo[128];
```

This string will be shown in the *Format* field of the *xv info* window. It should be set to something like this:

```
   Color PPM, raw format  (12345 bytes)
```

```
char shrtInfo[128];
```

A 'short' version of the info string. This gets displayed in the *info* line at the bottom of the *xv controls* and *xv info* windows when the image is loaded. It should look like this:

```
   512x400 PPM.
```

```
char *comment;
```

If your image file format supports some sort of comment field, and you find one in the file, you should `malloc()` a pointer to a null-terminated string and copy any and all comments into this field. If there are multiple comments in a file, you should concatenate them together to form one long string. This string *MUST* be null-terminated, as *xv* will expect to be able to use `strlen()` on it, and possibly other 'string' functions.

```
int numpages;
char pagebname[64];
```

These two fields will only be used if your are writing a `Load()` function for a format that may have multiple images per file. If your format only ever has a single image per file, you don't have to worry about (or do anything with) these two fields.

On the other hand, if your format *does* do multiple images per file, *and* the current file has more than one image in it, then what your program should do is *split* the multi-image file up into a temporary collection of single-image files, which should probably live in `/tmp` or something. Once you've done so, you should return the number of files created in `numpages`, and the 'base' filename of the page files in `pagebname`. The files created should have a common 'base', with the page number appended. (e.g., "/tmp/xvpg12345a.1", "/tmp/xvpg12345a.2", etc., where "/tmp/xvpg12345a." is the base filename (created by the `mktemp()` system function)) You should also load the first file and return its image in the normal way.

See the `LoadPS()` function in `xvps.c` for a complete example of how this is done. Also, note that if your format supports multiple image per file, you should also pass in a '`quick`' parameter, which will tell your function to only load the first 'page' of the file. This is used by the *visual schnauzer*, which needs to load images when it generates icon files. To speed things up, the *schnauzer* tells the `Load()` function to only load the first page, as that's all it need to generate the icon file.

Section G.1.1: Error Handling

Non-fatal errors in your `Load()` routine should be handled by calling the function `SetISTR(ISTR_WARNING, "%s: %s", bname, err)`, and returning a zero value. Where *bname* is the 'simple' name of your file (which can be obtained using `BaseName()` function in `xvmisc.c`), and *err* should be an appropriate error string.

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, as it means *xv* is likely to run out of memory in the near future anyhow.

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.

Warnings (such as 'truncated file') that may need to be noted can be handled by calling `SetISTR()` as shown above, but continuing to return '1' from the `Load()` routine, signifying success.

Also, if your load routine fails for *any reason*, it is your responsibility to `free()` any pointers allocated (such as the *pic* field and the *comment* field, and return NULL in these fields). Otherwise, there'll be memory leaks whenever an image load fails.

Section G.1.2: Hooking it up to *xv*

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

Edit `xv.h` and add a function prototype for any global functions you've written (presumably just `LoadPBM()` in this case). Follow the style used for the other `Load*()` function declarations.

Find the `RFT_*` definitions and tack one on the end for your format (e.g., `RFT_PBM`). This is a list of values that '`ReadFileType()`' can return. We'll be working on that soon enough.

Edit `xv.c`:

1. Tell the `ReadFileType()` routine about your format. Add an 'else-if' case that determines if the file in question is in your format. Note that it must be possible to uniquely identify your format by reading the first 16 characters (or so) of the file. If your file format *doesn't* have some sort of *magic number*, you won't be able to conveniently hook it into *xv*, though you can certainly come up with some sort of kludge...

2. Tell the `ReadPicFile()` routine about your format. Add another `case` for your format type, and have it call your `Load()` routine with the appropriate arguments.

3. Hook your file up into the *visual schnauzer*. Edit the file `xvbrowse.c`

   - The first thing you have to do is create a 'generic' icon for your file format. Copy one of the existing ones (such as '`bits/br_pbm.xbm`') to get the size and the general 'look' correct.

   - `#include` this icon at the top of the file.

   - Add an appropriately-named `BF_*` definition to the end of the list, and increase `BF_MAX` appropriately.

   - Have the icon pixmap created in the `CreateBrowse()` function, by doing something like this:

```
bfIcons[BF_PBM] = MakePix1(br->win, br_pbm_bits,
                           br_pbm_width, br_pbm_height);
```

- Hook your format into the `scanFile()` function. Find the following code:

```
switch (filetype) {
case RFT_GIF:      bf->ftype = BF_GIF;      break;
case RFT_PM:       bf->ftype = BF_PM;       break;
```

etc...

And add a case for your format. (To map `RFT_*` values into their corresponding `BF_*` values.)

- Hook your format into the `genIcon()` function. Find the following code:

```
sprintf(str, "%dx%d ", pinfo.w, pinfo.h);
switch (filetype) {
  case RFT_GIF:      if (strstr(pinfo.shrtInfo, "GIF89"))
                       strcat(str,"GIF89 file");
                     else
                       strcat(str,"GIF87 file");
  break;
  case RFT_PM:       strcat(str,"PM file");

  break;
```

etc...

And add a case for your format. This generates an appropriate info string that gets put in the icon files maintained by the *visual schnauzer* (and displayed whenever you click on an icon in the *schnauzer* window).

That should do it. Consult the files `xv.h`, `xv.c`, `xvbrowse.c`, and `xvpbm.c` for any further specifics.

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 *either* 8-bit colormapped images *or* 24-bit RGB images. Your `Write()` routine must be prepared to take either sort of image, and convert it (if necessary) to the image format that your file format dictates.

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

- Make a copy of `xvpbm.c`, calling it something appropriate. For the rest of this appendix, mentally replace the string '`xvpbm.c`' with the name of your new file.
- 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, and "`xvpbm.c`" to the end of the "`SRCS1 = ...`" 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 writing the file. The function is called with a large number of arguments, described below. The function should return '0' if everything succeeded, and '-1' on failure.

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

```
        byte *pic;
        int   ptype, w,h;
        byte *rmap, *gmap, *bmap;
        int   numcols, colorstyle, raw;
        char *comment;
   /*******************************************/
```

`file *fp;`

This is a pointer to an already-`fopen()`'d stream. Your function should neither open nor close this stream, as that all gets handled elsewhere in `xvdir.c`.

`byte *pic;`

This points to the image data that will be written. In the case of a `PIC8` image, `pic` will point to a `w*h` long array of bytes, one byte per pixel, starting at the top-left, and proceeding in normal scan-line order. There is no padding of any sort at the end of the lines.

In the case of a `PIC24` image, `pic` will point to a `w*h*3` long array of bytes. There are three bytes per pixel, in red, green, blue order. The pixels start at the top-left, and proceed in normal scan line order. There is no padding of any sort at the end of the lines.

`int ptype, w, h;`

These variables describe the format of `pic`. `ptype` can be set to either `PIC8` or `PIC24`. `w` and `h` are the width and height of the image, in pixels.

```
byte *rmap, *gmap, *bmap;
int   numcols;
```

These pointers point to the colormap associated with the image. They are only relevant when *ptype* is PIC8, meaning that *pic* is an 8-bit per pixel colormapped image. These arrays will each be `numcols` entries long, with a maximum length of 256. Do not attempt to access entries >= `numcols`, as the colormaps are *not necessarily* 256 entries long. You are guaranteed that pixel values found in `pic` will be within the range [0..numcols-1], so you don't have to check each pixel value. Also, do not attempt to access these arrays at all if `ptype` is PIC24, as these pointers will probably be NULL in that case.

`int colorstyle;`

The *Colors* choice selected in the *xv save* window. It can be either `F_FULLCOLOR`, `F_GREYSCALE`, or `F_BWDITHER`. It will *not* be `F_REDUCED`. If the user selects **Reduced Color** in the *xv save* window, the appropriate image will be computed, and you'll be given that image, and *colorstyle* will be set to `F_FULLCOLOR`.

Likewise, if the user has selected **B/W Dithered** in the *xv save* window, an appropriate black-and-white image will have been generated before your `Write()` routine is called, so you won't have to worry about that. Such an image will be a `PIC8` image, with a 2-entry colormap. It is up to you to decide which of the two colors should be written as *black*, and which should be written as *white*. You should do this by comparing the values of `MONO(rmap[0],gmap[0],bmap[0])` and `MONO(rmap[1],gmap[1],bmap[1])`. Whichever value is smaller is the darker of the two, and should be written as *black*.

`int raw;`

This is a value passed in specifically for the `WritePBM()` function, as PBM has two closely-related subformats (raw, and ascii) both of which are written by this one function. Your function won't need this, nor should it be passed in to your function.

`char *comment;`

This will point to a zero-terminated character string which contains the comments that should be written into the image file. Note that this string can be of *any* length, and it may contain any number of lines (separated by '\n' characters). If your image format supports comments in the file, you should write this information to the file. If it doesn't, you should just ignore this variable. Also, this variable may be NULL, (signifying 'no comments'), in which case it should not be used.

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.)

Write the function as you deem appropriate. See xvpbm.c for an example of a Write() routine that writes different formats for 1-bit per pixel images, 8-bit per pixel images, and 24-bit per pixel images, based on ptype and colorstyle.

Note: If your file format can only handle 8-bit images, and ptype is set to PIC24, you will have to call Conv24to8() to convert the 24-bit image into an 8-bit colormapped image that you can write to the file. See xvgifwr.c for an example of how this is done.

That done, edit 'xv.h' and add a function declaration for your new function. Search for 'WritePBM()' in the file for a sample declaration to copy.

Also find the block that begins with:

```
#define F_GIF       0
#define F_JPEG      ( 0 + F_JPGINC)
```

and add a definition for your format. Note that it'll be easiest to tack it on at the end.

These numbers *must* be contiguous, as they are used as indices into the fmtMB menu in xvdir.c.

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

Add another format name, in the appropriate position, to the saveFormats[] string array.

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

```
switch (fmt) {
  case F_GIF:
      rv = WriteGIF(fp, thepic, ptype, w, h, rp, gp, bp, nc, col,
picComments);
  break;
  case F_PM:
      rv = WritePM (fp, thepic, ptype, w, h, rp, gp, bp, nc, col,
picComments);
  break;
```

and add a case for your function.

Finally, if your format has a common 3 or 4 letter filename suffix (like, ".gif", ".jpg", etc.), you should modify the changeSuffix() routine in xvdir.c so that it recognizes your suffix, and puts your suffix on when someone selects your format.

And It's just that easy!

Section G.2.1: Writing Complex Formats

Okay, maybe it's not *that* easy...

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. You'll also have to change the way your Write() function gets called (as it will now get called from your pop-up dialog box). (Though, if you only feel like doing a quick hack, you can probably just use the `GetStrPopUp()` function to get a one-line character string from the user, and avoid the complication of writing your own dialog box.)

This is not recommended for anyone who doesn't understand Xlib programming. Frankly, it's not recommended for those who *do*, either, but they at least stand some chance of success.

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.

Appendix H: Adding Algorithms to *xv*

With the addition of the **Algorithms** menu in the *xv controls* window, *xv* can now perform standard image-processing algorithms. However, I'm not really into the whole image-processing *scene*, so I've only implemented a few algorithms.

Please! Feel free to add your own algorithms, it's easy, and if you'd care to donate them, they may find their way into future official releases of *xv*, and eternal fame and glory will be yours, in the form of a credit in the "Hall of Fame" listing..

Section H.1: Adding an Algorithm

For the purposes of this example, I'll be adding a new algorithm called 'Noise' which will simply add (or subtract) a small random amount from each pixel in the image. I can't see that this would be a very useful algorithm (which is why it's not already *in xv*), but then again, what do I know about such things...

Edit `xv.h`, and find the block that starts with:

```
#define ALG_NONE  0
#define ALG_SEP1  1  /* separator */
#define ALG_BLUR3 2
```

and add an additional definition at the end of the list (right before `ALG_MAX`) for your algorithm. Don't forget to increment `ALG_MAX` to reflect the additional algorithm:

```
#define ALG_TINF  6
#define ALG_OIL   7
#define ALG_NOISE 8
#define ALG_MAX   9
```

Edit `xvctrl.c`, and find where the array `algMList[]` is initialized. Add a string for your new algorithm. The string's position in the list must match the number that you assigned to the `ALG_*` value in `xv.h`:

```
static char *algMList[] = { "Undo All",
                            MBSEP,
                            "Blur (3x3)",
                            "Blur (7x7)",
                            "Edge Detection",
                            "Emboss",
                            "Oil Painting",
                            "Add Noise"};
```

Edit `xvalg.c`, and find the `DoAlg()` function. This function is called with an `ALG_*` value whenever something is selected from the **Algorithms** menu. Add a case for the new `ALG_NOISE` value, and have it call your top-level function, with no parameters:

```
  case ALG_TINF:  EdgeDetect(1);  break;
  case ALG_OIL:   OilPaint();     break;
  case ALG_NOISE: Noise();        break;
  }
```

Write your top-level function:

```
/************************/
static void Noise()
{
  byte *pic24, *tmpPic;
  /* turn on flapping fish cursor */
  WaitCursor();

  /* mention progress... */
  SetISTR(ISTR_INFO, "Running Noise algorithm...");

  /* generates a 24-bit version of pic, if necessary.
     also generates a w*h*3 buffer (tmpPic) to hold intermediate results */
  if (start24bitAlg(&pic24, &tmpPic)) return;
  /* do the noise algorithm */
  doNoise(pic24, pWIDE,pHIGH, tmpPic);

  /* if we're in PIC8 mode, convert pic24 back to PIC8. free pic24 & tmppic */
  end24bitAlg(pic24, tmpPic);
}
```

Now write the function that does the work of your algorithm. It will be passed a 24-bit RGB source image *srcpic*, its dimensions *w,h,* and a destination 24-bit image *dstpic* of the same size. If your algorithm is normally meant to be run on greyscale images (as so many image algorithms are), you should simply run it separately for each of the *Red, Green,* and *Blue* planes, and glue the results back together at the end of the algorithm.

```
/************************/
static void doNoise(srcpic, w, h, dstpic)
  byte *srcpic, *dstpic;
  int   w, h;
{
  byte *sp, *dp;
  int   x,y,newr,newg,newb;

  printUTime("start of doNoise");   /* print timing info */

  for (y=0; y<h; y++) {
    if ((y & 15) == 0) WaitCursor();

    sp = srcpic + y*w*3;      /* position sp,dp at start of line #y */
    dp = dstpic + y*w*3;
    for (x=0; x<w; x++) {
      newr = sp[0] + (random()&0x3f)-0x20;  /* add noise to red   component */
      newg = sp[1] + (random()&0x3f)-0x20;  /* add noise to green component */
      newb = sp[2] + (random()&0x3f)-0x20;  /* add noise to blue  component */
      RANGE(newr, 0, 255);   /* clip values to range[0..255] inclusive */
      RANGE(newg, 0, 255);   /* RANGE() is defined in xv.h */
      RANGE(newb, 0, 255);
      dp[0] = (byte) newr;   /* store new values in dstpic */
      dp[1] = (byte) newg;
      dp[2] = (byte) newb;

      sp += 3;  dp += 3;     /* advance to next 3-byte pixel in images */
    }
  }
  printUTime("end of doConvolv");
}
```

Note that this algorithm is written in about as non-optimal a way as possible, for the sake of clarity.

Also note that if you define `TIMING_TEST` at the beginning of `xvalg.c`, it will turn on code that will let you measure the CPU time your algorithm requires. Once you have a working algorithm, you may find this useful if you wish to try to optimize your algorithm for increased performance.

And that's all there is to it!

1

2

4

8

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z