Computer Systems Lab Project 2007-2008

The Applications of Image Processing Techniques to Sign Language Recognition

Byron Hood

Version 0.3

04/02/08

Table of Contents

Syslab Tech Project 2007-08 Data Structure Index

Here are the data structures with brief descriptions:

Syslab Tech Project 2007-08 File Index

Here is a list of all files with brief descriptions:

Syslab Tech Project Data Structure Documentation

chain Struct Reference

`#include <chain.h>`

Data Fields

1 int start_x
2 int start_y
3 int end_x
4 int end_y
5 int num_lines
6 double abs_len

7 double real_len
8 line ** members

Detailed Description

The structure which represents a chain, or group of lines. Usually, a chain of lines has a general sense of direction, and although it may not be perfectly straight, it will be reasonably close.

Field Documentation

int chain::start_x

The leftmost x-value of the chain.

int chain::start_y

The y-value of the point corresponding to the leftmost x-value of the chain.

int chain::end_x

The rightmost x-value of the chain.

int chain::end_y

The y-value of the point corresponding to the rightmost x-value of the chain.

int chain::num_lines

The number of lines in the chain.

double chain::abs_len

The length of the chain from starting to ending points in a straight line.

double chain::real_len

The actual length of the chain around whatever curve it follows.

line** chain::members

The lines that are members of the chain.

---

hand Struct Reference

`#include <interpret_lines.h>`

Data Fields

9 int pinky
10 int ring_finger
11 int middle_finger
12 int index_finger
13 int thumb
14 int orientation

15 char letter

Detailed Description

A structure to represent a hand and all of its fingers' positions, the hand's orientation, and anything else that is necessary.

Field Documentation

int hand::pinky

The position of the pinky.

int hand::ring_finger

The position of the ring finger.

int hand::middle_finger

The position of the middle finger.

int hand::index_finger

The position of the index finger.

int hand::thumb

The position of the thumb.

int hand::orientation

The orientation of the hand as defined in the hand orientation enum.

char hand::letter

The letter to which this hand and finger position corresponds in American Sign Language.

---

image Struct Reference

`#include <imagelib.h>`

Data Fields

16 char * name
17 int ** data
18 int format
19 int rows
20 int cols
21 int data_rows
22 int data_cols

Detailed Description

A structure designed to contain all of the information necessary to store an image and all of its associated details such as format, size, and so on.

Field Documentation

char* image::name

The filename of the image, minus the extension which determines the image's type.

int** image::data

The actual image data. Since inages are always handled in B&W there is no need for any further storage space or clever storing algorithms that might use the various bytes of the integer.

int image::format

The image format as specified in the enum below. Most popular formats are supported.

int image::rows, int image::cols

The number of rows and columns in the image.

int image::data_rows, int image::data_cols

The number of rows and columns in the matrix itself. This can only be changed by either an image expansion or by lock_image_resize().

---

line Struct Reference

`#include <line.h>`

Data Fields

23 int start_x
24 int start_y
25 int end_x
26 int end_y
27 double slope
28 double length
29 int thickness

Detailed Description

The structure in which the program stores information about a line. It has some useful stuff in it to facilitate use later on.

Field Documentation

int line::start_x

The leftmost x-value of the line.

int line::start_y

The y-value of the left end of the line.

int line::end_x

The rightmost x-value of the line.

int line::end_y

The y-value of the right end of the line.

double line::slope

The slope of the line.

double line::length

The length of the line.

int line::thickness

The thickness, in pixels, of the line.

---

list Struct Reference

`#include <list.h>`

Data Fields

30 void ** array
31 unsigned long int numitems
32 size_t ptr_size

Detailed Description

The list structure: an array of changeable size that can be added to and subtracted from. It is more versatile that an array in that it does not depend on having a pre- defined size.

Field Documentation

void** list::array

The array itself which can be mutated as needed to make sure it is full.

unsigned long int list::numitems

The number of items in the array.

size_t list::ptr_size

The size of each item/pointer in the list.

---

timestamp Struct Reference

```
#include <timer.h>
```

Data Fields

33 struct timeval start
34 struct timeval end
35 struct timeval inactive
36 int status

Detailed Description

A structure to hold data about a span of time. This includes data about when the time was started, when it stopped (if this has occurred yet), and how much time the timer has been inactive (if it has been stopped and then restarted).

Field Documentation

struct timeval timestamp::start `[read]`

The starting time of the timer.

struct timeval timestamp::end `[read]`

The ending time of the timer.

struct timeval timestamp::inactive `[read]`

The time that the timer became inactive.

int timestamp::status

The status of the timer (any of not started, started, stopped, restarted).

Syslab Tech Project 2007-08 File Documentation

src/chain.h File Reference

```
#include <stdarg.h>
#include "main.h"
#include "log.h"
#include "line.h"
```

Data Structures

37 struct chain

Functions

38 int addable (chain *, line *)
39 void add_to_chain (chain *, line *)
40 chain * chain_lines_args (int,...)
41 chain * chain_lines_array (int, line **)
42 int chainable (line *, line *)
43 chain * empty_chain ()
44 void resize_member_list (chain *, int)

Function Documentation

void add_to_chain (chain *, line *)

Add the given line to a chain. The program appends the line to the chain in the most logical place, usually determined with the line's x-coordinates, but also with the y-coordinates of its start and end points.

int addable (chain *, line *)

Almost exactly like chainable(), this function decides on whether or not a line should be _added_ to an already existing chain.

chain* chain_lines_args (int, ...)

This is one of several functions available to construct a chain of lines. This one takes a variable number of arguments and puts it all together to make a line chain, the basic unit of my line interpretation.

chain* chain_lines_array (int, line **)

Another one of the several functions responsible for the making of line chains based on a group of lines. This particular function makes the line chain based on a list (array) of lines passed to it.

int chainable (line *, line *)

This function, similar in a way to linecmp() of the line untility function set, decides if two lines are "chain- able" or, in other words, if they should be put together in a chain.

chain* empty_chain ()

Create an empty chain structure.

void resize_member_list (chain *, int)

Change the size of a chain's member list. This preserves any existing data, providing that the list is not made any shorter. Any data that is orphaned off of the end of a shortened list will be lost.

src/edge_detect.h File Reference

```
#include "main.h"
#include "log.h"
#include "timer.h"
```

Functions

45 int main (int argc, char *argv[])

Variables

46 char * infilename
47 char * outfilename
48 FILE * infile
49 FILE * outfile
50 unsigned char image1 [maxsize][maxsize]
51 unsigned char image2 [maxsize][maxsize]
52 int rows
53 int cols
54 int maxpixel
55 char format [3]

Function Documentation

int main (int *argc*, char * *argv*[])

The main function.

Variable Documentation

unsigned char image1[maxsize][maxsize]

The image data, in a matrix of size maxsize by maxsize. The default is 1000x1000 (and even this requires practically two to four megabytes of memory per matrix), and it is not adivsable to significantly increase this, even more so when increasing the dimensions provokes an squared increase in matrix size.

unsigned char image2[maxsize][maxsize]

Another image.

FILE* infile, outfile

The image input and output files.

char* infilename, outfilename

The filenames of the input and output files. Eventually, these may go away and become automatically generated (as even 5 frames per second makes manual naming impractical and this should be able to cope with 20-30 fps, depending on the platform. Additionally, to enhance readability, the file pointers are also here.

src/find_lines.h File Reference

```
#include <signal.h>
#include "main.h"
#include "log.h"
#include "timer.h"
#include "line.h"
```

Enumerations

56 enum boundary { OUT_BOUNDS_ABOVE, OUT_BOUNDS_BELOW, IN_BOUNDS }

Functions

57 void branch_out (int, int, double, line *, int)
58 int check_bounds (int, int, int)
59 int detect_lines (double)

Variables

60 enum boundary b
61 FILE * infile
62 int image [IMAGE_MAX_DIMENSION][IMAGE_MAX_DIMENSION]
63 line ** lines
64 int num_lines
65 int rows
66 int cols
67 int maxpixel
68 char format [3]

Enumeration Type Documentation

enum boundary

Used for boundary checking for the branch_out() function. Each value describes one of the three possible conditions of the index about to be used.

Enumerator:

OUT_BOUNDS_ABOVE
OUT_BOUNDS_BELOW
IN_BOUNDS

Function Documentation

void branch_out (int, int, double, line *, int)

Once a point is determined to be highlighted, branch out from that point at all thetas and try to find any lines which emanate from it.

Input: int row: The current row coordinate in the matrix. int col: The current column coordinate in the matrix. int rate: The rate at which to cycle through thetas looking for lines (given in radians). Best set to somewhere around pi / 10. line* result: A line struct which is ready for the function to set the result. If no line is detected, then result->start_x is set to INITIAL_VALUE and the function ends.

int check_bounds (int, int, int)

The function to help the program constrain the coordinates it uses within the image matrix to make sure no negative indices are ever used, something which results in the dreaded seg fault.

int detect_lines (double)

The actual line detector. The general method is to iterate through angles at a given rate (best is ~(pi / 10) rads) and look through the image for lines at that particular angle.

Variable Documentation

int image[IMAGE_MAX_DIMENSION][IMAGE_MAX_DIMENSION]

The image data, in a matrix of size IMAGE_MAX_DIMENSION square.

Definition at line 48 of file find_lines.h.

FILE* infile

The pointer to the input file from which image data is read. This is set by the user when running the program. If a file does not exist or none is specified, then the program throws a fatal error and exits.

line** lines

An array containing all of the data specifying what lines exist and where they exist, and so on.

int num_lines

The global for the number of lines being stored.

src/imagelib/imagelib.h File Reference

```
#include <jpeglib.h>
#include <libpng12/png.h>
#include <gif_lib.h>
#include <tiff.h>
#include "main.h"
#include "log.h"
```

Data Structures

69 struct image

Enumerations

70 enum formats { IMAGE_FORMAT_RAW, IMAGE_FORMAT_JPEG, IMAGE_FORMAT_PNG, IMAGE_FORMAT_TIFF, IMAGE_FORMAT_GIF }

Functions

71 void change_image_format (image *, int)
72 void change_image_name (image *, char *)
73 void resize_image (image *, int, int)
74 void lock_image_resize (image *)

Variables

75 enum formats image
76 enum formats void
77 enum formats bool
78 enum formats int

Enumeration Type Documentation

enum formats

The different supported image formats. These are also limited by what libraries are already on the machine. The program will not, for example, be able to read JPEG format images without libjpeg, or PNG without libpng. All images are handled in balck and white to save space in memory and on disk. Although they may be read with color, they will be saved using black and white and also handled in black and white.

Enumerator:

IMAGE_FORMAT_RAW Raw image format. In other words, the image data is stored in a file with no compression at all.

IMAGE_FORMAT_JPEG The JPEG image format. This uses a cosine-based transform and heavy compression techniques to save space at the cost of image quality. Unlike other formats, JPEG is by definition lossy.

IMAGE_FORMAT_PNG The PNG image format. This, unlike JPEG, is a lossless image format with a very good technique for compression.

IMAGE_FORMAT_TIFF The TIFF image format. TIFFs are sometimes used in scanning and digital imagery because they are a compromise between JPEG and raw.

IMAGE_FORMAT_GIF One of the oldest image formats is the Compuserve GIF format. It is still viable since most other image formats do not specify transparency well.

Function Documentation

void apply_image_callback (image *, void*, bool, int)

This will perform a specific function on each and every pixel in an image. It is highly useful for certain types of transformations. If the callback function is NULL, no action is taken and this function returns normally. If a callback fails then this function will return the called function's error code. All callbacks must:

79 return a floating-point value (errcodes -> negative)
80 accept as an argument a double
81 if the require_neighbors value is set then it must also take an argument that is an array of doubles with eight members; the array is populated from left to right and then top to bottom.

If the require_neighbors parameter is set, then a second data set is created transparently and the results from each callback are stored in the second data matrix. When all the callbacks are done, then the function replaces (destructively) the existing matrix with the new one.

void change_image_format (image *, int)

Changes the image format from one supported type to a different supported type.

void change_image_name (image *, char *)

Changes the name of the image (without extension) to the provided value. If the value provided is NULL then the old name is kept.

void lock_image_resize (image *)

This is a security feature so that unwanted resizing can be corrected if necessary; also it allows for

reverting instead of copying data, yielding a large speedup. Until this is called, the actual data matrix is unchanged except to resize it to make it larger. If the dimensions are smaller all that is changed are the ->rows and ->cols properties. Once this is called, though, it, with memory operations, resizes the actual data matrix. THIS WILL DELETE DATA.

void resize_image (image *, int, int)

Resize the image to the given size in rows and columns. Non-rectangular images are not supported. If either of the new image's dimensions are greater than the value of IMAGE_MAX_DIMENSION as defined in main.h, then that one is clipped to IMAGE_MAX_DIMENSION. If the dimension(s) provided are smaller than the ones currently in the image, the new dimensions are applied but the size of the data does not change. In other words the resizing is reversible until lock_image_data() is called. If the image dimensions are larger then no data is lost; lock_image_resize() then has no effect.

---

src/imagelib/read_image.h File Reference

```
#include "imagelib.h"
```

Functions

82 image * open_image (char *)
83 void read_gif_image (image *, FILE *)
84 void read_jpeg_image (image *, FILE *)
85 void read_png_image (image *, FILE *)
86 void read_raw_image (image *, FILE *)
87 void read_tiff_image (image *, FILE *)

Function Documentation

image* open_image (char *)

Open and read an image file. This assumes that the path to the image is relative to the current execution path, (and that all implied subdirectories already exist) unless the first character of the path is a slash. If the image is in raw format, the data, in an image struct, is returned. If the image is a compressed format, on the other hand, then the image is first decompressed and put into usable form before being returned in the image structure.

void read_gif_image (image *, FILE *)

Read and process a Compuserve GIF image and put the data into an image structure provided.

void read_jpeg_image (image *, FILE *)

Read and process a JPEG format image and then add the image data into an image structure provided.

void read_png_image (image *, FILE *)

Read and process a PNG format image and add the resulting image data to the structure provided.

void read_raw_image (image *, FILE *)

Read in and process image data from a raw image format, then place the result in the data value of the image structure provided.

void read_tiff_image (image *, FILE *)

Read in and process an image of TIFF format. The result data is placed in the given structure.

src/imagelib/write_image.h File Reference

```
#include "imagelib.h"
```

Functions

88 int write_image (image *i)
89 void write_gif_image (image *, FILE *)
90 void write_jpeg_image (image *, FILE *)
91 void write_png_image (image *, FILE *)
92 void write_raw_image (image *, FILE *)
93 void write_tiff_image (image *, FILE *)

Function Documentation

void write_gif_image (image *, FILE *)

Write a Compuserve GIF image to disk and return an int that indicates how the fnction performed. If the return value is a 0, then the fucntion executed correctly, and any processing should continue. Nonzero return codes indicate what error occurred; they indicate that the function did not terminate properly.

int write_image (image * *i*)

Save existing image data to disk and return a result code based on whether or not the operation succeeded. As per convention, a return value of 0 indicates that no error occurred, while any other return value indicates an error at some point. If the image has been resized, but those changes have not been locked in, and the resizing was not an expansion (in other words the resize would lose data), then a warning is generated that changes will be lost. The data saved will reflect the old dimension and not the new ones.

void write_jpeg_image (image *, FILE *)

Write a JPEG format image to disk and return a status code that indicates the success or reason for failure of the function.

void write_png_image (image *, FILE *)

Write a PNG format image to disk and return a status code which will inform the caller if the write was successful or if not, what error occurred.

void write_raw_image (image *, FILE *)

Write image data to disk from a raw image, and return a status code indicating whether or not it was

successful.

void write_tiff_image (image *, FILE *)

Write an image to disk of TIFF format. The function will return an integer representing the error code which the function encountered, or 0 if no errors occurred.

src/interpret_lines.h File Reference

```
#include "main.h"
#include "line.h"
#include "chain.h"
```

Data Structures

94 struct hand

Functions

95 chain ** make_chains (line **, int)
96 int find_hand_orientation (chain **)
97 int find_pinky_position (chain **)
98 int find_ring_finger_position (chain **)
99 int find_middle_finger_position (chain **)
100 int find_index_finger_position (chain **)
101 int find_thumb_position (chain **)
102 int match_hand_position (hand *)

Function Documentation

int find_hand_orientation (chain **)

Determine the hand orientation according to the provided chain list. This just examines the list and does not in any way mutate or change it. The return value is one of the values in the orientation enum above.

int find_index_finger_position (chain **)

Same thing but for the index finger.

int find_middle_finger_position (chain **)

Go through the same process for the middle finger. Again we mutate the list to delete used chains.

int find_pinky_position (chain **)

Find the pinky using the chain list provided. This is the first in several steps of interpreting the hand. When a chain is used to identify a finger, it is subsequently deleted so as not to confuse the rest of the interpreter. This function returns one of the values detailed in the finger enum above, which is then used as the position of that finger.

int find_ring_finger_position (chain **)

This is about the same as the pinky finder except that it works on the ring finger. After finishing with the chain or chains for the ring finger it deletes those.

int find_thumb_position (chain **)

And now, finally, the thumb.

chain make_chains (line **, int)**

Go through the remaining lines after the unnecessary ones have been filtered out, and chain them together if they ought to be chained. Then return a list of the chains the function has created. The list of lines will be changed such that the lines remaining are those which did not become part of any chain. This is the procedure: As we go through the lines array, we make chains. Add a line to a single-line chain if we aren't working on any particular chain right now. (a) For next lines, see if they can be added to that chain too; if they can, add them (b) We must start over analyzing the remaining lines after adding a new line because the new line could affect the outcome of other lines' beign assigned to that chain. NOTE: when adding a line delete it from the main lines array.

int match_hand_position (hand *)

Now we match the given positions against ones stored in XML files. If there is a match we say so and if no match is found then we either go back and do it again or we go on to the next image. The return value indicates if a match has occurred. Any such match will be stored in the hand structure.

src/line.h File Reference

```
#include "main.h"
#include "log.h"
```

Data Structures

103 struct line

Functions

104 int similar (line *, line *)
105 int linecmp (line *, line *)

Function Documentation

int linecmp (line *, line *)

Compare two lines for gross concepts of similarity.

int similar (line *, line *)

Whether or not two lines are similar enough to be counted as similar.

src/list.h File Reference

```
#include <stdarg.h>
#include "main.h"
```

Data Structures

106 struct list

Defines

107 #define POSITION_START 0
108 #define POSITION_END -999

Functions

109 list * blank_list (size_t)
110 list * new_list (size_t, unsigned long int)
111 list * init_list (size_t, unsigned long int, void **)
112 list * init_list_varg (size_t, unsigned long int,...)
113 list * init_list_array (size_t, unsigned long int, void **)
114 void push (list *, void *)
115 void append (list *, void *)
116 void * pop (list *)
117 void add (list *, void *, unsigned long int)
118 void * remove (list *, unsigned long int)
119 void * peek (list *, unsigned long int)
120 void set (list *, void *, unsigned long int, bool)
121 void apply_void_callback ((void *)(void *, unsigned long int))
122 void apply_return_callback ((void *)(void *, unsigned long int))

Define Documentation

#define POSITION_END -1

The end of the list. Since a list may never have negative indices -1 is safe.

#define POSITION_START 0

The position implied for the start of the list, almost always index 0.

Function Documentation

void add (list *, void *, unsigned long *int*), void append (list *, void *)

Add an item at an arbitrary position in the list. If the index provided is out of bounds, then it is rounded off to the limit at that end (e.g. -100 => 0, while a large number like 1000000 => numitems). Two special values are available for this: POSITION_START and POSITION_END, defined above.

void apply_return_callback ((void *)(void *, unsigned long int))

Apply a callback function to each individual item in the list and replace each item with the return value. The callback must accept as arguments a void* pointer and an unsigned long int which represents the index.

void apply_void_callback ((void *)(void *, unsigned long int))

Apply a void callback function to each individual item in the list. This callback function must accept as arguments a void* pointer and a ulong int. The first is the data item, the second is the index of the item. Any return value will be ignored.

list* blank_list (size_t)

Create a new list structure with the given pointer size and type, and initialize it to be blank.

list* init_list (size_t, unsigned long *int*, void **)

Initialize a new list to have the given elements to start out. The default is an array argument.

list* init_list_array (size_t, unsigned long *int*, void **)

list* init_list_varg (size_t, unsigned long *int*, ...)

list* new_list (size_t, unsigned long *int*)

Create a new list structure with a certain pre-allocated size.

void* peek (list *, unsigned long *int*)

Return, but do not remove, an item at an arbitrary index. Again this function will accept the POSITION_START and POSITION_END macros.

void* pop (list *)

Remove the last item in the list and shorten the list and return it. This is in constant time.

void push (list *, void *)

Add an item to the end of the list. This can be done in constant time.

void* remove (list *, unsigned long *int*)

Remove an item from an arbitrary position in the list and return it. The behavior of this function is similar to that of add() in that it will round to the closest legal index. This function also accepts the arguments POSITION_START and POSITION_END.

void set (list *, void *, unsigned long *int*, bool)

Set the value of an arbitrary item in the list to the given value. This functions accepts as indices the macros POSITION_START and POSITION_END.

src/log.h File Reference

```
#include "main.h"
```

Enumerations

123 enum { LEVEL_DEBUG, LEVEL_OUTPUT, LEVEL_WARNING, LEVEL_ERROR, LEVEL_FATAL }

Functions

124 void die (char *)
125 void debug (char *)
126 void error (char *)
127 void fatal (char *)
128 void finish ()

129 void log_file (int, char *)
130 void out (char *)
131 void sys_error (int err, void(*func)(char *))
132 void warn (char *)
133 void warning (char *)

Variables

134 FILE * logfile
135 int fileoff
136 int errorcount
137 int warncount

Enumeration Type Documentation

anonymous enum

The various levels of output which this program handles, from simple debug or verbose output to fatal errors such as missing arguments. Each has its own particular species of output and also registers differently in the log file to clarify which errors or messages were related to a crash or failure, if necessary.

Enumerator:

LEVEL_DEBUG
LEVEL_OUTPUT
LEVEL_WARNING
LEVEL_ERROR
LEVEL_FATAL

Function Documentation

void debug (char *)

The method to output debug information, especially when attempting to pinpoint the sources of errors and segmentation faults. This prints out a notice and also logs it to the runtime log file.

Input: char* message: the debugging message designed to help with crashes.

void die (char *)

The method to call when an error destabilizes the program or interrupts the flow of information. Basically, if an error is serious enough, then this method must be called to prevent the program from doing any damage to the operating system or files. This method outputs and saves to disk an error message and then exits.

Input: char* message: A pointer to any error message to include

void error (char *)

The method to call for a non-fatal error, which does not jeopardize the operation of the program or threaten to destabilize it (and perhaps the system). It prints out an error message and ups the error counter, but does not kill the program.

Input: char* message: theerror message which should gave a hint about cause of the error for future reference and fixing.

void fatal (char *)

An alias for die() above. In case I forget to call die() and try the next most logical choice.

Input: char* message: the message passed along to die() to initiate the process associated with a fatal error.

void finish ()

When we are ready to exit, call this to report the overall error total and save this to disk, then perform any other cleanup operations that are necessary. However, do not exit, but leave the exiting program or method to decide the manner of exit (e.g. EXIT_SUCCESS or EXIT_FAILURE).

No input or return value.

void log_file (int, char *)

The general purpose method which takes a message and error level, and writes this to the log in the appropriate manner. The advantage of going this way is having a standard way of writing everything to the log file so that one change changes the entire format.

Input: (1) int level: The level of output, from LEVEL_FATAL representing a total program collapse to LEVEL_DEBUG indicating that it is only printed if debug is enabled. (2) char* message: the message being written to the log file.

void out (char *)

This method handles any official program output so that this is logged in the runtime log file and so that it looks decent on the screen as it comes out.

The idea behind doing this here is to standardize any appearances for output used. On the whole, this should be sparing, and instead it is better to use debug() and and warn() unless the message is truly program output (such as a letter which has been determined).

Input: char* message: The message to be printed to standard output in addition to the log file.

void sys_error (int *err*, void(*)(char *) *func*)

This function takes care of any business with errno and system errors so that the other programs aren't burdened by unnecessary details on logging.

Input: (1) int err: The number of the error. Often, but not always, this is errno. (2) void (*func)(char*): the function to call, be it warn(), die(), or a simple error(). this is to allow the program the maximum choice in error severity.

void warn (char *)

This function exists to warn the user of a situation which could cause errors in the future. This does not describe anything going wrong yet, but circumstances are similar to situations in which things could fail.

Input: char* message: the warning message to give the user to warn them that circumstances which could bring about a crash exist.

void warning (char *)

An alias for warn(), in case I attempt to call an alternate warning function.

Input: char* message: the message to be passed along to warn().

Variable Documentation

int errorcount, warncount

Counter numbers for errors and warning issued during the course of the program. These remain constant over multiple calls and instances of the program so as to simply total the number over the entire run time. Each error that is generated increments the error counter, regardless of whether or not it is fatal. Each warning EXCEPT a log file warning (if the file becomes unavailable) increments this counter.

When the program exits, this method is called to finish up and print out the number of errors and warnings which it received over the course of its run time.

int fileoff

This is the variable which decides whether or not file logging is turned off. If any errors with respect to opening or closing this file occur, then this is set to a non- zero value and any further error, warning, and debug requests will only be printed to either stdout or stderr.

FILE* logfile

The pointer to the log file. This is used to simultaneously print error, warning, and debug messages out to stderr as well as the file (so that it is all preserved for later inspection, if there is a preponderance of output). If any file operations fail on this file, any logging is immediately turned off and a message of level LEVEL_WARNING is printed.

src/main.h File Reference

```
#include <math.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
```

Defines

```
138 #define INITIAL_VALUE  -9999
139 #define IMAGE_MAX_DIMENSION  1000
140 #define HIGHLIGHT_THRESHOLD  60
141 #define LINE_LENGTH_THRESHOLD  5
142 #define round(x)  ((x - floor(x) < 0.5) ? (int)floor(x) : (int)ceil(x))
143 #define sq(x)  pow(x,2)
144 #define min(x, y)  ((x <= y) ? x : y)
145 #define min3(x, y, z)  min(x, min(y, z))
146 #define min4(w, x, y, z)  min(min(w, x), min(y, z))
147 #define max(x, y)  ((x >= y) ? x : y)
148 #define max3(x, y, z)  max(x, max(y, z))
149 #define max4(w, x, y, z)  max(max(w, x), max(y, z))
150 #define avg2(x, y)  ((1.0 * (x + y)) / 2.0)
151 #define avg3(x, y, z)  ((1.0 * (x + y + z)) / 3.0)
152 #define avg4(w, x, y, z)  ((1.0 * (w + x + y + z)) / 4.0)
```

Define Documentation

#define avg2(x, y) ((1.0 * (x + y)) / 2.0), #define avg3(x, y, z) ((1.0 * (x + y + z)) / 3.0), #define avg4(w, x, y, z) ((1.0 * (w + x + y + z)) / 4.0)

This macro takes the average value of a specific number of values. It is another one of the easier-to-define-a- single-time functions like min() and max().

#define HIGHLIGHT_THRESHOLD 60

The value, on a scale of 0-255 (where 0 is pitch black and 255 is pure white), defining the minumum brightness a pixel must have to be considered "on." The primary application is for my edge-detection and my finding algorithms to be able to discern whether one particular pixel is bright enough to be worth counting.

#define IMAGE_MAX_DIMENSION 1000

This defines the maximum height or width that an image will be allowed to have before being cropped (truncated) at the 1000th pixel. After the 1000th pixel for a given row of an image, the reader will continue to read the characters for the line but will immediately discard them. Once it reaches the next line, it restarts storing data.

#define INITIAL_VALUE -9999

The initial value of structure variables set to avoid any confusion as to the results of a given function or process run using that structure. This is used especially in functions which would, under normal circumstances, return a structure (pointer) to the calling function, but which need to return an error indicating that the operation of the function was either unsuccessful or failed due to some system error.

#define LINE_LENGTH_THRESHOLD 5

The minimum length, divided by two, that a line must have to be considered a line. Collections of points that are not strung together in groups less than this * 2 are just that -- collections of points.

#define max(x, y) ((x >= y) ? x : y), #define max3(x, y, z) max(x, max(y, z)), #define max4(w, x, y, z) max(max(w, x), max(y, z))

Use a macro to find the maximum of two or more numbers of arbitrary type. Like with the min() macro, max() can relieve some of the ugliness from the coding that I have to do (the compiler gets the nasty stuff either way but at least it doesn't have feelings). Like with min(), max() will only return a value besides the first one provided if it is strictly greater than the first value.

#define min(x, y) ((x <= y) ? x : y), #define min3(x, y, z) min(x, min(y, z)), #define min4(w, x, y, z) min(min(w, x), min(y, z))

Find the minimum of two or more numbers. This type of operation is a surprisingly burdensome item to code with consistency, especially when repeated over and over. The resulting optimization is much more significant than the slight coding time optimization given by using a macro sq(x) instead of pow(x,2). Behaviorwise, this macro returns a subsequent value only if it is strcitly greater than the first. If the first and the subsequent values are equal, then the macro uses the first one.

#define round(x) ((x - floor(x) < 0.5) ? (int)floor(x) : (int)ceil(x))

This macro rounds a number of arbitrary type (presumed to be a double or float) to the nearest integer, and returns an integer representation of this value. Rounding comes in awfully useful when I am forced into losing precision, primarily when calculating the values of pixels during processes like edge detection.

#define sq(x) pow(x,2)

The C standard command to square a number x is, like in many languages, somewhat cumbersome compared to usual notation, especially as written out by hand. Therefore the sq() macro is here to more closely emulate the total ease of putting a superscripted two than using pow().

src/timer.h File Reference

```
#include <time.h>
#include <sys/time.h>
#include "main.h"
#include "log.h"
```

Data Structures

153 struct timestamp

Enumerations

154 enum states { TIMER_STARTED, TIMER_STOPPED, TIMER_RESTARTED, TIMER_NEW }

Functions

155 timestamp * init_timer ()
156 void start_timer (timestamp *)
157 void stop_timer (timestamp *)
158 void restart_timer (timestamp *)
159 long running_time (timestamp *)
160 long total_time (timestamp *)

Enumeration Type Documentation

enum states

The possible values of the status variable for a timer, so that the following methods can determine their behavior.

Enumerator:

TIMER_STARTED
TIMER_STOPPED
TIMER_RESTARTED
TIMER_NEW

Function Documentation

timestamp* init_timer ()

Initialize a timer.

void restart_timer (timestamp *)

Restart a timer.

long running_time (timestamp *)

Count the amount of time that a timer has been running.

void start_timer (timestamp *)

Start a timer.

void stop_timer (timestamp *)

Stop a timer.

long total_time (timestamp *)

Count the total time since a timer was started first.