GREEN:
SEGMENTATION OF AN AERIAL VIDEO RECORDING FOR TREE COUNTING

BY

PAUL GREGORY RADJA

B.S., University of Illinois, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois
ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my thesis advisor, Dr. R. Uribe, for his assistance and guidance in writing this thesis. I owe the creation and the body of this research to both Dr. Kevin Warren and Professor Brian Orland, both of whom have given me countless ideas and numerous hours of their time. Additionally, I would like to thank Minako Hashimoto, James Kuch and Larissa Larsen for reading the several revisions of this manuscript.

I would like to thank Kimberlyn de Buhr and my family for allowing me to be absent from their lives while I pursue my academic goals.

Most importantly, I would like to express gratitude to my parents who have picked me up when I have fallen and have taught me the importance of an education.
# TABLE OF CONTENTS

1. INTRODUCTION ............................................................................................. 1

2. HISTORY ............................................................................................................ 2
   2.1 Procedure 1. Chromaticity Filter ..................................................... 3

3. IMAGE ENHANCEMENT ............................................................................. 5
   3.1 Procedure 2. Green Filter .................................................................. 6
   3.2 Procedure 3. Mask Filter ................................................................... 7
   3.3 Procedure 4. Black and White Filter .............................................. 8
   3.4 Procedure 5. Variance Filter ............................................................. 9

4. IMAGE SEGMENTATION............................................................................. 10
   4.1 Procedure 6. Flood Filter .................................................................. 12

5. TREE COUNTING ........................................................................................... 15

6. CONCLUSIONS ............................................................................................... 17

APPENDIX A. ADDITIONAL FEATURES ................................................. 18
   A.1 Additional Tools ............................................................................... 18
   A.2 Additional Filters .............................................................................. 19
   A.3 Additional Image Enhancements .................................................... 20

APPENDIX B. SPECIFIC RESULTS ............................................................... 22

APPENDIX C. PROGRAM LISTING ............................................................ 23
   C.1 Green .................................................................................................... 23
   C.2 Tools ..................................................................................................... 71

REFERENCES .................................................................................................... 104
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Original Image</td>
<td>5</td>
</tr>
<tr>
<td>2. Output of Chromaticity Filter</td>
<td>5</td>
</tr>
<tr>
<td>3. Output of Green Filter</td>
<td>6</td>
</tr>
<tr>
<td>4. Output of Mask Filter</td>
<td>7</td>
</tr>
<tr>
<td>5. Output of Black and White Filter</td>
<td>8</td>
</tr>
<tr>
<td>6. Output of Variance Filter</td>
<td>9</td>
</tr>
<tr>
<td>7. Output of Flood Filter</td>
<td>11</td>
</tr>
<tr>
<td>8. Output of Tree Counting</td>
<td>16</td>
</tr>
<tr>
<td>9. Results of Tree Counting</td>
<td>21</td>
</tr>
<tr>
<td>10. Tree Count Comparison</td>
<td>22</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

The United States' Forest Service has been recording aerial video over many national parks to note deforestation from diseases and insects. This aerial video recording has been viewed by humans for a number of years. With the speed and ability of computers increasing daily, the forest service had requested our help exploring the possibility of using computers to scan these videos. To resolve this problem, a number of image manipulation tools were developed on a UNIX computer system. These tools require only minimal user interaction.
2. HISTORY

The first attempt at solving the problem of locating trees produced the View software package. This program displayed the image in an RGB (red green blue) space, allowing the user to see the spectral distribution of color. The RGB space is a 3-D coordinate system, in which where one axis is related to the color red, one is related to green and the other is related to blue. This coordinate system was chosen because of the ease with which most users could find the color they were searching for. For example, green colors are near the green axis. Darker greens would be nearest the origin and the lighter greens would be farthest from the origin. Another benefit of the RGB coordinate system selection was that most computers use RGB values for storing and displaying images [1]. Using the RGB space then provides a more accurate representation of the data.

To store and retrieve images from the computer system, two standard image formats were selected. These two formats were Targa and Tagged Image. Since these two are standard formats, some programs could easily provide the input for View, or other programs could use the output from View, giving it added flexibility.

Once an initial image was obtained, View allowed the user to select a point within the image. A number of filters could then be applied to this point. After some experimentation, it was found that the filters were helpful in distinguishing clumps of trees from their background. However, shadows and inconsistent luminance throughout the image created a problem when attempting to distinguish trees by color alone.

To combat the irregular luminance, a chromaticity filter was created. This filter would equalize the luminance for all pixels, in order to remove shadows and bright spots. Procedure 1 was used to provide the chromaticity filter.
Quantitatively, the equations use the magnitude of the color point, or the distance the color lies from the origin, and then divides the three color components by this value. This procedure will set the luminance to unity, causing the image to be quite dark. To counteract this problem, each component is then multiplied by 256 to make the image brighter. Now the image has a fixed luminance which facilitates segmentation of the image. Segmentation attempts to divide the image into individual trees.

2.1 Procedure 1. Chromaticity Filter

The chromaticity filter will set all pixels to the same intensity level.

\[
pixel\_magnitude = \sqrt{old\_red^2 + old\_green^2 + old\_blue^2}
\]

new_red = old_red * 256 / pixel_magnitude
new_green = old_green * 256 / pixel_magnitude
new_blue = old_blue * 256 / pixel_magnitude

old_red, old_green and old_blue are the old color components of a pixel.
new_red, new_green and new_blue are the new color components of a pixel.

The chromaticity filter worked well in removing the luminance problem, though it destroyed both subtle nuances in the tree clumps and the possibility of distinguishing individual trees within these clumps. It was apparent that the spatial data as well as the color data had to be considered. As View's tool kit began to grow with the additional tools, the limitation of View's platform (the IBM Personal Computer) became apparent.

View's original development took place on the IBM PC platform which provided a number of obstacles that were not easily surmounted. The primary limitation was memory related. During the development of the IBM PC, the
original creators of the operating system placed a one megabyte limit on the memory that could be present in the computer. One megabyte does not provide adequate room for both the program and an image, since an image itself can be in the neighborhood of 0.6 megabytes. This limit still exists, but programmers with special software packages are able to use more than the one megabyte. These software packages were rather clumsy in accessing memory beyond the one megabyte limit. Solving this problem would add a significant amount of time to the project.

Another drawback of the IBM PC platform was the particular IBM PC used for development. This machine had an 8-bit video board rather than a 24-bit video board thus creating a color problem. The user has to be able to view the image as realistically as possible. An 8-bit video provided only 256 colors while a 24-bit video board allowed 16 million colors.

Furthermore, the IBM PC platform proved to be too slow in calculating a number of filters in View. Many of the filters that were constructed within the View software package were mathematically intensive. These filters caused the PC to slow down to an unacceptable level. A faster machine was required before further development could continue.

Due to the problems with the IBM PC platform, the View program was no longer developed. A new direction was taken in the research, and a new program, Green, had already begun to solve the above problems. The development of Green would be undertaken on a UNIX platform, which did not have the difficulties that were present on the IBM PC. View's results and ideas were carried over into Green as much as possible.
To create the desired tree counting function, the first tool implemented was a chromaticity filter, the same filter previously mentioned. This filter essentially fixes all intensity data in an image. The effect produced is the removal of shadows and light spots (Figure 2), when compared to the original image (Figure 1).

Figure 1. Original Image

Figure 2. Output of Chromaticity Filter
Next, the segmentation between the trees and the background is considered. With the knowledge that healthy trees are green, a filter could be designed to remove all nongreen material. At this point, a threshold value is entered by the user. This value distinguishes the "green" of the trees from the background (Procedure 2). Upon the completion of this step, the trees have been successfully removed from the background, provided the proper threshold value was given (Figure 3). The next and most complicated step is to separate groups of trees. The following set of functions provides such an operation.

3.1 Procedure 2. Green Filter

The green filter removes nongreen data from the image.

if pixel_green < threshold then
    remove pixel

Figure 3. Output of Green Filter

In an earlier step, the image had the intensity data removed so that color differentiation could be used to separate the trees from the background. Intensity data are now required to detect subtle differences within the image. These
differences highlight the fact that two touching trees are not one but two trees. The first step in this process is a masking feature.

Using Procedure 3, the image produced from the green filter will now regain all the intensity information that was lost to the chromaticity filter. The effect is that the original image is recreated without the background.

### 3.2 Procedure 3. Mask Filter

The mask filter will allow an image to regain the lost intensity data.

if old_pixel is not the_color_black then
    new_pixel = color_from_the_original_image

A black and white filter is applied to this image to remove all the colored data. To produce the black and white image, Procedure 4 is applied to all pixels in the image.
3.3 Procedure 4. Black and White Filter

Upon running this filter, an image will be converted from a colored image to a black and white image.

\[ \text{new}_\text{red} = \text{new}_\text{green} = \text{new}_\text{blue} = \sqrt{\text{old}_\text{red}^2 + \text{old}_\text{green}^2 + \text{old}_\text{blue}^2} \]

old_red, old_green and old_blue are the old color components of a pixel.
new_red, new_green and new_blue are the new color components of a pixel.

Figure 5 - Output of Black and White Filter

The output is then put through a variance filter (Procedure 5) [2]. The variance filter will highlight the changes that occur throughout the image (Figure 6). The algorithm uses all eight neighboring pixels to recompute the value of a pixel. This computation is done by taking the sum of the square of the difference between the neighboring pixel values and the pixel that is being recomputed. Upon the completion of this computation on all pixels, the edges of the trees will become
visible. Note that neighboring trees are separated by a thin bright line. The flood routine was then developed to distinguish neighboring trees.

### 3.4 Procedure 5. Variance Filter

The variance filter will perform an edge detection operation on the current image.

For \( y = 0 \) to image_height

For \( x = 0 \) to image_width

\[
\text{SUM} = 0;
\]

For \( a = -1 \) to \( 1 \)

For \( b = -1 \) to \( 1 \)

\[
\text{SUM} = \text{SUM} + (\text{IMAGE}[y][x] - \text{IMAGE}[y-a][x-b])^2
\]

\[
\text{IMAGE\_OUT}[y][x] = \sqrt{\text{SUM}}
\]

Figure 6. Output of Variance Filter
4. IMAGE SEGMENTATION

The flood routine visualizes data as a number of pools rather than a number of pixels. What is meant by this statement is that the darker areas in the image are deep sections of a pool and the lighter areas are shallower sections of a pool. The algorithm begins at the deepest a pool can be, black, and works up to the highest a pool can be, white. All pools are filled equally and at the same time. When two or more pools touch, a decision has to be made whether this is one pool or a number of pools. To make this decision, the algorithm notes the depth of the touching pools. Should one pool be much deeper than the rest, the area in question will be considered a separating area. This separating area currently joins two or more separate pools together. The separating area has to be removed in order to separate the pools. If there are no pools much deeper than the rest, the area in question is a joining area. This area will be marked and recorded and will join these neighboring pools together. These pools will continue to be filled and tested with other neighboring pools. By this time the trees are separated and all are distinguished by a unique color (Figure 7).

With this basic understanding of the flood routine, a more detailed description will be given. Please consult Procedure 6 for the pseudo-code on the flood routine.
Figure 7 - Output of Flood Filter
4.1 Procedure 6. Flood Filter

The flood filter will separate touching pools.

\[
\text{for } z=0; z<256; z++; \quad (* \text{Start at greatest depth and work up } *) \\
\text{for } y=0; y< \text{image_height}; y++; \quad (* \text{Go through out image } *) \\
\text{for } x=0; x< \text{image_width}; x++; \\
\text{if } \text{image}[y][x] == z \text{ and } \text{image}[y][x] \text{ not part of any pool} \\
\text{then } \quad (* \text{pixel at current depth } *) \\
\text{Find all pixels at same depth that are grouped with } \\
\text{image}[y][x] \text{ and place in group_list} \\
\text{Place all pixels in touching_group if} \\
\quad 1) \text{ they are not part of group_list} \\
\quad 2) \text{ they are part of earlier pools} \\
\text{if touching_group is empty} \\
\quad \text{start new pool here} \\
\text{else} \\
\text{pool_diff} = 0 \\
\text{for } i=0; i<\text{num_of_touching_pools}; i++ \\
\quad \text{if } (z - \text{pool_depth(touching_pool}[i]) > \text{threshold}) \text{ then} \\
\quad \text{pool_diff}++ \\
\text{if pool_diff} > 0 \quad (* \text{unattach two pools } *) \\
\quad \text{color all pixels in group_list black} \\
\text{else } (* \text{addition to pool } *) \\
\quad \text{color all pixels in group the color of singular pool} \]
Flood starts scanning the image at the lowest level at which a pool may be, this is, when $z$ is equal to zero. At each increment of the pool level, the image is scanned to find if any pixels happen to lie at this level. When a pixel is found to be at this level and has not already been processed, the flood routine will begin to scan for a pixel group. A pixel group is a set of contiguous pixels at the same level. Should there be no pixels in the pixel group, the pixel in question must be the base of a pool. Therefore, this pixel is considered to be the start of a pool. The flood routine will then stop processing this pixel and return to scanning for other pixels at this level. If the pixel group is not empty, the formation of the touching group has to commence.

A touching group is created from all pixels that neighbor a pixel in the pixel group but are not part of the pixel group. A pixel must also be part of another pool for it to be placed in the touching group. When the formation of a touching group is completed, a decision process may begin as to the nature of the pixel group.

The pixels in the touching group are scanned to find the nature of the pools they comprise. If these pools are deeper, with respect to the current level, than the threshold, a pool difference counter will then be incremented. When this scan is completed, a determination is made about the pixel group's nature. If the pool difference counter is greater than zero, the pixel group is connecting a number of pools together. Therefore, the pixel group has to be marked, processed, and colored black to be disassociated from these touching pools. If, however, the difference counter is zero, the pixel group is a new pool. The pixel group is made a unique color, corresponding to a pool. The new pool's depth and position are recorded for future computations.

The above process will occur throughout the image and at all pool levels. Upon the completion of this step, all trees should be segmented and uniquely...
colored according to the threshold value. The image is now ready to have the objects counted.
5. TREE COUNTING

The final step in the tree segmentation and counting procedure is cataloguing all the trees. The user is requested to select the maximum and minimum tree sizes. All trees that fall within these two values are catalogued. Those that fall outside this boundary are unmarked and discarded with the idea that they were produced by other green objects such as grass or by noise in the image. This operation is very similar to producing the pixel group in the flood routine.

The image is scanned until it finds the first black pixel. At this time, a tree group begins to form. The tree group is built upon this first discovered pixel and expands to encompass all contiguous pixels that are not null. Upon the completion of the tree group, the object's type may be resolved. The area of the tree group is compared to the area of the maximum and minimum tree sizes. If the area of the tree group is above the maximum, this object is considered to be grass or other green objects that were able to pass through up to this point. Should the tree group's area be less than the minimum, the object will be considered noise due to the image acquisition or processing. This smaller area might also be a result of the object being a small tree or bush which is not wanted in the final output (Figure 8). If the tree group's area lies between these two limits, the tree group's center and average radius will be recorded for future processing.
The set of algorithms mentioned above will give a relatively experienced user the ability to quickly locate trees within an image. Along with these tools, a number of other tools were included so that Green may be used for other imaging applications.
6. CONCLUSIONS

The Green program was able to work well under a defined set of conditions. A suitable image has to be of high quality. The original proposal requested the incorporation of images from video technology. This technology may still be used provided that Super-VHS is used to capture and store the image. The image itself must be captured under certain conditions to provide a reasonably accurate response. It was found that images with trees that are larger than 10x10 pixels appear to work best. This is mainly due to the fact that it will set up "nice" pools for the flood algorithm. The image also has to be captured when the sun is directly above the area being captured. This will minimize the number and size of the shadows and will allow for maximum color differentiation across an individual tree.

This project was going to be the front end of a visualization program that would allow a user to "walk" within a forest and interact with the growth and mortality rates of trees within the forest. The Green program would produce too much data which may use up the available storage in the computer. However, this may not present a problem as storage technology improves.

Another use, which was found during development, is the ability of the program to quickly discover areas of the forest which have become diseased. With this in mind, the Green program could quickly view a large forest and find the percentage of diseased forest with very little user input.
APPENDIX A. ADDITIONAL FEATURES

A.1 Additional Tools

The most helpful tool included in the Green program was the zoom function. Upon running this function the first time, another window will appear. This window provides a zoom into the area where the pointer is. This function was useful when trying to decide whether an object was in fact a tree or a smaller vegetative form. Each subsequent selection of the zoom option increased the magnification by a factor of two. Thirty-two is the maximum magnification and after this it would reset to a magnification of two. Rapid zoom operation was needed so that the user was not hindered by the update process of the zoom window. To accommodate this fact, the zoom function was written using only an integer operation, which can be computed faster than floating-point operations [1].

Much like the zoom function, the histogram function also provides the user with some required data about the current image [3]. The histogram option allows the user to select an area within the image; then, a display of the intensity spread of red, green and blue will follow. This option allows the user to access useful information about the colors that are present within the image. Using this knowledge, the user will then know how the color channels are being used within that portion of the image. This option gives the data required so that the user may easily create a filter or use a filter that uses singular channel information.

The histogram option allows the user to view each channel separately, so that the differences between each will be apparent. Another graph, the chromaticity graph, will display differences present throughout the color spectrum. Upon selecting this option, another window will be displayed showing how the colors are
spread throughout the image. Using this option, the user can easily recognize what colors are predominant within the image. For example, in some forests, the chromaticity will show a group of colors in the green area, denoting the trees, another group will be in the browns, denoting the soil. The main benefit of the chromaticity display is that the user may now use or create a filter designed around a specific color, or group of colors.

A.2 Additional Filters

In addition to the variance filter previously mentioned, a fractal filter was included for use with black and white images [3]. The fractal filter replaces all pixels by their "surface area". Rather than conceptualizing pixel values as image data, the fractal filter views the data as altitude data. The fractal filter will then compute the overall change in surface area. If all pixels are of the same value, then this value will be zero. However, if there is a deviation or an edge, this value will become greater than zero. The filter in general is an excellent edge detection algorithm, but it does not provide an output that would be suitable for the flood routine.

A Kirsh filter was also tested and therefore included in the final Green package [3]. The Kirsh filter uses the derivative, or slope, of the pixel value in all eight directions. The maximum derivative is kept, and this value becomes the new pixel value. Much like the fractal filter, the Kirsch filter is valuable but it did not provide the proper output.
A.3 Additional Image Enhancements

Many of the images brought into Green are not in the proper form for tree segmentation or counting. Green, therefore, provides some features for massaging the images into the correct form. For the most part the features mentioned in Chapter 3 should provide the necessary means to create a suitable image, but additional enhancements are included for more unique image problems.

A median filter was created in an attempt to alleviate noise that may be present in an image [2]. This function is accomplished by replacing each pixel value with the median of its eight neighboring pixels. The median filter will therefore remove some of the spurious data within the image. However, it will also remove the small edges, required in some images, to segment individual trees from a group.

Green's functions have been centered on the desire to segment individual trees. In some cases, the user may want to learn not about individual trees, but rather about groups of trees. This function is provided through the box filter, which allows the user to select an area within the image and then create a table containing all the colors that lie within the selected area. The colors of all pixels within the image will then be compared with those in the table. If the color of the pixel matches any color in the table, the pixel will then be removed from the image. A figure is given to the percentage of the image that has been removed upon the completion of the filter operation. This operation is helpful when looking at a forest that is being consumed by some type of pest. The user can select an area in which the pest has infested the forest and then the filter will find all other areas. The user will then be told the percentage of the forest that has been destroyed by the pest.

When completing an image segmentation and counting of trees, it became helpful to overlay the results onto the original image. This function is possible with
the overlay option. The overlay option takes the red boxes that are created from the tree counting procedure and places them on the last image loaded, which should be the original image. This allows the user to easily see the accuracy of their selected thresholds (Figure 9).

Figure 9. Result Masked on Original Image
APPENDIX B. SPECIFIC RESULTS

Two images were counted for trees by both the Green program and a qualified landscape architect. The results are presented in Figure 10 for two criteria (thresholds) used by Green.

<table>
<thead>
<tr>
<th>Image</th>
<th>Green's Tree Count</th>
<th>Expert Count</th>
<th>Green Criteria Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM10.TGA</td>
<td>124</td>
<td>144</td>
<td>A</td>
</tr>
<tr>
<td>BM00.TGA</td>
<td>89</td>
<td>77</td>
<td>B</td>
</tr>
</tbody>
</table>

Figure 10. Tree Count Comparison

Notice that the criteria used by the qualified landscape architect is probably different from the criteria used by Green. These two criteria could easily be matched better with some initial runs during which the thresholds of Green are adjusted.
#include <gl/gl.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <device.h>
/* #include <get.h> */
#include "tiffio.h"

#include <string.h>

int threshold,test;
#include "datastr.c"
#include "sgigt.c"
#include "sgisv.c"

FILE *testone();
FILE *testtwo();
void parseheader();
void find_image();
void deQ();
void make_box();
void Filter_box();
void traverse();
void equalize();
void Image_Update();
void Check();
void Histogram();
void ThrHld();
void Window_Input();
void Color_Grad();
void chromo_display();
void zoom();
double floor();
long Get_Input();
double Get_Input_d();
int traverse_2();
FILE *ImageN();
FILE *Open_Up();
FILE *Get_File_Name();
long int Window2,Window,Zoom;
void Make_Root();
void Change_Dir();
void section(), warp2();
long int Thr_Win;
int New;
int max_x, min_x, max_y, min_y;
unsigned long count;
char DIR[80] = "//data//pradja//images//";
unsigned long int hist=0,chist = 0;
unsigned long int *z_p;
int z_f=1;

int test = 0;

int xmax,xmin,ymax,ymin;

double floor(double x) {
int a;
a = (int) x;
return ((double)a);}

main ()
{
TIFF *tiff;
long vert[2];
short colorse[3],tst2;
int x,y,z,junk,Redo;
Colorindex i;
long int a,b,c,d,x1,y1,x2,y2,CurrentWindow;
long xp,yp,x_old,y_old;
char buffer[80],buffer2[80];

struct header hdr;
unsigned long int *image, *image2, *temp, *orig;
FILE *fpi;
FILE *fpo;
long main_menu,win_menu,fit_menu,fil_menu,oth_menu,warp_menu,warp_param,
rem_back,seg_obj;
Device device;
short device_value;
double t,s;
node *root[DEPTH];

qreset();
qdevice (LEFTMOUSE);
qdevice (RIGHTMOUSE);
qdevice (MIDDLERMOUSE);
qdevice (MOUSEX);
qdevice (MOUSEY);
tiff = NULL;

image = malloc ((long) 1600 * (long)1600 * (long)sizeof(long int) );
image2 = malloc ((long) 1600 * (long)1600 * (long)sizeof(long int) );
orig = malloc ((long) 1600 * (long)1600 * (long)sizeof(long int) );
z_p = malloc ((long) 400 * (long)400 * (long)sizeof(long int) );

hdr.image_width = 0;
fpi = ImageN (&hdr,image);

if ((fpi!=NULL) I I (hdr.TIFF)) {

Make_Root(root);

prefsize (hdr.image_width,hdr.image_height);
Window = winopen("iMage Workbench");
zbuffer(0);
doublebuffer();
RGBmode();
gconfig();

prefsize (hdr.image_width,hdr.image_height);
Window2= winopen("Output");
zbuffer(0);
doublebuffer();
RGBmode();
gconfig();

ThrHld(TRUE);

warp_menu = newpup();
addtopup (warp_menu,"dxdu%x61 | dxdxdudu%x62 | dxdxdxdududu%x63 | dydv%x64 | dydydv
v%x65",0);
addtopup (warp_menu,"dydu%x66 | dxdv%x67 | dxdxdudv%x68 | Reset%x69",0);

win_menu = newpup();
addtopup (win_menu,"Zoom%x10 | Histogram%x11 | Chromaticity%x12");
oth_menu = newpup();
addtopup (oth_menu,"Fract%x30%1 | Kirsch%x24 | Threshold%x23 | Color
Gradiant%x25 | B/W Gradiant%x27",0);
addtopup (oth_menu,"Edge Detection%x31%1 | Equalize%x21 | Median
Filter%x22%1 | Minority-Majority%x28 | Remove Isolation%x26 | Clean Up%x32",0);
addtopup (oth_menu,"Overlay%x50 | Box Filter%x56 | Remove Image%x52 | Move
Pixel%x55 | Outline%x59",0);

fil_menu = newpup();
addtopup (fil_menu,"Input%x40 | Output%x41 | Directory%x42",0);
rem_back = newpup();
addtopup (rem_back,"Chromaticity lmage%x53 | Green%x20 | Mask%x51",0);
seg_obj = newpup();
addtopup (seg_obj,"Black and White%x54 | Variance %x29 | Segment Obj%x58",0);

main_menu = newpup();
addtopup (main_menu,"Files%m | Remove Background%m | Segment
Objects%m | Count
Objects%x57 | Tools%m | Other%m | Warp%m | Quit",fil_menu,rem_back,seg_obj,wi
n_menu,oth_menu,warp_menu);

winset (Window2);
color(BLUE);
clear();
swapbuffers();
winset (Window);
color(BLUE);
clear();
swapbuffers();

if (!hdr.TIFF) {
    Load_Image (root,fpi,&hdr,image,orig);
  /* show (root); */
lrectwrite(0,0,hdr.image_width - 1,hdr.image_height - 1,image);
}

swapbuffers();
unqdevice(MIDDLEMOUSE);
qdevice(MIDDLEMOUSE);
winset (Window);
getorigin (&a,&b);

a = 0;
New = 0;

while (1) {
    while (!qtest()) {
        if ((New) && (Redo) && (!getbutton(RIGHTMOUSE))) {
            printf (" UPDATE \n");
            Color_Grad(image,image2,&hdr,b);
            winset(Window2);lrectwrite(0,0,hdr.image_width - 1,hdr.image_height - 1,image2);
            swapbuffers();
            New = 0;
        }
    }
    switch (device = qread (&device_value)) {
        case INPUTCHANGE:
            CurrentWindow = 0;
            if (device_value! = 0) CurrentWindow = device_value;
            break;
        case WINSHUT :
            if ((device_value == Window) || (device_value == Window2)) {
                printf (" Window shut \n");
            }
    }
}
freepup(main_menu); fclose(fpi);
Free_Data(root); free(image); free(image2);
gexit(); exit(0);
break;
case ESCKEY:
    printf (" ESC key hit \n"); freepup(main_menu); fclose(fpi);
    Free_Data(root); free(image); free(image2);
gexit(); exit(0);
break;
case REDRAW:
    winset(Window2); Irectwrite(0,0,hdr.image_width - 1 ,hdr.image_height - 1,image2);
    swapbuffers();
    winset(Window); lrectwrite(0,0,hdr.image_width - 1 ,hdr.image_height - 1,image);
    swapbuffers();
    break;
case MIDDLEMOSUE:
    unqdevice(MIDDLEMOUSE);
    while (getbutton(MIDDLEMOUSE)) {} 
    temp = image; image = image2; image2 = temp; printf (" Done with Swap\n");
    winset(Window2); lrectwrite(0,0,hdr.image_width - 1 ,hdr.image_height - 1,image2);
    swapbuffers();
    winset(Window); lrectwrite(0,0,hdr.image_width - 1 ,hdr.image_height - 1,image);
    swapbuffers();
    qdevice(MIDDLEMOUSE);
    break;
default : break;
if (CurrentWindow == Thr_Win) {
    junk = threshold;
    ThrHld (FALSE);
    if (junk!=threshold) {
        menu_sel (a,image,image2,orig,&hdr,root,fpi,fpo,TRUE);
    }
    xp = getvaluator (MOUSEX);
    yp = getvaluator (MOUSEY);
}
if (((xp=x_old) || (yp=y_old)) && (Zoom!=NULL)) {
    x_old = xp; y_old = yp;
    zoom(image,image2,&hdr,xp,yp);
}
if ((device == RIGHTMOUSE) && (CurrentWindow == Window)) {
    winset (Window);
```c
int getsize (&a, &b);
 int a = dupup(main_menu);
 if (a!=8) {
    menu_sel (a, image, image2, orig, &hdr, root, fpi, fpo, FALSE);
 } else {
    freepup(main_menu);
    if (!hdr.TIFF) { Free_Data(root); fclose(fpi); }
    free(image); free(image2);
    gexit(); exit(0);
 }

 void menu_sel (long a, unsigned long *image, unsigned long *image2, unsigned long *orig, struct header *hdr, node [DEPTH], FILE *fpi, FILE *fpo, int UPDATE) {
    unsigned long *temp;
    int x, y, z, Redo;
    long b, c, d;
    char buffer[80], buffer2[80];
    double t, s;
    /**** warp params ****/
    double dx dv = 0;
    double dx dxdu = 0.0;
    double dx dxdu = 0.0000000;
    double dx dx dudu = 0.0000000;
    double dx dx dv dv = 0.0000000;
    double dxdxdudv = 0.0000000;
    double dy dv = 1;
    double dy dy dv dv = 0.0000000;
    double dy du = 0.0000000;
    double dy du = 0.0000000;
    double dy dudu = 0.0000000;
    double dy dy dudu = 0.0000000;
    double dy du = 0.0000000;
    double dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
    double dy dy du = 0.0000000;
```

case 10:
    if (Zoom==NULL) {
        prefsize (400,400);
        Zoom = winopen("Zoom");
        zbuffer(0);
        doublebuffer();
        RGBmode();
        gconfig();
        z_f=1;
    }
    else {
        if (++z_f > 4) {z_f = 1;])
            x = (int) pow(2,z_f);
            printf("Zoom Factor: %d \n",x);
            break;
    }

    x = (int) pow(2,z_f);
    printf("Zoom Factor: %d \n",x);
    break;

    case 11: Histogram (image,image2,hdr); printf(" Done with
Histogram \n");break;

    case 12 : chromo_display (root); break;

    case 20 : Green (image,image2,hdr); printf(" Done with
Green\n"); break;
    case 21 : equalize (image,image2,hdr); printf(" Done with Equalize\n");
    break;
    case 22 : median_filter (image,image2,hdr); printf(" Done with Median
Filter\n"); break;
    case 23 : Thr_Clean (image,image2,hdr); printf(" Done with threshold
\n"); break;
    case 24 : Kirsh (image,image2,hdr); printf(" Done with Kirsch
filter\n"); break;
    case 25 : Color_Grad(image,image2,hdr,b); printf(" Done with color
grad\n"); break;
    case 26 : Move_Binary (image,image2,hdr,4); printf(" Done removing
isolation\n"); break;
    case 27 : Move_Binary (image,image2,hdr,5); printf(" Done with B/W
grad\n"); break;
    case 28 : Min_Maj (image,image2,hdr); printf(" Done with
Min_Maj\n"); break;
    case 29 : Vary (image,image2,hdr); printf(" Done with Vary\n");
    break;
    case 30 : Fract (image,image2,hdr); printf(" Done with Fract\n");
    break;
    case 31 : Move_Color (image,image2,hdr); printf(" Done with Edge
Detection\n"); break;
case 32 : while (Min_Maj(image, image2, hdr) > 300) { temp = image; image = image2; image2 = temp; 
    Min_Maj(image, image2, hdr); temp = image; image = image2; image2 = temp; } printf(" Done with Clean Up\n"); break;

case 40 : printf(" free up + close %ld \n", raster);
    if (!hdr->TIFF) {fclose(fpi);
    if (!hdr->TIFF) {Free_Data(root);
    if (root==NULL) { printf(" nill root \n");

    fpi = NULL;
    hdr->TIFF = 0;

    while ((fpi==NULL) && (!hdr->TIFF)) {
        fpi = ImageN (hdr, image);
        printf(" %d \n",hdr->TIFF);

        winset (Window); getorigin (&a,&b);
        winset (Window2); getorigin (&c,&d);
        winclose (Window2); winclose (Window);

        preposition (a+hdr->image_width,a,b+hdr->image_height,b);
        Window = winopen("iMage Workbench");

        zbuffer(0);
        doublebuffer();
        RGBmode();
        gconfig();

        preposition (c+hdr->image_width,c,d+hdr->image_height,d);
        Window2= winopen("Output");

        zbuffer(0);
        doublebuffer();
        RGBmode();
        gconfig();

        if (!hdr->TIFF)
            Load_Image (root,fpi,hdr,image,orig);
        a = 40;
        printf(" load done\n");
    break;
    case 41 :
    fpo = NULL;
    Get_Input (buffer,0,1);
strcpy (buffer2,DIR);
strcat (buffer2,buff);
if (strstr (buffer2,".tif") ) {
tiffsv(buffer2,image,hdr);
strcpy(buffer,"TIFF"); }
else {
fpo = fopen (buffer2,"wb");
if (fpo != NULL) {
   Save_Image (fpi,fpo,hdr,root,image);
   strcpy(buffer,"TARGA");
   fclose(fpo);
}
   printf (" Write to %s as a %s file is done.
",buffer2,buffer);
break;
}
case 42 :Change_Dir();
   break;

   case 50 : Overlay (image,image2,orig,hdr); printf (" Done with overlay
"");
break;
   case 51 : Mask (image,image2,orig,hdr); printf (" Done with mask
"");
break;
   case 52 : Move_Binary (image,image2,hdr,6); printf ("Image removed
"");
break;
   case 53 : Chromo_pic (image,image2,hdr,root); printf (" Done with
Chromaticity Image
");
   case 54 : Black_White(image,image2,hdr); printf (" Done with Black-
White Image
");
   break;
   case 55 : Move_Pixel (image,image2,hdr); printf (" Done with
Move_Pixel
");
   break;
   case 56 : Filter_Box (image,image2,hdr,root); printf (" Done with Filter
Box
");
   break;
   case 57 : Scan_Line (image,image2,hdr); printf (" Done with Object Count
"");
   break;
   case 58 : Flood (image,image2,orig,hdr); printf (" Done with Object
Segmentation
");
   break;
   case 59 : Outline (image,image2,orig,hdr); printf (" Done with
Outline
");
   default : break;
   }
}
if (UPDATE) {
   switch(a) {
   case 20 : Green (image,image2,hdr); printf (" Done with green
"");
break;
   case 23 : Thr_Clean (image,image2,hdr); printf (" Done with threshold
"");
break;
}
}}
case 25: Color_Grad(image, image2, hdr, b); printf("Done with color
gradation\n"); break;
case 27: Move_Binary(image, image2, hdr, 5); printf("Done with B/W
gradation\n"); break;
case 31: Move_Color(image, image2, hdr); printf("Done with Edge
Detection\n"); break;
case 58: Flood(image, image2, orig, hdr); printf("Done with Object
Segmentation\n"); break;
default: break;}

if ((a>59) && (!UPDATE)) {
    b = a - 60;
    if ((b!=0) && (b!=9)) {
        t = Get_Input_d();

    switch (b) {

        case 1: dxdu = t; break;
        case 2: dxdxdudu = t; break;
        case 3: dxdxdxdududu = t; break;
        case 4: dydv = t; break;
        case 5: dydydvdv = t; break;
        case 6: dydu = t; break;
        case 7: dxdv = t; break;
        case 8: dxdxdudv = t; break;
        case 9: dxdv = 0;
            dxdxdudu = 0.0; dxdxdxdududu = 0.0000000;
            dxdxdvdv = 0.000; dxdxdudv = 0.00; dydv = 1;
            dydydvdv = 0.0000; dydu=0.0; dxdv=1;
            dydydu = 0.00; break;

        default: break;
    }
    t = 1.0; s=0;
    warp2(image, image2, hdr, 0, hdr->image_width-5, &t, &s, 0,
            dxdv, dxdxdudu, dxdxdxdududu, dxdxdvdv, dxdxdudv, 
            dydv, dydydvdv, dydu, dxdv, dydydu );
    printf("Done with warp \n");
}

if ((a==7) && (b==2)) { Redo = TRUE; } else { Redo = FALSE; }
ponseset(Window2); lrectwrite(0, 0, hdr->image_width - 1, hdr->image_height - 1, image2);

33
swapbuffers();
        winset(Window); lrectwrite(0,0,hdr->image_width - 1,hdr->image_height - 1,image);
        swapbuffers();
    }

void find_image(FILE *fpi, struct header *head)
{/*****************************************************/
    /* Find_image will move the file pointer to point to */
    /* the first pixel of an image. */
    /* */
    /* Expects: *fpi: file pointer */
    /* *head: pointer to file header information */
    /* */
    /* Returns: *fpi: file pointer pointing to image. */
    /*****************************************************/
    {
        long address;

        /* jump over color_map and identifier field */
        address = head->color_map * head->cmap_len * (head->cmap_bitsize / 8) + head->id_length;
        address = address + 18;
        fseek(fpi,address,SEEK_SET);
    }

void deQ()
{
    unqdevice(LEFTMOUSE);
    unqdevice(MIDDLEMOUSE);
    unqdevice(RIGHTMOUSE);
    unqdevice(MOUSEX);
    unqdevice(MOUSEY);
}

void Filter_Box (unsigned long int *image,unsigned long int *image2,struct header *hdr,
    node *root[DEPTH])
{/**************************/}
    /* This routine will ask for a bounding */
    /* box and then filter all colors that are*/
    /* within the box. */
    /* */
    /**************************/}
    {
        long int x1,y1,x2,y2;


void Histogram(unsigned long int *image, unsigned long int *image2, struct header *hdr, 
        node *root[DEPTH])
{ /* Histogram will ask for a bounding box */
    /* and then show a B/W and color histogram*/
    /* for that specified area. */
    /* */
    /***************************************************************************/
    long data[256];
    long g[256], b[256];
    long s[3], t;
    int a, e, c, d;
    long inm;
    long int x1, y1, x2, y2, x, y;
    long max=1, max2=1;
    long cv[3];
    long vect[2];
    long index;

    make_box(&x1, &y1, &x2, &y2, hdr, image);
    if (hist == 0) {
        prefsize(260, 310);
        hist = winopen("Histogram");
        zbuffer(0);
        doublebuffer();
        RGBmode();
        gconfig();
        prefsize(260, 310);
        chist = winopen("Histogram");
        zbuffer(0);
        doublebuffer();
        RGBmode();
        gconfig();
        winset(chist);
RGBcolor (0x00, 0x00, 0x00);
clear();

winset (hist);
RGBcolor (0x00, 0x00, 0x00);
clear();

for (x=0; x<256; x++) { data[x] = g[x] = b[x] = 0; }
if (y1 > y2) { y = y2; y2 = y1; y1 = y; }
if (x1 > x2) { x = x2; x2 = x1; x1 = x; }
for (y = y1; y < y2; y++) {
    for (x = x1; x < x2; x++) {
        inm = image[y * hdr->image_width + x];
        c = (int) (inm & 0x00000ff);
        e = (int) ((inm & 0x000ff000) >> 8);
        a = (int) ((inm & 0xff000000) >> 16);
        data[a]++;
        g[e]++;
        b[c]++;
        if ((data[a] > max) && (a != 0)) { max = data[a]; if (max > max2) { max2 = max; } }
        if (g[e] > max2) { max2 = e; }
        if (b[c] > max2) { max2 = b[c]; }
    }
}
for (x = 0; x < 256; x++) {
    bgnline();
    RGBcolor (0xff, 0xff, 0xff);
    vect[0] = x; vect[1] = 0;
    v2i(vect);
    vect[0] = x; vect[1] = data * 300 / max;
    v2i(vect);
    endline();
    bgnline();
    RGBcolor ((short)x, (short)x, (short)x);
    vect[0] = x; vect[1] = 310;
    v2i(vect);
    vect[0] = x; vect[1] = 301;
    v2i(vect);
    endline();
}
index = (long) log10 ((double)max);
index = (long) pow (10, index);
if (max / index < 10) { index = index / 5; }
if (max / index > 45) { index = index * 5; }

printf (" Scale is at: %ld\n", index);
t = data[0];
data[0] = 0;
for (y = 0; y < 300; y += (300 * index) / max) {
    bgnline();
    RGBcolor (0xff, 0x00, 0xff);
    vect[0] = 0; vect[1] = y;
    v2i(vect);
    vect[0] = 256; vect[1] = y;
    v2i(vect);
    endline();
}
data[0] = t;
swapbuffers();
winset (chist);
for (x = 0; x < 256; x++) {
    s[0] = data[x]; a = 1;
    s[1] = g[x]; e = 2;
    s[2] = b[x]; c = 3;

    if (s[0] < s[1]) { t = s[1]; s[1] = s[0]; s[0] = t; a = 2; e = 1; }
    if (s[1] < s[2]) { t = s[2]; s[2] = s[1]; s[1] = t; d = e; e = c; c = d; }
    if (s[0] < s[1]) { t = s[1]; s[1] = s[0]; s[0] = t; d = a; a = e; e = d; }

    bgnline();
    switch (a) {
        case 1: RGBcolor (0xff, 0x00, 0x00); break;
        case 2: RGBcolor (0x00, 0xff, 0x00); break;
        case 3: RGBcolor (0x00, 0x00, 0xff); break;
        default: break;
    }
}

vect[0] = x; vect[1] = 0;
v2i(vect);
vect[0] = x; vect[1] = s[0] * 300 / max2;
v2i(vect);
endline();

bgnline();
switch (e) {
case 1: RGBcolor (0xff,0x00,0x00); break;
case 2: RGBcolor (0x00,0xff,0x00); break;
case 3: RGBcolor (0x00,0x00,0xff); break;
default: break;
}
vect[0] = x; vect[1] = 0;
v2i(vect);
vect[0] = x; vect[1] = s[1] * 300 / max2;
v2i(vect);
endline();

bgnline();
switch (c) {
case 1: RGBcolor (0xff,0x00,0x00); break;
case 2: RGBcolor (0x00,0xff,0x00); break;
case 3: RGBcolor (0x00,0x00,0xff); break;
default: break;
}
vect[0] = x; vect[1] = 0;
v2i(vect);
vect[0] = x; vect[1] = s[2] * 300 / max2;
v2i(vect);
endline();
}
swapbuffers();

void zoom (unsigned long int *image,unsigned long int *image2,struct header *hdr,
long x, long y)
/* *******************************************/
/* This is my favorite function because */
/* I optimized it rather nicely */
/* */
/* Zoom just that. Whenever the mouse */
/* moves the zoom window will be updated */
/* */
/* */
/* *******************************************/
{
long a,b,c,d;
long u,v;
long p,q;
int t,s;
int value;

int i, x_f, y_f;
unsigned long int temp;

t = (int) pow(2, z_f);
winset (Window);
getorigin (&a, &b);
c = x-a-(200/ t); d = y-b-(200/t);

y_f = d << (z_f);
value = 0x01ff;
for (a = 0; a<400; a++) {
    d = a* 400;
    x_f = c << (z_f);
    temp = ((y_f >> (z_f) & value) * hdr->image_width);
for (b = 0; b<400; b++) {
    z_p[d+b] = image[temp + ((x_f>>(z_f)&value))];
    x_f++;)
    y_f++;)

winset(Zoom); lrectwrite(0, 0, 399, 399, z_p);
swapbuffers();

}

void make_box(long int *x1, long int *y1, long int *x2, long int *y2,
struct header *hdr, unsigned long int *image)
/*****************************************************/
/* This function just makes a box in the */
/* image workbench window according to */
/* mouse movements. The rubber banding is */
/* thrown in for FREE! */
/* */
/*****************************************************/
{
    long int a, b:

    while (!getbutton(LEFTMOUSE)) {
        getorigin (&a, &b);
        *x1 = getvaluator(MOUSEX) - a;
        *y1 = getvaluator(MOUSEY) - b;
        setlinestyle(0);
        RGBcolor(0xff, 0x00, 0xff);
        while (getbutton(LEFTMOUSE)) {

    }
lrectwrite(0,0,hdr->image_width - 1,hdr->image_height - 1,image);
getorigin(&a,&b);
*x2 = getvaluator(MOUSEX) - a;
*y2 = getvaluator(MOUSEY) - b;
if (*x2>hdr->image_width) *x2 = hdr->image_width;
else if (*x2< 0 ) *x2 = 0;
if (*y2>hdr->image_height) *y2 = hdr->image_height;
else if (*y2< 0 ) *y2 = 0;
sboxs(*x1,*y1,*x2,*y2);
swapbuffers();
}

void traverse(node *root[DEPTH])
/***********************************************************/
/* This function is used for debugging */
/* */
/* */
/***********************************************************/
{
/* node *xpos, *ypos, *zpos;
int temp;
fremove = 0;
xpos = root->right;
while (xpos != NULL ) {
    ypos = xpos->up;
    while (ypos != NULL ) {
        zpos = ypos -> out;
        while (zpos != NULL ) {
            printf(" node %d %d %d \n",zpos->x,zpos->y,zpos->z);
        zpos = zpos->out;
    }
    ypos = ypos->up;
}
xpos = xpos->right;
}*/

void Image_Update ( unsigned long int *image, unsigned long int *image2,node *root[DEPTH], struct header *hdr)
/***********************************************************/
/* Whenever colors are dropped from the */
/* image selection this routine is called */
/* to update the image. */
/* */
/* */
/***********************************************************/
node *start;
int x,y;
unsigned char red,blue,green;
unsigned long int data;
int FOUND=TRUE;

for (y=0;y<hdr->image_height;y++) {
  for (x=0;x<hdr->image_width;x++) {
    data = image[y * hdr->image_width + x];
    red = data & 0xFF;
    green = (data & 0xFF00) >> 8;
    blue = (data & 0xFF0000) >>16;
    start = Find_pnt2(red,green,blue,root,&FOUND);
    if (!FOUND) image2[y * hdr->image_width + x] = 0xFF;
    else { image2[y*hdr->image_width + x] = data; }
  }
}

void Make_Root(root)

node *root[DEPTH];

FILE *alterimage (FILE *fpi, struct header *hdr, char name[100])

FILE *fpo;
char newname[8];
unsigned long data;
unsigned char up,down,red,green,blue;
int x,y;

strcpy (newname,name);
strcpy (strrchr(newname,'.'),".24\0");

printf (" The new image name will be: %s \n",strrchr(newname,'/')+1);
fpo = fopen (newname,"wb");
hdr->image_bitsize = 24;
mk_header(fpo,hdr);
save_cmap(fpi,fpo,hdr);
for (y=0;y<hdr->image_height;y++) {
    for (x=0;x<hdr->image_width;x++) {
        down = getc (fpi);
        up = getc (fpi);
        blue = (down & 0x01F) <<3;
        green = ((up & 0x03) << 6) | ((down & 0xE0) >> 2);
        red = (up & 0x7C) <<1;
        putc (blue,fpo);
        putc (green,fpo);
        putc (red,fpo);
    }
}
fclose (fpi);
fclose (fpo);
fpo = fopen (newname,"rb");
return (fpo);
}

void parseheader (FILE *fpi,struct header *head)
{
    int temp;
    fseek(fpi,0L,SEEK_SET);
    temp = getc(fpi);
    head->id_length = temp;

    /***********************************************************/
    /* parse_header will read in the header from a given file    */
    /* Expects: *fpi: file pointer                               */
    /* Returns: *head: new header information                    */
    /***********************************************************/
    */
}
head->color_map = getc(fpi);
head->type = getc(fpi);
temp = getc(fpi);
head->cmap_org = getc(fpi) * 256 + temp;
temp = getc(fpi);
head->cmap_len = getc(fpi) * 256 + temp;
head->cmap_bitsize = getc(fpi);
temp = fgetc(fpi);
head->image_x_org = getc(fpi) * 256 + temp;
temp = fgetc(fpi);
head->image_y_org = getc(fpi) * 256 + temp;
temp = getc(fpi);
head->image_width = getc(fpi) * 256 + temp;
temp = getc(fpi);
head->image_height = getc(fpi) * 256 + temp;
head->image_bitsize = getc(fpi);
head->image_descrp = getc(fpi);

 FILE *Open_Up (struct header *hdr)
/*****************************/
/* This calls and tests files to be opened*/
/* */
/* */
/*****************************/
{ FILE *fpi;

char name[80];

fpi = ImageN(hdr,name);
parseheader (fpi,hdr);
if (hdr->image_bitsize == 16) {
    printf ("--- This image is only 16bit\n");
    printf ("--- Doing conversion. \n");
    fpi = alterimage (fpi,hdr,name);
}
return (fpi);

FILE *ImageN (struct header *hdr, unsigned long int *image)
/*****************************/
/* This routine was donated by James Kuch */
/* */
/* It allows the user to select an input file */
/* */
/*****************************/
{  
FILE *temp;
int i;
char tmp[180],num[4];
char tbuff[80], nbuff[20];
int num_img_names;
char img_names [30][60];
long image_menu;
char path[30];
FILE *fpi;
unsigned long int WinOp;
noport();
prefsize (400,400);
WinOp = winopen("Output");

zbuffer(0);
doublebuffer();
RGBmode();
gconfig();
swapbuffers();
strcpy (path,DIR);
strcpy (tmp, "ls ");
strcat (tmp, path);
strcat (tmp, "*.24 ");
strcat (tmp, "ls ");
strcat (tmp, path);
strcat (tmp, "*.tif ");
strcat (tmp, "ls ");
strcat (tmp, path);
strcat (tmp, "*.tga > /tmp/image_names");

system (tmp);
  temp = fopen ("/tmp/image_names","r");
  num_img_names = 0;

  while (!feof (temp)) && (num_img_names < 30) fscanf (temp,"%s",img_names[num_img_names++]);

fclose (temp);
strcpy (tmp,"rm /tmp/image_names");
system (tmp);

image_menu = defpup ("Images \t");
for (i=0;i<num_img_names-1;i++) {
    strcpy (tbuff,img_names[i]);
    /* strcpy (strrchr(tbuff,'.'),"\0"); */
    strcpy (nbuff,strrchr(tbuff,'/'));
    addtopup(image_menu, &nbuff[1]);
}
i = -1;

while ((i = dopup(image_menu))==-1) {} 
freepup (image_menu);

strcpy (tbuff,img_names[i-1]);
if (strstr (tbuff,".tif")) {
    printf (" Loading tiff...\n");
    sgigt (tbuff,image,hdr);
    hdr->TIFF = TRUE;
    fpi = NULL;
}
else {
    fpi = fopen (tbuff,"rb");
parseheader (fpi,hdr);
    if (hdr->image_bitsize == 16) {
        printf ("-- This image is only 16bit\n");
        printf ("--- Doing conversion. \n");
        fpi = alterimage (fpi,hdr,img_names[i-1]);
    }
    winclose (WinOp);
    return (fpi);
}

void ThrHld (int START)
/***********************
/* This routine opens up and works a */
/* thresholding interface. the value is */
/* stored in global var : threshold */
/* */
/* */
/***********************
{
unsigned long int thr;
int x,y, old, TEST;
char buffer[10];

long int a,b,CurrentWindow;
short device_value;
Device device;

if ((!START) && (getbutton(RIGHTMOUSE))) {
  TEST = TRUE;
  while ((getbutton(RIGHTMOUSE)) && (TEST)) {
    winset (Thr_Win);
    old = threshold;
    getorigin (&a,&b);
    x = getvaluator (MOUSEX);
    y = getvaluator (MOUSEY);
    TEST = ((x>a-10)&&(x<a+290)&&(y>b)&&(y<b+51));
    setlinestyle (0);
    x = x - a;
    if (x < 0) { x = 0; }
    if (x > 255){ x = 255; }
    threshold = x;
    if ((TEST) && (x!=old)) {
      New = 1;
      old = x;
      RGBcolor (0x00,0x00,0x00);clear();
      RGBcolor (0xff,0x00,0x00);
      rectf (x-5,0,x+5,49);
      cmov2i(265,10);
      RGBcolor(0x00,0xff,0x00);
      sprintf (buffer,"%d",threshold);
      charstr(buffer);
    }
  }
}

else if (START) {
  presize (290,51);
  if (Thr_Win == 0) {
    Thr_Win = winopen ("Threshold");
    zbuffer(0);
    singlebuffer();
    RGBmode();
    gconfig();
    printf (" Update window
");
}
void Color_Grad (unsigned long int *image, unsigned long int *image2,
struct header *hdr, int th)
/
******************************************/
/* This routine will do a color gradient */
/* on the image. */
/******************************************/
{

int x,y,u,v,ya,xa,us,uf,vs,vf,dist;
unsigned long int a,b,c;
long bit[9],data,input,pos;
unsigned long int zero = 0;
int d1,d2,d3,d4,d5,d6,d7,d8;
long menu2;
int tem,t2,t3,bw=0;

for (y=1;y<hdr->image_height-1;y++)
   for (x=1;x<hdr->image_width-1;x++)
      {

rgb_data (x-1, y,&dl,&d2,&d3,image,hdr);
rgb_data (x+1,y,&d4,&d5,&d6,image,hdr);

dist = abs (d1 - d4) + abs(d2 - d5) + abs(d3 - d6);
rgb_data (x,y+1,&d1,&d2,&d3,image,hdr);
rgb_data (x,y-1,&d4,&d5,&d6,image,hdr);

dist += abs (d1 - d4) + abs(d2 - d5) + abs(d3 - d6);

if (dist < threshold) { input = 0; }
else /*if (dist > 255) */ { input = 0xffffff;}
/* else input = (((dist<<8) + dist) << 8) + dist; */

47
void Scan_Line ( unsigned long int *image, unsigned long int *image2, struct header *hdr)
/*****************************
/* Scan_Line counts how many objects */
/* are in the image. They should be */
/* separated with black (background). */
/* */
/*****************************
{
char linec[600],linel[600],linet[600];
int x,y,FOUND,MARK,tree=0;
unsigned long data;
long x1,y1,x2,y2,area_x,area_i,a,b;
FILE *fpo;

/* Tree *root,*current,*before,*new1,*c2,*b2; */
int x_c,y_c,radius;

fpo = NULL;
fpo = fopen ("/usr/people/vizlab/trees/data/tree.dat","w");
if (fpo == NULL) {printf (" Unable to open file\n");}
for (y=1;y<hdr->image_height;y++) {
  for (x=1;x<hdr->image_width;x++) {
    image2[y*hdr->image_width+x] = image[y*hdr->image_width+x] & 0x7FFFFF;
    /* make_box (&xl,&yl,&x2,&y2,hdr,image); */
    while (!getbutton(LEFTMOUSE)) {
      while (getbutton(LEFTMOUSE)) {
        max_x = max_y = 0;
        min_x = min_y = 10000;
        getorigin (&a,&b);
        x = getvaluator (MOUSEX) - a;
        y = getvaluator (MOUSEY) - b;
        xmax=0;ymax=0;xmin=0x0fff;ymin=0x0fff;
        count = 0;
        Mark_Up (x,y,image2,hdr);
      }
    }
  }
}
}
area_x = count + 10;
printf(" Maximum tree size: %d \n",area_x);

count = 0;
while (!getbutton(LEFTMOUSE)) {} 
while (getbutton(LEFTMOUSE)) {} 
max_x = max_y = 0;
min_x = min_y = 10000;
getorigin(&a,&b);
x = getvaluator(MOUSEX) - a;
y = getvaluator(MOUSEY) - b;
xmax=0;ymax=0;xmin=0xffff;ymin=0xffff;
Mark_Up(x,y,image2,hdr);
area_i = count - 10;
printf(" Minimum tree size: %d \n",area_i);

for(y=1;y<hdr->image_height;y++) {
  for(x=1;x<hdr->image_width;x++) {
    image2[y*hdr->image_width+x] = image[y*hdr->image_width+x] & 0x7FFFFF;
  }
}

for(y=1;y<hdr->image_height-2;y++) {
  for(x=1;x<hdr->image_width-1;x++) {

    if ((image2[y*hdr->image_width + x] != 0x000000) & (image2[y*hdr->image_width + x] & 0x800000) != 0x800000) {
      max_x = 0; min_x = 2048; max_y = 0; min_y = 2048; count = 0;
      Mark_Up(x,y,image2,hdr);

      if ((count < area_x) & (count > area_i)) {
        tree++;
        x_c = min_x + (max_x - min_x)/2;
        y_c = min_y + (max_y - min_y)/2;
        radius = ((max_x - min_x) + (max_y - min_y)) / 2;
        printf(" %d %d %d %d \n",tree,x_c,y_c,radius);
        if (fpo != NULL) {
          fprintf(fpo," S %d %d 2 2 80 13 50 %d N \n",x_c,y_c,radius);
        }

        for (x2 = min_x; x2 <= max_x; x2++) {
          image2[max_y * hdr->image_width + x2] = 0x00FF00;
          image2[min_y * hdr->image_width + x2] = 0x00FF00;
        }
      }
    }
  }
}

for (x2 = min_x;x2=max_x;x2++) {
  image2[max_y * hdr->image_width + x2] = 0x00FF00;
  image2[min_y * hdr->image_width + x2] = 0x00FF00;
for (y2 = min_y;y2=max_y;y2++) {
  image2[y2 * hdr->image_width + min_x] = 0x00FF00;
}
image2[ y2 * hdr->image_width + max_x] = 0x00FF00;
}
}
}
}
}
if (fpo!=NULL) { fclose (fpo); }
printf (" I count %d objects
",tree);
}
Mark_Up (int x, int y, unsigned long int *image2,struct header *hdr) {
    if ((x>-1) && (x<hdr->image_width) && (y>-1) && (y<hdr->image_height)) {
        if ((image2[ y * hdr->image_width + x] != 0x00000) && ((image2[y * hdr->image_width + x] & 0x800000) != 0x800000) && (image2[y * hdr->image_width + x] != 0x00FF00)) {
            image2[y*hdr->image_width+x] = image2[y*hdr->image_width+x] | 0x800000;
            count++;}
        if (x > max_x) { max_x = x;}
        if (x < min_x) { min_x = x;}
        if (y > max_y) { max_y = y;}
        if (y < min_y) { min_y = y;}
        Mark_Up (x+1,y,image2,hdr);
        Mark_Up (x+1,y+1,image2,hdr);
        Mark_Up (x+1,y-1,image2,hdr);
        Mark_Up (x,y+1,image2,hdr);
        Mark_Up (x,y+1,image2,hdr);
        Mark_Up (x,y-1,image2,hdr);
        Mark_Up (x-1,y,image2,hdr);
        Mark_Up (x-1,y+1,image2,hdr);
        Mark_Up (x-1,y-1,image2,hdr);
    }
}

void Change_Dir()
/*****************************/
/* Inputs a new directory name */
/* */
/* */
/*****************************/
{
    int MARK,i;
Get_Input (DIR,strlen(DIR),1);
MARK=FALSE;
for (i(strlen(DIR)-1;i>-1)&&(MARK);i--) {
  if (DIR[i]=='') {
    if (DIR[i]='/') {
      DIR[++i] = '/';
      DIR[++i] = '/';
      DIR[++i] = 0;
      MARK = TRUE;
    } else {
      DIR[++i] = 0;
      MARK = TRUE;
    }
  }
}

long Get_Input (char data[80],int start,int Alpha)
/*************************************************************************/
/* Opens a window and allows the user to input data. */
/*************************************************************************/
{
  unsigned long int In_Win;
  int count;
  Device dev;
  short val;
  long got;
  prefsize (400,40);
  In_Win = winopen ("Input");
zbuffer(0);
singlebuffer();
RGBmode();
gconfig();
if (start == 0) {for (count=0;count<80;count++) { data[count] = 0;}}

  winset (In_Win);
  color(BLACK);
clear();
RGBcolor(0xff,0x00,0x00);
cmov2i(10,10);
charstr(data);
count = start;
val = got = 0;
qdevice (KEYBD);

dev = qread (&val);

while (val!=13) {

if (dev==KEYBD) {

if (!(Alpha) && (val> 47) && (val< 58)) {
    data[count++]=val;
    val = val - 48;
    got = got * 10 + (long)val;
    val = 0;
}

else if ((val != 8) && (Alpha)) { data[count++]=val; }

if ((val == 8) && (count>O)) {
    data[--count] = ' ';
    got = got / 10;
}

RGBcolor(0x00,0x00,0x00);
clear();
RGBcolor(0xff,0x00,0x00);
cmov2i(10,10);
charstr(data);
}

dev = qread (&val);
}

winclose(In_Win);
return (got);
}

double Get_Input_d ()
/*****************************/
/* Same as above but allows for the */
/* input of numerical data.        */
/*                               */
/*****************************/
{
    unsigned long int In_Win;


int count, deci = 10;
int FRAC, NEG;
Device dev;
short val;
double got, po;
char data[80];
int x, start = 0;
prefsize (400, 40);
In.Win = winopen ("Input");
zbuffer(0);
singlebuffer();
RGBmode();
gconfig();

if (start == 0) {for (count = 0; count < 80; count++) { data[count] = 0; }}

winset (In.Win);
color(BLACK);
clear();

RGBcolor(0xff, 0x00, 0x00);
cmov2i(10, 10);
charstr(data);

count = start;
val = got = 0;
qdevice (KEYBD);

dev = qread (&val);

while (val != 13) {
    if (dev == KEYBD) {
        if ((val > 47) && (val < 58) || (val == '.') || (val == '-')) {
            data[count++] = val;
        }
        if ((val == 8) && (count > 0)) {
            data[--count] = ' '
        }
        RGBcolor(0x00, 0x00, 0x00);
clear();
RGBcolor(0xff, 0x00, 0x00);
cmov2i(10, 10);
charstr(data);

} 

dev = qread (&val);

} 

FRAC=FALSE; NEG = FALSE; 
for (x=0;x<strlen(data);x++) {
  if (data[x] == '-') {NEG = TRUE; }
  if (!FRAC) {
    if (data[x] != '.') {
      got= got* 10 + (double)(data[x] - 48);
    } else { FRAC= TRUE;}
  
  } 
  else {
    got= got+ (double) (data[x]-48) / (double)deci; 
    deci = deci * 10; 
  }
}

if (NEG) { got = - got; } 
winclose(In_Win); 
return (got); }

void warp2 (unsigned long int *image,unsigned long int *image2, struct header *
  hdr, int us,int uf, double *dxdui, double *dydui, int sel, 
  double dxdv, 
  double dx dx dv, 
  double dx dx dx dudu, 
  double dx dx dx dx dudu, 
  double dx dx d dv, 
  double dx dx dv d, 
  double dx d v, 
  double dy dv d v, 
  double dy d v, 
  double dy d u, 
  double dx d, 
  double dy d u d u d)

/***************************************************************/ 
/* Performs a warping of the image */
/ * */
/* */
/***************************************************************/ 
{
  int x,y,POUND,MARK,tree=0,x2,y2;
unsigned long data;

int u,v;
int now;

double last;
double last2;

double yd;
double x, start = 0, start2 = 1;
double inc;

int old, d, index = 100;
double ind = 0;
int q, p = 0;
double old2, delta;

for (v = 0; v < hdr->image_height; v++) {
  for (u = 0; u < hdr->image_width; u++) {
    image2[v*hdr->image_width + u] = 0;
  }
}

start2 = dxdu;
yd = 0;
for (v = 0; v < hdr->image_height; v++) {
  x = 1;
  xd = 0;
  dxdu = start2;
  for (u = 1; u < hdr->image_width; u++) {
    dydu += dydydudu;
    yd += dydu;
    y = (int) yd;

    if ((x < hdr->image_width) && (y < hdr->image_height)) {
      data = image[(y) * hdr->image_width + x];
      image2[(v)*hdr->image_width + (u)] = data;

      dxdxdudu += dxdxdxdududu;
      dxdu += dxdxdudu;
      x += dxdu;
      x = (int)x;
    }
  }
}
start2 += dxdxdudv;
dxdv += dxdxdvdv;
start += dxdv;
dxdu = start + dxdv;
dydv += dydyvdv;
yd+= dydv;
}

winset(Window2);lrectwrite(0,0,hdr->image_width - 1,hdr->image_height - 1,image2);
swapbuffers();

}

void equalize (unsigned long int *image,unsigned long int *image2, struct header *hdr) {
  /*******************************************************************************/
  /* Equalize the black and white image. */
  /*******************************************************************************/
  long set[256],x,y,T,t;
  int p;
  T = hdr->image_width * hdr->image_height;
  for (x=0;x<256;x++) { set[x] = 0; }
  for (y=0;y<hdr->image_height;y++) {
    for (x=0;x<hdr->image_width;x++) {
      set[(image[y*hdr->image_width + x] & 0xff)]++;}
    y=0;
    p=0;
    y = 0;
    T = T - set[0];
    set[0] = 0;
    for (x=1;x<256;x++) {
      y += set[x]; set[x] = (255 * y) / T;}
  for (y=0;y<hdr->image_height;y++) {
    for (x=0;x<hdr->image_width;x++) {
      p = set[(image[y*hdr->image_width + x] & 0xff)];
      image2[y*hdr->image_width + x] = p<<16 | p<<8 | p;}}

winset(Window2);lrectwrite(0,0,hdr->image_width - 1 ,hdr->image_height -
1,image2);
  swapbuffers();
}

void make_pass (unsigned long int *image,unsigned long int *image2, struct
header *hdr, int matrix[8][8])
/***************************/
/*
/* Performs matrix mult on the image.  */
/*
/*****************************/
{

int x,y,t,c,calc,sum;
sum = matrix[0][0] + matrix[0][1] + matrix[0][2] +
  matrix[1][0] + matrix[1][1] + matrix[1][2] +
  matrix[2][0] + matrix[2][1] + matrix[2][2];
if (sum==0) { sum = 1; }
for (y = 1; y < hdr->image_height-1;y++) {
  for (x=1; x < hdr->image_width-1;x++) {
    calc = matrix[0][0] * (image[(y-1)*hdr->image_width +(x-1)] & 0xff) + matrix[0][1] *
      (image[(y-1)*hdr->image_width +(x)] & 0xff) + matrix[0][2] * (image[(y-1)*hdr-
        >image_width +(x+1)] & 0xff)+
      matrix[1][0] * (image[(y)*hdr->image_width +(x-1)] & 0xff) + matrix[1][1] *
      (image[(y)*hdr->image_width +(x)] & 0xff) + matrix[1][2] * (image[(y)*hdr-
        >image_width +(x+1)] & 0xff)+
      matrix[2][0] * (image[(y+1)*hdr->image_width +(x-1)] & 0xff) + matrix[2][1] *
      (image[(y+1)*hdr->image_width +(x)] & 0xff) + matrix[2][2] * (image[(y+1)*hdr-
        >image_width +(x+1)] & 0xff);
    calc = calc / sum;
    calc = calc & 0xff;
    image2[y*hdr->image_width + x] = calc<<16 | calc<<8 | calc ;}
}

void maximum (unsigned long int *image,unsigned long int *image2, struct header
*hdr)
/***************************/
/*
/* Take maximum of both images.  */
/*
/*****************************/
{
int x, y;
for (y = 1; y < hdr->image_height-1; y++) {
    for (x = 1; x < hdr->image_width-1; x++) {
        if (image2[y*hdr->image_width + x] > image[y*hdr->image_width + x]) {
            image[y*hdr->image_width + x] = image2[y*hdr->image_width + x];
        }
    }
}

void rotate_m (int m[8][8])
{ /* Used to rotate a matrix. */
    int t;
    t = m[0][0];
    m[0][0] = m[0][1];
    m[0][1] = m[0][2];
    m[0][2] = m[1][2];
    m[1][2] = m[2][2];
    m[2][2] = m[2][1];
    m[2][1] = m[2][0];
    m[2][0] = m[1][0];
    m[1][0] = t;
}

void Kirsh (unsigned long int *image, unsigned long int *image2, struct header *hdr)
{ /* Kirsh filter routine. */
    int x, y;
    int matrix[8][8];
    unsigned long int *hold;
    hold = malloc ((long) 512 * (long) 512 * (long) sizeof(long int));
    for (y = 1; y < hdr->image_height-1; y++) {

for (x=1; x < hdr->image_width-1;x++) {
    hold[y*hdr->image_width + x] = image[y*hdr->image_width + x];
}

matrix[0][0] = 1; matrix[0][1] = 0; matrix [0][2] = -1;
matrix[1][0] = 1; matrix[1][1] = 0; matrix [1][2] = -1;
matrix[2][0] = 1; matrix[2][1] = 0; matrix [2][2] = -1;

printf (" Matrix in use: \n");
printf (" %d %d %d \\n",matrix[0][0],matrix[0][1],matrix[0][2]);
printf (" %d %d %d \\n",matrix[1][0],matrix[1][1],matrix[1][2]);
printf (" %d %d %d \\n",matrix[2][0],matrix[2][1],matrix[2][2]);
printf (" \n");
make_pass(hold,image,hdr,matrix);
winset(Window);lrectwrite(0,0,hdr->image_width - 1 ,hdr->image_height - 1,image); swapbuffers();
for (x=0;x<7;x++) {
    rotate_m(matrix);
    make_pass(hold,image2,hdr,matrix);
    maximum (image,image2,hdr);
    winset(Window);lrectwrite(0,0,hdr->image_width - 1 ,hdr->image_height - 1,image); swapbuffers();
}

free (hold);

void Vary (unsigned long int *image,unsigned long int *image2, struct header *hdr) {
    int x,y,a,b;
    int sum,c,d,e;
    int max,min;
    for (y = 1; y < hdr->image_height-1;y++) {
        for (x=1; x < hdr->image_width-1;x++) {
            sum = 0;

59
for (a=-1; a<2; a++) {
    for (b=-1; b<2; b++) {
        c = (image[(y)*hdr->image_width + x]&0xff);
        d = (image[(y+a)*hdr->image_width + x+b]&0xff);
        e = (image[(y)*hdr->image_width + x]&0xff) - (image[(y+a)*hdr->image_width + x+b]&0xff);
        sum += e * e;
    }
    sum = (int) sqrt ((double)sum);
    image2[(y)*hdr->image_width + x] = sum<<8 | sum<<16 | sum;
}
}

void Thr_Clean (unsigned long int *image, unsigned long int *image2, struct header *hdr)
/
/**************************
/* Remove pixels according to the threshold. */
/* */
/**************************/
{
    int x, y;

    for (y = 0; y < hdr->image_height; y++) {
        for (x = 0; x < hdr->image_width; x++) {
            image2[y*hdr->image_width + x] = image[y*hdr->image_width + x];
            if (((image[y*hdr->image_width + x] & 0xff) < threshold)) {
                image2[y*hdr->image_width + x] = 0x00;
            }
        }
    }
}

void Range (unsigned long int *image, unsigned long int *image2, struct header *hdr)
/
/**************************
/* Remove objects not within range. */
/* */
/**************************/
{

int x,y,a,b;
int sum,c,d,e;
int max,min;

for (y = 1; y < hdr->image_height-1;y++) {
    for (x=1; x < hdr->image_width-1;x++) {
        sum = 0; min = 0x1ff; max=0x00;
        for (a=-1;a<2;a++) {
            for (b=-1;b<2;b++) {
                c = (image[(y)*hdr->image_width + x]&0xff);
                d = (image[(y+a)*hdr->image_width + x+b]&0xff);
                if (d<min) { min = d; }
                if (d>max) { max = d; }}}
        sum = max-min;
        image2[(y)*hdr->image_width + x] = sum<<8 | sum<<16 | sum;
    }
}

void Fract (unsigned long int *image,unsigned long int *image2, struct header *hdr)
{                           
/* Perform a fractal filter on image. */
{                           
    int x,y;
    int a,b,c,d,e;
    double total_area,f,g,h,i,j,k,l,t;
    int s = 1;
    int max=0;

t = 1;
for (y = 1; y < hdr->image_height-1;y++) {
    for (x=1; x < hdr->image_width-1;x++) {
        total_area = 0;
        a = image[(y)*hdr->image_width + x]&0xff;
        b = image[(y)*hdr->image_width + x+1]&0xff;
        c = image[(y+1)*hdr->image_width + x]&0xff;
        d = image[(y+1)*hdr->image_width + x+1]&0xff;
        e = (a+b+c+d)>>1;
        ...
f = sqrt(pow((double)(b - a), 2.0) + pow((double)(e - a - b), 2.0) + t);
g = sqrt(pow((double)(c - b), 2.0) + pow((double)(e - b - c), 2.0) + t);
h = sqrt(pow((double)(d - c), 2.0) + pow((double)(e - c - d), 2.0) + t);
i = sqrt(pow((double)(a - d), 2.0) + pow((double)(e - d - a), 2.0) + t);

total_area = .25 * (f + g + h + i);
image2[(y)*hdr->image_width + x] = (int)total_area;
if (max<(int)total_area) { max = (int)total_area; }
}
}
printf (" max: %d \n",max);
for (y = 1; y < hdr->image_height-1;y++) {
  for (x=1; x < hdr->image_width-1;x++) {
    a = (255 * image2[(y)*hdr->image_width + x]) / max;
    a = a & 0xff;
    image2[(y)*hdr->image_width + x] = a<<16 | a<<8 | a;
  }
}

void Green(unsigned long int *image,unsigned long int *image2, struct header *hdr)
/* Filter out non green information */
/>
int x,y;
int r,g,b;
for (y = 1; y < hdr->image_height-1;y++) {
  for (x=1; x < hdr->image_width-1;x++) {
    r = image[(y)*hdr->image_width + x]&0x0000ff;
    g = (image[(y)*hdr->image_width + x]&0xff0000)>>8;
    b = (image[(y)*hdr->image_width + x]&0x00ff0000)>>16;
    image2[(y)*hdr->image_width + x] = image[(y)*hdr->image_width + x];
    g = g - threshold;
    if ((g<r) || (g<b)) {
      image2[(y)*hdr->image_width + x] = 0x000000;
    }
  }
}
void Mask(unsigned long int *image, unsigned long int *image2, unsigned long int *orig, struct header *hdr)

FLICTED
/* Replace image data with image2 data */
/* where image is not black(background) */
/

void Overlay(unsigned long int *image, unsigned long int *image2, unsigned long int *orig, struct header *hdr)

void Outline(unsigned long int *image, unsigned long int *image2, unsigned long int *orig, struct header *hdr)

----------------------
```c
void Flood (unsigned long int *image, unsigned long int *image2, unsigned long int *orig, struct header *hdr) {
    int MARK;
    for (y = 1; y < hdr->image_height-1; y++) {
        for (x = 1; x < hdr->image_width-1; x++) {
            image2[(y)*hdr->image_width + x] = orig[(y)*hdr->image_width + x];
            MARK = FALSE;
            if (image[(y)*hdr->image_width + x] != 0) {
                for (a = -1; ((a < 2) && (!MARK)); a++) {
                    for (b = -1; ((b < 2) && (!MARK)); b++) {
                        if (image[(y+a) * hdr->image_width + x + b] == OxOO) {
                            MARK = TRUE;
                        }
                    }
                }
                if (MARK) {
                    image2[(y)*hdr->image_width + x] = Oxffffff;
                } else {
                    image2[(y)*hdr->image_width + x] = Ox000000;
                }
            }
        }
    }
}
```
Fresh_Tree (root2);
for (y = 1; y < hdr->image_height-1; y++) {
    for (x = 1; x < hdr->image_width-1; x++) {
        image2[y*hdr->image_width + x] = 0x000000;
    }
}
for (z = 1; z < 255; z++) {
    for (y = 1; y < hdr->image_height-1; y++) {
        for (x = 1; x < hdr->image_width-1; x++) {
            if (((image[y*hdr->image_width+x] & 0xff) == z) && ((image2[y*hdr->image_width+x] & 0xff) == 0x00)) {
                SearchOut (image, image2, root2, x, y, z, hdr, &clr);
            }
        }
    }

    winset(Window2); lrectwrite(0, 0, hdr->image_width - 1, hdr->image_height - 1, image2);
    swapbuffers();
}

    winset(Window2); lrectwrite(0, 0, hdr->image_width - 1, hdr->image_height - 1, image2);
    swapbuffers();
    winset(Window); lrectwrite(0, 0, hdr->image_width - 1, hdr->image_height - 1, image);
    swapbuffers();
    clr = 0x800000;
for (y = 1; y < hdr->image_height-1; y++) {
    for (x = 1; x < hdr->image_width-1; x++) {
        if ((image2[y*hdr->image_width+x] & 0xf00000 != 0x800000) && (image2[y*hdr->image_width+x] != 0x00)) {
            Expand2 (image2, x, y, hdr, clr);
            clr += 40;
            clr = clr ^ 0x008000;
        } else {
            if ((image2[y*hdr->image_width+x]) == 0x80ff00) {
                image2[y*hdr->image_width+x] = 0x000000;
            }
        }
    }
}
void SearchOut (unsigned long int *image, unsigned long int *image2, Tree root2[DEPTH], int x, int y, int z, struct header *hdr, unsigned long *clr)

{ /* Find next object that touches */

    unsigned long temp;
    int x1, y1, a, b, c, cnt, FOUND;
    unsigned long make;
    unsigned long Touch[50];
    unsigned long int yf, xf2, yf2;
    temp = 0x00000000;

    xmax = 0; ymax = 0; xmin = 0x0ffff; ymin = 0x0ffff;
    Expand (image, image2, x, y, z, hdr); /* Find bounding box */

    cnt = 0;
    for (y1 = ymin; y1 < ymax; y1++) {
        yf = y1 * hdr->image_width;
        for (x1 = xmin; x1 < xmax; x1++) {
            if (image2[yf + x1] == 0x000000) {
                for (a = -1; (a < 2); a++) {
                    yf2 = (y1 + a) * hdr->image_width;
                    for (b = -1; (b < 2); b++) {
                        xf2 = yf2 + x1 + b;
                        if ((image2[xf2] & 0xf00000) == 0xa00000) {
                            cnt++;
                        }
                    }
                }
            }
        }
    }
}

Shake_Tree (root2);
}

swap buffers();
winset (Window); lrectwrite (0, 0, hdr->image_width - 1, hdr->image_height - 1, image);
swap buffers();

Shake_Tree (root2);
temp = image2[xf2];
FOUND = FALSE;
if (cnt == 0) {Touch[cnt++] = temp;}
else {
    for (c = 0;((c<cnt) && (!FOUND));c++) {
        if ( Touch[c] == temp ) { FOUND = TRUE; }}
    if (!FOUND) {
        Touch[cnt++] = temp;
        if (cnt>49) { cnt=49; printf(" Overload
");}
    }
}
}
}

make = 0;
if (cnt == 0) {
    (*clr) += OxOf;
test++;
    Add_Tree (*clr,z,root2);
    make = *clr;
}
else if (cnt == 1) {
    make = Touch[0];
}
else {
    make = 0x0000;
y1 = 0;a=256;
temp = 0xffffffff; y1 = 0;
for (x1 = 0;x1<cnt;x1++) {
    if ( Touch[x1] < temp) { temp = Touch[x1]; }

    if ( z - Get_Tree(Touch[x1],root2) > threshold) {
        y1++;
    }
}
make = temp;
if (y1>0) { make = 0x80ff00; )
for (y1 = ymin; y1 < ymax + 1; y1++) {
    yf = y1 * hdr->image_width;
    for (x1 = xmin; x1 < xmax + 1; x1++) {
        xf2 = yf + x1;
        if (image2[xf2] == 0x0000ff) {
            image2[xf2] = make;
        }
    }
}

void Expand (unsigned long int *image, unsigned long int *image2, int x2, int y2, int z2, struct header *hdr)
{
    if ((x2 > -1) && (x2 < hdr->image_width) && (y2 > -1) && (y2 < hdr->image_height) && ((image[y2 * hdr->image_width + x2] & 0xff) == z2) && (image2[y2 * hdr->image_width + x2] != 0x0000ff)) {
        image2[y2 * hdr->image_width + x2] = 0x0000ff;
        if (x2 > xmax) { xmax = x2; }
        if (x2 < xmin) { xmin = x2; }
        if (y2 > ymax) { ymax = y2; }
        if (y2 < ymin) { ymin = y2; }

        /* Expand (image,image2, x2-1, y2+1,z2, hdr); */
        Expand (image, image2, x2 - 1, y2 + 1, z2, hdr);
        /* Expand (image,image2, x2 , y2+1,z2, hdr); */
        Expand (image, image2, x2 + 1, y2 + 1, z2, hdr);
        Expand (image, image2, x2 + 1, y2, z2, hdr);
        Expand (image, image2, x2, y2, z2, hdr);
        Expand (image, image2, x2 - 1, y2 + 1, z2, hdr);
        /* Expand (image,image2, x2+1, y2-1,z2, hdr); */
        Expand (image, image2, x2 + 1, y2 - 1, z2, hdr);
        /* Expand (image,image2, x2 , y2-1,z2, hdr); */
        Expand (image, image2, x2, y2 - 1, z2, hdr);
        /* Expand (image,image2, x2-1, y2-1,z2, hdr); */
    }
}

void Expand2 (unsigned long int *image, int x, int y, struct header *hdr, unsigned long clr)
{ /*

}}

68
/* */
/* Build up objects on all 8 dim. */
/* */
/* */
/**************************/

if ((x>-1) && (x<hdr->image_width) && (y>-1) && (y<hdr->image_height)) {
  if (((image[y*hdr->image_width+x] & Ox000000) != Ox800000) && (image[y*hdr->image_width+x] != Ox000000)) {
    image[y*hdr->image_width+x] = clr;
    Expand2 (image, x-1, y+1,hdr,clr);
    Expand2 (image, x+1, y+1,hdr,clr);
    Expand2 (image, x+1, y+1,hdr,clr);
    Expand2 (image, x-1, y+1,hdr,clr);
    Expand2 (image, x+1, y-1,hdr,clr);
    Expand2 (image, x-1, y-1,hdr,clr);
    Expand2 (image, x+1, y-1,hdr,clr);
    Expand2 (image, x-1, y-1,hdr,clr);
  }
}

void Add_Tree (unsigned long clr, int deep, Tree *root2[DEPTH])
/**************************/

/* Add tree to trees in current list */
/* */
/* */
/**************************/

{ int i,x,y;
  int data;
  Tree *current,*new;
  i = clr & Ox03ff;
  current = root2[i];
  new = (Tree *) malloc (sizeof (Tree));
  new->next = NULL;
  new->x_start = deep;
  new->count = clr;
  if (current == NULL) {
    root2[i] = new;
  } else {
    current = root2[i];
    while (current->next != NULL) {current = current->next;}
  }
  new->next = current;
  current = new;
  current->parent = NULL;
  current->x_start = deep;
  current->count = clr;
  current->color = clr;
  current->left = NULL;
  current->right = NULL;
}
current->next = new;

int Get_Tree (unsigned long clr, Tree *root2[DEPTH])
{**************今***************************/
/*
*/
/* Return pointer to tree in list    */
/*
*/
/*
*/
/******今***************************/
{
    int x=1000,i,FOUND=FALSE;
    Tree *current;
    x = 0;
    i = clr & 0x03ff;
    current = root2[i];
    while ((current!=NULL) && (!FOUND)) {
        if (current->count != clr) { current=current->next; }
        else { FOUND = TRUE; x = current->x_start; }
    }
    return (x);
}

void Fresh_Tree ( Tree *root2[DEPTH])
{**********************今***************************/
/*
*/
/* Clear tree data set   */
/*
*/
/*
*/
/******今***************************/
{
    int x;
    for (x=0;x<DEPTH;x++) {
        root2[x] = NULL;
    }
}

void Shake_Tree (Tree *root2[DEPTH])
{**********************今***************************/
/*
*/
/* Remove everything from tree data set */
/*
*/
/*
*/
/******今***************************/
}
{ 
  Tree *current,*temp;
  int i;

  for (i=0;i<DEPTH;i++) 
  { 
    current = root2[i];
    while (current!=NULL) 
    { 
      temp = current;
      current = current->next;
      test ++;
      free (temp);
    }
    root2[i] = NULL;
  }
}

## C.2 Tools

#define DEPTH 4096
#define TRUE 1
#define FALSE 0
#include <stdio.h>
#include <gl/gl.h>
#include <math.h>
#include <stdlib.h>
#include <math.h>
#include <device.h>
struct header {
  int id_length;
  int color_map;
  int type;
  int cmap_org;
  int cmap_len;
  int cmap_bitsize;
  int image_x_org;
  int image_y_org;
  int image_width;
  int image_height;
  int image_bitsize;
}
typedef struct _vet {
    double x;
    double y;
    double z;
} Vector;

typedef struct _tree_me {
    int y_start;
    int y_last;
    int x_start;
    int x_finish;
    unsigned long count;
    struct _tree_me *next;
} Tree;

typedef struct look {
    int x;
    int ystart;
    int ylast;
    int ydif;
    int link;
    struct look *right, *left;
} scan;

typedef struct _3Dpoint {
    unsigned char x,y,z;
    unsigned long data;
    int xt,yt;
    int count;
    int input;
    unsigned char type,flag;
    struct _3Dpoint *out,*in;
} node;

typedef struct in_line {
    int value;
    int green;
    int blue;
int mag;
struct in_line *right;
} line_obj;

int filter_number,fremove,threshold,max_d_filter;
char buffer[80];
int Find_pnt_ins (), Find_pnt(),find_pos();
unsigned char find_pos_ret();
node *Find_pnt2();
node *Get_Start();
void Bridge(), Right_Neighbor(), Up_Neighbor(), Future_Filter(),
Future_Explosion(),Undo(),Free_Data(),Reset_Data(),Save_Image(),
mark_header(),parseheader(),save_cmap(),Change_Directory(),Load_Image(),
Change_Dir(),Chromo_pic(),Unit_Vector(),Box_Filter(),Black_White(),
Move_Pixel(),read_store(),Read_in_bin(),Move_Binary(),Move_Color(),
line_look(),add_to_list(),scan_image(),show(), rgb_data;

int Min_Maj();
void median_filter(), bubble_sort(), mask(), Set_in(), Clear_in(),
Read_in(),Display_Text();
int Lines,length;
double Magnitude();
int Point_dist();
node *Draw_Blue(),*start;
FILE *Open_File(),*alterimage();
int f_max();

int Find_pnt_ins ( x, y, z, root)
/*****************************/
/ */ Find_pnt_ins will find a point in the point table. If the point does not */
/ * exist in the table it will be added to the table. */
/ * */
/ */ Expects: The Point, Root node */
/ */
/*****************************/
/
unsigned char x,y,z;
node *root[DEPTH];

{ node *newnd, *current, *last;
  int OK = TRUE;
  int a,b,i;
  unsigned long data;
data = ((((unsigned long) z << 8) + (unsigned long) y) << 8) + (unsigned long) x;
\[ i = (((\text{int})x \& 0x78) \ll 5) + (((\text{int})y \& 0x78) \ll 1) + ((\text{int})z \& 0x78) \gg 3); \]

\[ \text{current} = \text{root}[i]; \]

if (current == NULL) {
    newnd = (node *) malloc (sizeof (node));
    newnd->x = x;
    newnd->y = y;
    newnd->z = z;
    newnd->count = 1;
    newnd->input = 1;
    newnd->type = 0;
    newnd->data = data;
    newnd->out = newnd->in = NULL;
    root[i] = newnd;
    fremove++;
} else {
    if (data < current->data) {
        fremove++;
        newnd = (node *) malloc (sizeof (node));
        newnd->x = x;
        newnd->y = y;
        newnd->z = z;
        newnd->data = data;
        newnd->count = 1;
        newnd->input = 1;
        newnd->type = 0;
        newnd->out = current;
        root[i] = newnd;
    } else {
        while ((current->out!=NULL) && (current->data < data)) {
            last = current;
            current = current->out;
        }
    }
}

if ((current->data != data)) {
    fremove++;
    newnd = (node *) malloc (sizeof (node));
    newnd->x = x;
    newnd->y = y;
    newnd->z = z;
    newnd->data = data;

newnd->count = 1;
newnd->input = 1;
newnd->type = 0;
newnd->out = newnd->in = NULL;
if (current->data < data) {
    newnd->out = current->out;
    current->out = newnd;
} else {
    newnd->out = last->out;
    last->out = newnd;
}
else {
    current->input = current->count = current->count + 1;
}
}
return (OK);

node *Find_pnt2 (x, y, z, root, FOUND)
****************************************************************************/

/* Find_pnt will find a point in the point table. If the point does not */
/* exist it will return FALSE, if it does exist it will return TRUE */
/* */
/* Expects: Point, Current Transformation */
/* Returns: TRUE if point exists, FALSE otherwise */
****************************************************************************/

unsigned char x, y, z;
node *root[DEPTH];
int *FOUND;
node *last, *current;
unsigned long data;
int i;

data = ((((unsigned long) z << 8) + (unsigned long) y) << 8) + (unsigned long) x;
i = (((int)x & 0x78) << 5) + (((int)y & 0x78) << 1) + ((int)z & 0x78) >> 3);
current = root[i];
if (current!=NULL) {
    while ((current->out!=NULL) && (current->data != data)) {
        current = current->out;
    }
}
if ( ((current!=NULL) && (current->data == data) && (current->count != 0)) ) {
    *FOUND = TRUE; }
else { *FOUND = FALSE; current = NULL; }

return (current); 

node *Get_Start (horiz, vert, root, hdr, image)

/* Given a horizontal and vertical position from the screen image, */
/* Get_Start will return a pointer to the color in the cube that */
/* corresponds to that point. */
/* */
/* *Expects: horiz (horizontal viewing image location) */
/* vert (vertical viewing image location) */
/* root (root of data structure) */
/* fpi (image file pointer) */
/* hdr (image file header) */
/* */
/* *Returns: (pointer to node in data structure) */
/* */
/* *********************************************** */

int horiz, vert;
node *root[DEPTH];

unsigned long int *image;
struct header *hdr;

{ int Found;
  node *temp;
  int x, y;
  unsigned char r, g, b;
  unsigned long int data;
  long int address;
  char buffer[80]; /* calculate x,y image position */

  data = image[hdr->image_width * vert + horiz];

  r = (unsigned char) (data & 0xFF);
  g = (unsigned char) ((data & 0xFF00)>>8);
b = (unsigned char) ((data & 0xFF0000) >> 16);

temp = Find_pnt2 (r,g,b,root,&Found); /* Look for the point */
if (!Found) { temp = NULL; } /* If not found return null else return */
return (node *) temp; /* the pointer */
}

void Undo (root, Filter_Number)
/******************************-**********************-***************-*************-*********
/* Undo will simply move the data from the element last to the count element */
/* if last is not -1. */
/* */
/******************************-**********************-***************-*************-*********/
node *root[DEPTH];
unsigned char *Filter_Number;
{
    /* node *xpos, *ypos, *zpos;

    xpos = root; handy Set all last variables to -1 */
    /* while (xpos != NULL ) {
        ypos = xpos;
        while (ypos != NULL ) {
            zpos = ypos;
            while (zpos != NULL ) {
                if ((zpos-> type) == *Filter_Number) {
                    zpos-> count = zpos->input;
                    zpos-> type = 0;
                }
                zpos = zpos->out;
            }
            ypos = ypos->up;
        }
        xpos = xpos->right;
    } */

    *Filter_Number = *Filter_Number - 1;
    if (*Filter_Number < 0 ) { *Filter_Number= *Filter_Number + 1; }
}

void Free_Data (root)
/******************************-**********************-***************-*************-*********/
/* Free_Data will travel through the data structure and free up each node. */
/* The root node will not be removed. */
/* */
node *root[DEPTH];
{
    node *current,*temp;
    int i;

    for (i=0;i<DEPTH;i++) {
        current = root[i];
        while (current!=NULL) {
            temp = current;
            current = current->out;
            free (temp);
        }
        root[i] = NULL;
    }
}

void Reset_Data (root)
{  
    node *root[DEPTH];
}

void Save_Image (fpi,fpo, hdr, root, image)
{  
    FILE *fpo, *fpi;
}
struct header *hdr;
node *root[DEPTH];
unsigned long *image;
{ char buffer[80];
 int x,y,c;

unsigned long pos;
unsigned char red,green,blue;
mk_header(fpo,hdr); /* Produce the introduction */
save_cmap (fpi,fpo,hdr);

for (y=0;y<hdr->image_height;y++) {
 for (x=0;x<hdr->image_width;x++) {

   pos = (unsigned long int) ((unsigned long int)(y)*(unsigned long int)hdr-
->image_width + (unsigned long int)x);
   red = (unsigned char) ( image[pos] & 0xFF );
   green = (unsigned char) ((image[pos] & 0xFF00) >> 8);
   blue = (unsigned char) ((image[pos] & 0xFF0000) >> 16);

   fputc (blue,fpo);
   fputc (green,fpo);
   fputc (red,fpo);
 }
 }

void mk_header(FILE *fpo, struct header *head)
 { /**************************************************************************/
   /* mk_header will move the current header elements to a */
   /* file. */
   /* */
   /* */
   /* Expects: *fpo : pointer to output file */
   /* *head: pointer to current header info */
   /**************************************************************************/
   {  fseek (fpo,0L,SEEK_SET); /* Move pointer to head of file */
     putc(head->id_length, fpo);
     putc(head->color_map, fpo);
     putc(head->type, fpo);
     putc((head->cmap_org & 255), fpo);
     putc((head->cmap_org >> 8), fpo);
     putc((head->cmap_len & 255), fpo);
     putc((head->cmap_len >> 8), fpo);
     putc(head->cmap_bitsize, fpo);
   }
putc((head->image_x_org & 255), fpo);
putc((head->image_x_org >> 8), fpo);
putc((head->image_y_org & 255), fpo);
putc((head->image_y_org >> 8), fpo);
putc((head->image_width & 255), fpo);
putc((head->image_width >> 8), fpo);
putc((head->image_height & 255), fpo);
putc((head->image_height >> 8), fpo);
putc(head->image_bitsize, fpo);
putc(head->image_descrip, fpo);

void save_cmap(FILE *fpi, FILE *fpo, struct header *head)
{
    fseek (fpi,181,SEEK_SET);
    fseek (fpo,181,SEEK_SET);
    /* Set file pointers properly */
    for (i=0; i<head->color_map * head->cmap_len * (head->cmap_bitsize / 8) + head->id_length;
        j++)
    {
        tmp = getc(fpi);
        putc(tmp, fpo);
    }
}

void Inverse (node *root[DEPTH])
{
    node *xpos, *ypos, *zpos;
    int temp;
    fremove = 0;
    xpos = root->right;
    while (xpos != NULL )
    {
        ypos = xpos->up;
        "80"
while (ypos != NULL) {
    zpos = ypos->out;
    while (zpos != NULL) {
        if (zpos->count == 0) {
            fremove++;  
            zpos->count = zpos->input; zpos->type = 0;
        } else {
            zpos->count = 0; zpos->type = filter_number;
        }
        zpos = zpos->out;
    }
    ypos = ypos->up;
}
xpos = xpos->right;

void Alter_Image (FILE *fpi, struct header *hdr, char *Dir_in, char *file_in) {
    FILE *fpo;
    char *temp, file_name[80], junk[3] = "\0";
    printf (" A 16-bit image has been specified as an input file");
    printf (" This image will be converted to a 24 bit image");
    printf (" The file name will be changed to a .24 extension");
    /* temp = strchr(file_in,46); */
    *(temp + 1) = '2';
    *(temp + 2) = '4';
    *(temp + 3) = '\0';
    /*fpo = Open_File (Lines,file_in,junk,Dir_in,TRUE); */
    /* convert_16_2_24 (fpi,fpo,hdr,"",file_in); */
    fclose (fpi);
    fclose (fpo);
    /* fpi = Open_File (Lines,file_in,junk,Dir_in,FALSE); */
    if (fpi != NULL) {
        parseheader (fpi,hdr);
    }
}

void Check_Image ( node *root,FILE *fpi,struct header *hdr, char Lines[10][80],
    char *Dir_in, char *file_in) {
    parseheader (fpi,hdr);
    if (hdr->image_bitsize == 16) {
        Alter_Image (fpi,hdr,Dir_in,file_in);
    /* Load_Image (root,fpi,hdr,Lines,Dir_in,file_in); */
}
void Load_Image (node *root[DEPTH],FILE *fpi,struct header *hdr,
    unsigned long int *image, unsigned long int *orig)
{
    int temp,color,x,y,z,xpos;
    unsigned char red,blue,green;
    short colorse[3];
    long vert[2];
    char buffer[80];
    node *foo;
    int a,b,i,left,right;
    unsigned long total;
    printf (" ");
    printf (" University of Illinois at Urbana \ Champaign \\
    Dr. Kevin Warren \\
    Prof. Brian Orland \\
    Paul Radja \\
    ");
    fremove = 0;
    hdr->TIFF = FALSE;
    for (y=0;y<512;y++) {
        for (x=0;x<512;x++) {
            orig[512 * y + x] = image[512 * y + x] = 0x00;
        }
    }
    find_image (fpi,hdr);
    total = 0;
    for (y = 0;y<hdr->image_height;y++) {
        for (x=0;x<hdr->image_width;x++) {
            blue = (unsigned char) fgetc (fpi);
            green= (unsigned char) fgetc (fpi);
            red = (unsigned char) fgetc (fpi);
            temp = Find_pnt_ins (red,green,blue,root);
            if (temp == FALSE) {blue=green=0;red=255;}
            orig[y*hdr->image_width +x] = image[y*hdr->image_width +x] =
            (((long)blue<<8)+(long)green)<<8) + (long)red;
        }
    }
    printf (" This image has %d unique colors ( 24 bit ) \n ",fremove);
    printf (" Size of image: %d x %d \n ",hdr->image_width,hdr->image_height);
}
void Chromo_pic ( unsigned long int *image, unsigned long int *image2, struct header *hdr, node *root[DEPTH] )
{

    unsigned long int data;
    unsigned char red, blue, green;
    Vector V;

    int y, x, mult, green_adjust, temp;

    mult = 255;

    for (y = 0; y < hdr->image_height; y++) {
        for (x = 0; x < hdr->image_width; x++) {

            V.z = (double) ((image[y*hdr->image_width + x] & 0xFFF00000) >> 16);
            V.y = (double) ((image[y*hdr->image_width + x] & 0xFF000) >> 8);
            V.x = (double) ((image[y*hdr->image_width + x] & 0xFF));

            Unit_Vector (&V);

            blue = (char)(V.z * mult);
            green = (char)(V.y * mult);
            red = (char)(V.x * mult);

            temp = Find_pnt_ins (red, green, blue, root);
            if (temp==FALSE) { red = 0; green = blue = 255;
            data = (((unsigned long)blue<<8)+(unsigned long)green)<<8) + (unsigned long)red;
            image2[y*hdr->image_width + x] = data;

        }
    }

}

void Black_White ( unsigned long int *image, unsigned long int *image2, struct header *hdr )
{
    char temp[10];
    unsigned char red, blue, green;
    unsigned long int data;

83
int y, x, mult, green_adjust;
Vector V;
double mag, sqr3;

sqr3 = sqrt ((double)3);

for (y = 0; y < hdr->image_height; y++) {
    for (x = 0; x < hdr->image_width; x++) {
        V.z = (double) ((image[y*hdr->image_width + x] & 0xFF0000) >> 16);
        V.y = (double) ((image[y*hdr->image_width + x] & 0xFF00) >> 8);
        V.x = (double) ((image[y*hdr->image_width + x] & 0xFF));

        if ((V.x == 255) && (V.y == 0) && (V.z == 0)) {
            red = blue = green = 0;
        } else {
            mag = Magnitude (V);
            red = blue = green = (char)(mag * sqr3 /4);
        }

        data = (((unsigned long)blue<<8)+(unsigned long)green)<<8) + (unsigned long)red;
        image2[y*hdr->image_width + x] = data;
    }
}

int f_max(line_obj* a, line_obj* b, line_obj* c, line_obj* d, line_obj* e, int f)
{
    if (a->mag > f) f = a->mag;
    if (b->mag > f) f = b->mag;
    if (c->mag > f) f = c->mag;
    if (d->mag > f) f = d->mag;
    if (e->mag > f) f = e->mag;
    return f;
}

void Move_Pixel (unsigned long int *image, unsigned long int *image2, struct header *hdr)
{ Vector input;
    line_obj *a00,*a01,*a02,*a03,*a04, /* matrix of values */
         *a10,*a11,*a12,*a13,*a14,
         *a20,*a21,*a22,*a23,*a24,
unsigned char m0,m1,m2,m3,m4,m5;
unsigned long int data;
int x,y,i, maxi;
int STEP,th,mag;


line1 = (line_obj *) malloc(sizeof (line_obj));
line1->right = NULL; line1->value = 0;
line2 = (line_obj *) malloc(sizeof (line_obj));
line2->right = NULL; line2->value = 0;
line3 = (line_obj *) malloc(sizeof (line_obj));
line3->right = NULL; line3->value = 0;
line4 = (line_obj *) malloc(sizeof (line_obj));
line4->right = NULL; line4->value = 0;
line5 = (line_obj *) malloc(sizeof (line_obj));
line5->right = NULL; line5->value = 0;

Set_in (line1,hdr);
Set_in (line2,hdr);
Set_in (line3,hdr);
Set_in (line4,hdr);
Set_in (line5,hdr);

Read_in (0,hdr,line3,image,TRUE);
Read_in (1,hdr,line4,image,TRUE);
Read_in (2,hdr,line5,image,TRUE);

/***************************************************/

for (y=0;y<hdr->image_height;y++) {

    a00 = line1; a01 = a00->right; a02=a01->right; a03=a02->right; a04=a03->right;
    a10 = line2; a11 = a10->right; a12=a11->right; a13=a12->right; a14=a13->right;
    a20 = line3; a21 = a20->right; a22=a21->right; a23=a22->right; a24=a23->right;
    a30 = line4; a31 = a30->right; a32=a31->right; a33=a32->right; a34=a33->right;
    a40 = line5; a41 = a40->right; a42=a41->right; a43=a42->right; a44=a43->right;

    for (x=0;x<hdr->image_width;x++) {


maxi = f_max (a00,a01,a02,a03,a04,0);
maxi = f_max (a10,a11,a12,a13,a14,maxi);
maxi = f_max (a20,a21,a22,a23,a24,maxi);
maxi = f_max (a30,a31,a32,a33,a34,maxi);
maxi = f_max (a40,a41,a42,a43,a44,maxi);
if ((maxi > th) && (a22->mag == maxi)) {
    image2[y*hdr->image_width + x] = 0xFF;
} else {
    data = (((unsigned long)a22->blue<<8)+(unsigned long)a22->green)<<8) + 
    (unsigned long)a22->value;
    image2[y*hdr->image_width + x] = data;
}

a00 = a01; a01=a02; a02= a03; a03=a04; a04 = a04->right;
a10 = a11; a11=a12; a12= a13; a13=a14; a14 = a14->right;
a20 = a21; a21=a22; a22= a23; a23=a24; a24 = a24->right;
a30 = a31; a31=a32; a32= a33; a33=a34; a34 = a34->right;
a40 = a41; a41=a42; a42= a43; a43=a44; a44 = a44->right;
}
temp = line1;
line1 = line2;
line2 = line3;
line3 = line4;
line4 = line5;
line5 = temp;
Read_in (y+3,hdr,line5,image,TRUE);
}
Clear_in (line1,hdr);
Clear_in (line2,hdr);
Clear_in (line3,hdr);
Clear_in (line4,hdr);
Clear_in (line5,hdr);

}
void Move_Color (unsigned long int *image, unsigned long int *image2, struct header *hdr) {

int x,y,u,v,xa,us,uf,vs,vf;
unsigned long int set[9],a,b,c;
unsigned long int data,zero = 0;
int d1,d2,d3,d4,d5,d6,d7,d8,th = 90;

th = threshold;
for (y=0;y < hdr->image_height ; y++) {
  us = 0; uf = 3;
  if (y==0) { us = 1; set[0] = set[1] = set[2] = zero;}
  for (x=0; x < hdr->image_width ;x++) {
    for (u=us;u<uf;u++) {
      for (v=vs;v<vf;v++) {
        set[u*3+v] = (image[(y+u-1)*hdr->image_width + x + v -1]);
      }
    }
  }
}

d1 = Point_dist(set[0],set[4]);
d2 = Point_dist(set[1],set[4]);
d3 = Point_dist(set[2],set[4]);
d4 = Point_dist(set[3],set[4]);
d5 = Point_dist(set[5],set[4]);
d6 = Point_dist(set[6],set[4]);
d7 = Point_dist(set[7],set[4]);
d8 = Point_dist(set[8],set[4]);

/* Test for horizontal line */
if ((d4<th) && (d5<th) && ((d7>th) || (d2>th))) {
  image2[y*hdr->image_width + x] = 0xFF;
}

/* Test for vertical line */
else if ((d2<th) && (d7<th) && ((d4>th) && (d5<th) && (d5<th)) || (d5>th) && (d4<th))) {
  image2[y*hdr->image_width + x] = 0xFF;
}

/* Test for / type line */
else if ((d3<th) && (d6<th) && ((d1>th) && (d8<th) || (d8>th) && (d1<th))) {
  image2[y*hdr->image_width + x] = 0xFF;
}
/* Test for \ type line */

else if ((d1<th) && (d8<th) && ((d3>th) & (d6<th) || (d6>th) && (d3<th))) {
    image2[y*hdr->image_width + x] = 0xFF;
else { image2[y*hdr->image_width + x] = image[y*hdr->image_width + x ];}

vs = 0; vf = 3;
}

}

int Min_Maj (unsigned long int *image, unsigned long int *image2,
            struct header *hdr)
{
    int x,y,u,v,us,uf,vs,vf,lost,gain,t2;
    unsigned char a,b,c,bit[9];
    unsigned long pos,input;

    lost = gain = 0;
    us = 1; uf = 3;
    for (y=0;y < hdr->image_height ; y++) {
        if (y == hdr->image_height - 1) { uf = 2;}
        vs = 1;
        vf = 3;
        for (x=0; x < hdr->image_width ;x++) {
            input = image[(y)*hdr->image_width + x];

            if ( x==hdr->image_width-1) { vf = 2;]

            for (u=us;u<uf;u++) {
                for (v=vs;v<vf;v++) {
                    pos = (unsigned long int)(((unsigned long int)(y+u-1) * (unsigned long
int)hdr->image_width + (unsigned long int)x + (unsigned long int)v) -1);
                    a = (unsigned char) ( image[pos] & 0xFF );
                    b = (unsigned char) ((image[pos] & 0xFF00 ) >> 8);
                    c = (unsigned char) ((image[pos] & 0xFF0000 ) >> 16);
                    bit[u*3+v] = 0;

88
if ((a == 0xFF)) { bit[u*3+v] = 1; }

}

}

    if ((t2 < 4) && (bit[4])) { input = 0; lost++; }
else if ((t2 > 4) && (!bit[4])) { input = 0xFFFFFF; gain++; }

image2[y*hdr->image_width + x] = input;
vs = 0;
}
us = 0;
}

printf( "Pixels lost: %d gained: %d\n", lost, gain);
return (lost+gain);

}

void Move_Binary (unsigned long int *image, unsigned long int *image2,
        struct header *hdr, int th)
{

int x,y,u,v,ya,xa,us,uf,vs,vf,dist,lost,gain;
unsigned long int a,b,c;
long bit[9],data,input, pos;
unsigned long int zero = 0;
int d1,d2,d3,d4,d5,d6,d7,d8;
long menu2;
int tem,t2,t3,bw=0;

fremove = lost = gain = 0;
if ( (th == 4)) { bw = 1; }
for (y=0; y < hdr->image_height; y++) {
    us = 0; uf = 3;
    if (y==0) { us = 1; }
else if (y == hdr->image_height - 1) { uf = 2; }

    for (x=0; x < hdr->image_width; x++) {


input = image[(y)*hdr->image_width + x];
vs = 0; vf = 3;
if (x==0) { vs = 1; }
if (x==hdr->image_width-1) { vf = 2; }
for (u=us;u<uf;u++) {
for (v=vs;v<vf;v++) {
pos = (unsigned long int)((unsigned long int)(y+u-1)* (unsigned long int)hdr->image_width + (unsigned long int)x + (unsigned long int)v -1);
a = (unsigned char) (image[pos] & 0xFF);
b = (unsigned char) ((image[pos] & 0xFF00) >> 8);
c = (unsigned char) ((image[pos] & 0xFF0000) >> 16);
if (bw) { bit[u*3 + v] = ( long int)a; }
else if (a == 0xFF) { bit[u*3+v] = 1; }
else { bit[u*3 + v] = 0; }
}
}
tem = 0;
if (th == 1) {
if (tem) {
input = 0xFF;
}
else if (th == 2) { /* Calculate the max point */
}
else if (() & & (bit[4])) { /* Removes junk */
if (t2 == 0) { input = 0; }
}
else if () { /* Assume bw image --- will do gradient with max point*/
data = (-bit[0] + + bit[2] +
data = labs(data);
data = data + labs(input);
if (data > 255) { data = 255; }
if (data < threshold) { data = 0; }

input = (((data<<8) + data) << 8) + data;

    (bit[4] >= bit[5]) &&
    input = 0xFF; */
else if ((th == 5) && (!bit[4])) { input = 0;}
else if ((th == 6)) {
    bit[3] +
    bit[5] +
    if ((t2 < 4) && (bit[4])) { input = 0;lost++;}
    else if ((t2 > 4) && (!bit[4])) {input = 0xFF;gain++;}
}
else if ((th == 7) && (bit[4])) {
    t2 = bit[0] + bit[3] + bit[6];
    if (t2==0) { fremove++; }
}

image2[y*hdr->image_width + x] = input;
}

if (th == 7) { printf (" Tree counter: %d\n",fremove); }
if (th == 6) { printf (" Pixels lost: %d gained: %d\n",lost,gain);}
}

void Set_in (line_obj *line_start, struct header *hdr)
{ line_obj *start,*new1;
  int i;

  start = line_start;
  for (i=0;i<(hdr->image_width+6);i++) {
    new1 = (line_obj *) malloc (sizeof (line_obj));
    if (new1 != NULL) {
      start->right = new1;
      new1->right = NULL;
    }
  }
  // ...
void Clear_in (line_obj *line_start, struct header *hdr)
{
  line_obj *next,*begin;

  int i;
  begin = line_start;
  for (i=0;i<hdr->image_width;i++)
  {
    next = begin->right;
    free (begin);
    begin = next;
  }

}

void Read_in (int y, struct header *hdr, line_obj *line_in, unsigned long int *image, int COMP_MAG)
{
  line_obj *start;
  int x;

  start = line_in;
  if (y< hdr->image_height) {
    for (x=0;x<hdr->image_width;x++)
    {
      start->blue = ((image[y*hdr->image_width + x] & 0xFF0000 ) >> 16);
      start->green= ((image[y*hdr->image_width + x] & 0xFF00 ) >> 8);
      start->value= (image[y*hdr->image_width + x] & 0xFF);
      if (COMP_MAG) {
        start->mag = (int)sqrt ((double)start->blue * (double)start->blue +
                                (double)start->green * (double)start->green + (double)start->value * (double)start->value);
        }
        start = start->right;
    }
  }
  else {
    for (x=0;x<hdr->image_width;x++)
    }
}
```c
start->value = start->green = start->blue = start->mag = 0;
start = start->right;
}
}

int Point_dist (unsigned long int a, unsigned long int b)
{
    double dist;
    unsigned char ar,ag,ab,br,bg,bb;

    ab = (unsigned char) ((a & 0xFF0000) >> 16);
    ag = (unsigned char) ((a & 0xFF00) >> 8);
    ar = (unsigned char) (a & 0xFF);
    bb = (unsigned char) ((b & 0xFF0000) >> 16);
    bg = (unsigned char) ((b & 0xFF00) >> 8);
    br = (unsigned char) (b & 0xFF);

    dist = (double) (pow ((double)abs(ab - bb), 2) + pow ((double)abs(ag - bg), 2) + pow
                   ((double)abs(ar - br), 2));
    dist = sqrt(dist);
    return (int)dist;
}

void Read_in_bin (int y, struct header *hdr, line_obj *line_in, FILE *fpi)
{
    line_obj *start;
    int i;
    long address;

    start = line_in;
    address = hdr->color_map * hdr->cmap_len * (hdr->cmap_bitsize / 8) + hdr->id_length + 18;
    address = address + 3 * ((long) y * (long) hdr->image_width);
    fseek(fpi, address, SEEK_SET);
    start = start->right; /* Leave the begining 0 */
    start = start->right;
    if (y <= hdr->image_height) {
        for (i = 0; i < hdr->image_width; i++) {
            start->value = fgetc(fpi);
            if (start->value > 64) start->value = 255;
        }
    }
}
```
else { start->value = 0;
    fgetc (fpi); fgetc(fpi);
    start = start->right;
}
}
else {
    for (i=0; i<hdr->image_width; i++) {
        start->value = 0;
        start = start->right;
    }
}

void read_store ( data, fpo, hdr, x, y)
unsigned char data;
int x,y;
FILE *fpo;
struct header *hdr;
  { long pos,pos2;
    unsigned char temp;
    int test;
    temp = 0;
    if ((x>=O) && (y>=O) && (x<hdr->image_width) && (y < hdr->image_height)) {
        fputc ((int)data,fpo);
        fputc ((int)data,fpo);
        fputc ((int)data,fpo);
    }
}

unsigned char find_pos_ret (FILE *fpi, struct header *hdr, int x , int y,long address)
{
    unsigned char temp;
    temp = 0;
    address = address + (long) 3 * ((long)y * (long)hdr->image_width + (long)x);
    fseek(fpi,address,SEEK_SET);
    temp = (unsigned char) fgetc(fpi);
    return (temp);
}

int find_pos (FILE *fpi, struct header *hdr, int x, int y)
{ long pos;
    int temp = FALSE;
    if ((x>=O) && (y>=0) && (x<hdr->image_width) && (y < hdr->image_height)) {

```c
find_image(fpi, hdr);
    pos = 3 * (y * hdr->image_width + x);
fseek (fpi, pos, SEEK_CUR);
temp = TRUE;
return (temp);
}

double Magnitude (Vector A)
/*****************************************************************************/
/*
* Magnitude will compute the magnitude of a given vector
*/
/*
* Expects: A vector
* Returns: Magnitude of the given vector
*/
/*
*****************************************************************************/
{
    double a;
a = sqrt ((A.x) * (A.x) + (A.y) * (A.y) + (A.z) * (A.z));
    return(a);
}

void Unit_Vector (Vector *A)
/*****************************************************************************/
/*
* Unit_Vector will take a given vector and change it into a unit vector
*/
/*
* Expects: A vector
* Returns: A unit vector
*/
/*****************************************************************************/
{
    double Mag;
    Mag = Magnitude(*A); /* Compute magnitude of the vector */
    if (Mag != 0) {
        A->x = A->x / Mag; /* Change vector to a unit vector */
        A->y = A->y / Mag;
        A->z = A->z / Mag;
    }
}
```
void median_filter(unsigned long int *image, unsigned long int *image2, struct header *hdr)

/********************************************
/* The median_filter function will take an */
/* image and perform a median filter on it. */
/* The input image is give in image and the */
/* output image is image2. Information about*/
/* these images is stored in the hdr. */
/* The filter will not work on the edge */
/* pixels so that the code may be simplified*/
/*

*********************************************/
{
    int x,y,u,v,ya,xa;
    unsigned char set[3][9],a,b,c;
    unsigned long int data;

    for (x=0; x<hdr->image_width; x++) {
        image2[y*hdr->image_width + x] = image[y*hdr->image_width + x];
    }

    for (y=1; y < hdr->image_height-1; y++) {
        image2[y*hdr->image_width] = image2[y*hdr->image_width];
        for (x=1; x < hdr->image_width -1; x++) {
            for (u=0; u<3; u++) {
                for (v=0; v<3; v++) {
                    set[0][u*3+v] = (unsigned char) (image[(y+u-1)*hdr->image_width + x + v -1] & 0xFF);
                    set[1][u*3+v] = (unsigned char) ((image[(y+u-1)*hdr->image_width + x + v -1] & 0xFF00) >> 8);
                    set[2][u*3+v] = (unsigned char) ((image[(y+u-1)*hdr->image_width + x + v -1] & 0xFFF000) >> 16);
                }
            }
        }
    }
}
bubble_sort (&set);
image2[y*hdr->image_width + x] =
((((unsigned long)set[2][4]<<8)+(unsigned long)set[1][4]<<8) + (unsigned long)set[0][4]);

image2[y*hdr->image_width + hdr->image_width -1] = image[y*hdr->image_width + hdr->image_width -1];
}
for (x=0;x<hdr->image_width;x++) {
image2[y*hdr->image_width + x] = image[y*hdr->image_width + x];
}

void bubble_sort( char set[3][9] )
{
int i, ALTER = TRUE;
char temp;

while (ALTER) {
ALTER = FALSE;
for (i=0;i<8;i++) {
if (set[0][i] > set[0][i+1]) /* Sort Red */
    temp = set[0][i+1];
    set[0][i+1] = set[0][i];
    set[0][i] = temp;
    ALTER = TRUE;
}
if (set[1][i] > set[1][i+1]) /* Sort Green */
    temp = set[1][i+1];
    set[1][i+1] = set[1][i];
    set[1][i] = temp;
    ALTER = TRUE;
if (set[2][i] > set[2][i+1]) /* Sort Blue */
    temp = set[2][i+1];
    set[2][i+1] = set[2][i];
    set[2][i] = temp;
    ALTER = TRUE;
}
}
void mask (FILE *fpi, FILE *fpo, struct header *hdr, char *Dir_in)
{
    FILE *fprn;
    int x, y, th;
    unsigned char red, green, blue;

    /* fprn = Open_Input (Lines, Lines[8], Dir_in); */
    printf(" 1) Mask out filter 2) Add white ");
    /* ADisplay_Prompt (Lines); */
    /* th = atoi (Lines[8]); */
    if ((th!=1) && (th!=2)) { th = 1; }
    if (fprn!=NULL) {
        mk_header (fpo, hdr); /* Produce the introduction */
        save_cmap (fpi, fpo, hdr);
        find_image (fpi, hdr);
        find_image (fpm, hdr);
        for (y=0; y<hdr->image_height; y++) {
            for (x=0; x<hdr->image_width; x++) {
                blue = (unsigned char) fgetc(fprn);
                green = (unsigned char) fgetc(fpm);
                red = (unsigned char) fgetc(fpm);

                if ((x==0) && (x==hdr->image_width)) {
                    blue = (unsigned char) fgetc(fpi);
                    green = (unsigned char) fgetc(fpi);
                    red = (unsigned char) fgetc(fpi);
                    fputc((int)blue, fpo);
                    fputc((int)green, fpo);
                    fputc((int)red, fpo);
                } else {
                    blue = (unsigned char) fgetc(fpi);
                    green = (unsigned char) fgetc(fpi);
                    red = (unsigned char) fgetc(fpi);
                    fputc(blue, fpo);
                    fputc(green, fpo);
                    fputc(red, fpo);
                }
            }
            fseek(fpi, 3L, SEEK_CUR);
        }
    } else {
        blue = (unsigned char) fgetc(fpi);
        green = (unsigned char) fgetc(fpi);
        red = (unsigned char) fgetc(fpi);
        fputc(blue, fpo);
        fputc(green, fpo);
        fputc(red, fpo);
    }
}

fclose (fpm);
else {
    printf(" Unable to open mask file");
}
}

void scan_image (FILE *fpi, struct header *hdr) {
    int x,y;
    int in,temp;
    int count =0;
    scan *start;

    start = (scan *) malloc (sizeof (scan));
    start->x=-1;
    start->ylast=-20;
    start->right = NULL;
    start->left  = NULL;
    start->ydif = 0;
    find_image (fpi,hdr);
    for (y=0;y<hdr->image_height;y++) {
        for (x=0;x<hdr->image_width;x++) {
            in = (unsigned char) fgetc(fpi);
            fgetc(fpi);
            fgetc(fpi);
            if (in==255) {
                add_to_list (x,y,start);
            }
            line_look (start,&count);
        }
    }
}

void add_to_list (int x, int y, scan *start) {
    int i;
    scan *search,*news;
    search = start;
    /* Find the insert point */
    while ((search->right != NULL) && (search->right->x < x)) {
        search = search->right;
    }
    /* Bring others down */
    if (((search->x)==x) && ((search->x)==x)) {
        search->ydif = abs (search->ylast - y);
        search->ylast = y;search->x = x;
    }
}

else { /* Bring that one in */
    news = malloc(sizeof(scan));
    news->right = search->right;
    search->right = news;
    news->left = search;
    news->link = 0;
    news->x = x; news->ystart = news->ylast = y;}
}

void line_look (scan *start, int *count)
{
    int i, POSSIB;
    scan *search, *news;

    search = start;

    /*Mark if a line is present */
    while ((search->right != NULL)) {
        search = search->right;
        if (search->ylast == search->right->ylast) {
            search->link = search->link + 1;
        } else if (search->ylast == search->left->ylast) {
            search->link = search->link + 1;
        }
    }
    search = start;

    /* Look for a completed line */
    while (search->right != NULL) {
        search = search->right;
        if ((search->ydif == 0) && ((search->right->ydif > 0) || (search->left->ydif > 0))) {
            if (POSSIB == TRUE) {
                *count = *count + 1;
            }
            POSSIB = !POSSIB;
        }
    }
}

void Box_Filter (unsigned long int *image, unsigned long int *image2,
    long int x1, long int y1, long int x2, long int y2,
    node *root[DEPTH], struct header *hdr)
/***/
  /* Box_Filter will allow the user to specify a box in the image and then */

100
/* remove all colors that are within that box */

/* Expects: horiz (current horizontal position) */
/* vert (current vertical position) */
/* root (root of data structure) */
/* fpi (image file pointer) */
/* hdr (image file header) */
/* ftype (0: remove outright 1: do a color filter) */

Returns: Nothing

*******************************

int button_press=1,xo,xf,yo,yf,a,b,l_h,s_h,s_v,l_v;
int xob,yob,xfb,yfb;
long xto,yto,zto,total;
long distance,temp;

node *remove;
fremove = 0;

if (1) {
    if (x1> x2) { temp = x2; x2 = x1; x1 = temp; }
    if (y1> y2) { temp = y2; y2 = y1; y1 = temp; }

    for (a = x1;a<x2+1;a++) { /* Remove colors from cube */
        for (b = y1;b<y2+1;b++){
            remove = Get_Start(a,b,root,hdr,image);

            if (remove!=NULL) {
                if (remove->count != 0) {
                    remove->count = 0;
                    remove->type = filter_number;

                    printf(" Number removed: %d \n",fremove);

                    fremove++;
                }
            }
        }
    }
}

void chromo_display (node *root[DEPTH])
{

long int Chroma;
node *current;
int temp,x,y;
short color[3],a,b,c;
long pnt[2];
int i,r,g,count=0;
unsigned long data;
prefsize (475,350);
Chroma = winopen("Chromaticity");
zbuffer(0);
singlebuffer();
RGBmode();
gconfig();
clear();
for (y = 0;y<200;y++) {
  pnt[1] = y + 25;
  for (x=0;x<200-y;x++) {
    bgnpoint();
    pnt[0] = x;
    color[0] = 200 - y - x;
    color[1] = y;
    color[2] = x;
    c3s(color);
    v2i(pnt);
    endpoint();
  }
}
for (i=0;i<DEPTH;i++) {
  current = root[i];

  while (current!=NULL){
    data = current->data;

    if ((current->x + current->y + current->z) != 0 ) {
      bgnpoint();
      color[0] = current->x;
      color[1] = current->y;
      color[2] = current->z;
pnt[0] = 250 + (long)(current->z) * 200 / (long)(current->x + current->y +
current->z);
current->z);

/*
 printf(" %d %d %d %ld %ld \n",current->x,current->y,current->z,pnt[0],pnt[1]);*/
c3s(color);
v2i(pnt);
endpoint();

}

current = current->out;
}

void show (node *root[DEPTH]) {

int i,count;
node *current;

for (i=0;i<DEPTH;i++) {
 printf("------------------ %d -----------\n",i);
 for (current=root[i];current!=NULL;current=current->out) {
 printf(" %d %d %d %ld\n",current->x,current->y,current->z,count);
 count++;
 }
}

void rgb_data (int x, int y, int *r, int *g, int *b, unsigned long int *image, struct
header *hdr )
{
 unsigned long data;
 data = image[ y * hdr->image_width + x];
 *r = (int) ( data & 0xFF );
 *g = (int) ((data & 0xFF00 ) >> 8);
 *b = (int) ((data & 0xFF0000 ) >> 16);
}
REFERENCES

