The Applications of Image Processing Techniques to Sign Language Recognition with Webcams

version 0.2

Byron Hood

January 18, 2008

Table of Contents

Syslab Tech Project 2007 Data Structure Index

Syslab Tech Project 2007 Data Structures

Here are the data structures with brief descriptions:

Syslab Tech Project 2007 File Index

Syslab Tech Project 2007 File List

Here is a list of all files with brief descriptions:

Syslab Tech Project 2007 Data Structure Documentation

line Struct Reference

#include <line.h>

Data Fields

1 int start_x
2 int start_y
3 int end_x
4 int end_y
5 double slope
6 double length
7 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 starting x coordinate of the line. This is always further to the left, or at the same x-value as the line's ending x-value. This way it is known for sure that the line will always "go" from left to right.

int line::start_y

The line's starting y-coordinate. There is no restriction on this to be greater or smaller than the final y-coordinate, because any such restriction would eliminate half of all possible lines.

int line::end_x

The x-coordinate of the end of the line. It will always be greater than/equal to the starting x.

int line::end_y

The ending y-coordinate of the line.

double line::slope

The slope of the line calculated using $\sqrt{\frac{(y_{end}-y_{start})}{(x_{end}-x_{start})}}$ when the line is created.

double line::length

The lengh of the line, calculated using $\sqrt{(x_{start}+x_{end})^2+(y_{start}+y_{end})^2}$.

int line::thickness

The thickness of the line. This is set as the program goes along detecting lines. Whenever another line is found to be one pixel away, the thickness is incremented and the other line is destroyed, saving not only memory but headaches in interpretation.

---

timestamp Struct Reference

#include <timer.h>

Data Fields

8 struct timeval start
9 struct timeval end
10 struct timeval inactive
11 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 time, to the precision of a microsecond, of when the timer was started.

struct timeval timestamp::end `[read]`

The time, to the precision of a microsecond, of when the timer was stopped.

struct timeval timestamp::inactive `[read]`

Since a timer may be restarted, this records, to the microsecond, any time during which the timer has been inactive.

int timestamp::status

The status can be one of the following: TIMER_NEW, TIMER_STARTED, TIMER_STOPPED, TIMER_RESTARTED.

Syslab Tech Project 2007 File Documentation

code/find_lines.c File Reference

`#include "find_lines.h"`

Functions

12 void branch_out (int row, int col, double rate, line *result, int c)
13 int check_bounds (int d, int cur, int tot)
14 int detect_lines (double rate)
15 int main (int argc, char *argv[])

Function Documentation

void branch_out (int *row*, int *col*, double *rate*, line * *result*, int *c*)

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 -999 and the function ends.

Here do some manipulation so that dcol is always positive, and if any, drow is the one that is negative. I have had problems with the program dealng with negative values of either drow or dcol and this is part of the fix.

This part is where boundaries of the image are tested. If we are outside the boundaries then we set the end to be at the boundary and set the reached_(whichever) flag.

Since dcol is _always_ greater than 0, we can eliminate some of the tests for dcol. The tests must still be performed, however, for drow.

Now expand the line on the higher-x side, if possible, and if the end has not been reached.

Expand the line on the lower-x side, if possible, and if the start has not been found already.

int check_bounds (int *d*, int *cur*, int *tot*)

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 *rate*)

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.

code/find_lines.h File Reference

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include "log.h"
#include "timer.h"
#include "line.h"
```

Defines

16 #define maxsize 1000
17 #define HIGHLIGHT_THRESHOLD 60
18 #define round(x) ((x-floor(x) < 0.5) ? (int)floor(x) : (int)ceil(x))
19 #define sq(x) pow(x,2)

Enumerations

20 enum boundary { OUT_BOUNDS_ABOVE, OUT_BOUNDS_BELOW, IN_BOUNDS, OUT_BOUNDS_ABOVE, OUT_BOUNDS_BELOW, IN_BOUNDS }

Functions

21 int check_bounds (int, int, int)
22 void branch_out (int, int, double, line *, int)
23 int detect_lines (double)

Variables

24 enum boundary b
25 int mem_alloc = 0
26 int free_count = 0
27 char * infilename
28 FILE * infile
29 int image [maxsize][maxsize]
30 line * lines [10000]
31 int l_c
32 int rows
33 int cols
34 int maxpixel
35 char format [3]

Define Documentation

#define HIGHLIGHT_THRESHOLD 60

The minimum brightness of a pixel to be considered "on"

#define maxsize 1000

The maximum dimensions of an image. Anything larger will be truncated.

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

Abusing the preprocessor to define a really simple method ;-)

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

Abuse it some more >)

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

OUT_BOUNDS_ABOVE
OUT_BOUNDS_BELOW
IN_BOUNDS

Variable Documentation

int cols

The number of columns in the image.

char format[3]

The format of the image.

int free_count = 0

The amount of memory freed.

int image[maxsize][maxsize]

The image data, in a matrix of size maxsize by maxsize.

FILE* infile

The input image file.

char* infilename

The name and pointer to the input file from which image data is read. These are 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 immediately. Although this behavior is strange, it makes the overall application a little bit more stable.

int l_c

A line counter.

line* lines[10000]

A list of all the lines in the image.

int maxpixel

The maximum value of a pixel.

int mem_alloc = 0

The amount of memory that has been dynamically allocated.

int rows

Some information about the image in question, such as how many rows and columns, as well as the maximum number of colors the image has. This is necessary information when it is printed as the header of the new image. The format string is necessary to hold information on the image type.

---

code/gl_find_lines.c File Reference

This is really the same as find_lines but with some GL output attached. Therefore, to save paper, I am cutting this file's documentation off.

---

code/line.c File Reference

```
#include "line.h"
```

Functions

36 int linecmp (line *l1, line *l2)
37 int similar (line *l1, line *l2)

Function Documentation

int linecmp (line * *l1*, line * *l2*)

This assumes that the lines are similar and compares them to determine the degree of similarity. Returns 0 if the lines are the same, or one line contains the other, and 1 or 2 if the lines are not the same but are next to each other; the number corresponds to the line which should be kept while the other is merged into it. If the lines are not the same or at all related, the function will return 3.

TODO: This function is currently limited based on the assumption that two lines with thicknesses of greater than one will not be passed as arguments.

int similar (line * *l1*, line * *l2*)

A heuristic algorithm that uses a point system to determine whether or not two points are part of the same line.

Generally, 20 heuristic points are enough to be the same line.

code/line.h File Reference

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "log.h"
```

Data Structures

38 struct line

Functions

39 int similar (line *, line *)
40 int linecmp (line *, line *)

code/log.c File Reference

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include "log.h"
```

Functions

41 void die (char *message)
42 void debug (char *message)
43 void error (char *message)
44 void fatal (char *message)
45 void finish ()
46 void log_file (int level, char *message)
47 void out (char *message)
48 void sys_error (int err, void(*func)(char *))
49 void warn (char *message)
50 void warning (char *message)

Function Documentation

void debug (char * *message*)

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 * *message*)

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 * *message*)

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 * *message*)

An alias for die().

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 *level*, char * *message*)

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 * *message*)

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 * *message*)

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 * *message*)

An alias for the warn().

code/log.h File Reference

Enumerations

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

Functions

52 void die (char *message)
53 void debug (char *message)
54 void error (char *message)
55 void fatal (char *message)
56 void finish ()
57 void log_file (int level, char *message)
58 void out (char *message)
59 void sys_error (int err, void(*func)(char *))
60 void warn (char *message)
61 void warning (char *message)

Variables

62 FILE * logfile
63 static int errorcount
64 static int warncount

Enumeration Type Documentation

enum levels

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

Variable Documentation

int errorcount `[static]`

Count numbers of errors 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.

FILE* logfile

You guessed it -- 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.

int warncount `[static]`

Counts the number of warning issued by the program at runtime. It remains constant over the course of many calls and instances. Each warning EXCEPT a log file warning (if the file becomes unavailable) increments this counter.

code/main.c File Reference

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

Functions

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

Function Documentation

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

The main execution method, called to start the entire program. This type of framework allows the program to be very flexible in terms of multiple processes running simultaneously, the order of how things are done, etc.

code/main.h File Reference

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "log.h"
#include "timer.h"
```

Defines

66 #define maxsize 1000

Functions

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

Variables

68 char * infilename
69 char * outfilename
70 FILE * infile
71 FILE * outfile
72 int image1 [maxsize][maxsize]
73 int image2 [maxsize][maxsize]
74 int rows
75 int cols
76 int maxpixel
77 char format [3]

Define Documentation

#define maxsize 1000

The maximum dimensions of an image. Anything larger will be truncated. Therefore it is advisable to use a 640x480 webcam or to automatically scale images down. this number is also a security measure: otherwise the program may end up taking too much memory and/or processor power.

Variable Documentation

int cols

The number of columns in the image.

char format[3]

The format string of the image.

int 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 advisable to significantly increase this, even more so when increasing the dimensions provokes an squared increase in matrix size.

int image2[maxsize][maxsize]

Image data after edge detection is complete.

FILE* infile

The input image file.

char* infilename

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.

int maxpixel

The maximum pixel value.

FILE * outfile

The output file pointer.

char * outfilename

The output filename.

int rows

Some information about the image in question, such as how many rows and columns, as well as the maximum number of colors the image has. This is necessary information when it is printed as the header of the new image. The format string is necessary to hold information on the image type.

code/timer.c File Reference

```
#include "timer.h"
```

Functions

78 timestamp init_timer ()
79 timestamp start_timer (timestamp *timer)
80 timestamp stop_timer (timestamp *timer)
81 timestamp restart_timer (timestamp *timer)
82 long running_time (timestamp *timer)
83 long total_time (timestamp *timer)

Function Documentation

timestamp init_timer ()

Initialize the timer with default values. This is the best way, as it uses the defaults which ensure that errors will not happen.

timestamp restart_timer (timestamp * *timer*)

Restart a previously stopped timer and calculate the elapsed time when the timer was inactive, so that a total time and also a running time can be calculated with the same timer.

Definition at line 63 of file timer.c.

long running_time (timestamp * *timer*)

Calculate the total amount of time which the timer has been running (not when it was stopped).

timestamp start_timer (timestamp * *timer*)

“Start” the timer by setting the start_time variable to the current time. This will serve as a reference when the program needs an elapsed time later on.

timestamp stop_timer (timestamp * *timer*)

Stop the timer by recording the time of the command and placing this in the end_time variable. Also, set its status to stopped, so that other timer-based methods can execute accordingly.

long total_time (timestamp * *timer*)

Calculate the total amount of time since the timer was first started, regardless when it was stopped and/or started.

code/timer.h File Reference

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

Data Structures

84 struct timestamp

Enumerations

85 enum { TIMER_STARTED, TIMER_STOPPED, TIMER_RESTARTED, TIMER_NEW }

Functions

86 timestamp init_timer ()
87 timestamp start_timer (timestamp *)
88 timestamp stop_timer (timestamp *)
89 timestamp restart_timer (timestamp *)
90 long running_time (timestamp *)
91 long total_time (timestamp *)

Enumeration Type Documentation

enum timer_type

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

---

code/xml_parse.c File Reference

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>
#include "xml_parse.h"
#include "log.h"
```

Functions

92 int is_tag (char *test)
93 int is_value (char *test)
94 char * next_tag ()
95 char * next_value (char *tag)
96 int main (int argc, char *argv[])
97 char * parsexmlfile (char *filename)

98   int strpos (char *str, char ch)

Function Documentation

int is_tag (char * *test*)

A method to test whether or not a string is a tag. This returns one of a set of values, describing what sort of tag the string is (or that the string is not a tag at all).

int is_value (char * *test*)

The method to determine whether or not a given string is a value (often this is enclosed within an opening and a closing tag). If yes, then the value type (either a set of tags or a string) is returned. If not, then a value which indicates as much is returns.

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

A temporary function which will disappear as soon as the rest of the program has been thoroughly tested.

char* next_tag ()

The function to find the next XML tag in the string, after a certain point. It returns a string of the tag and its value.

char* next_value (char * *tag*)

The method which returns the text in the content pointer up to the next closing tag for the opener specified.

char* parsexmlfile (char * *filename*)

The main method to parse an XML file (at least one of the complexity level used to store hand information). This calls all of the appropriate tag find mechanisms and parses the file into a computer-readable format.

The order of tags after this is as follows: 1) orientation -- either "forward" or "sideways" 2) thumb-pinky -- positions of these fingers. choices are documented in hands/spec.xml. 3) motion -- to be implemented later, for letters like 'j' and 'z'

int strpos (char * *str*, char *ch*)

Find the position in a string of a character. This is functionality which has annoyed me by being absent from built-in C libraries.

code/xml_parse.h File Reference

Defines

99   #define maxlength 5000

Enumerations

100 enum tags { TAG_LETTER, TAG_OPEN, TAG_CLOSE, TAG_NONE }
101 enum values { VALUE_TEXT, VALUE_TAGS, VALUE_NONE }

Functions

102 int is_tag (char *test)
103 int is_value (char *test)
104 char * next_tag ()
105 char * next_value (char *tag)
106 char * parsexmlfile (char *filename)
107 void set_letter (char *tag, char *value)
108 int strpos (char *str, char ch)

Variables

109 FILE * infile
110 char * content

Define Documentation

#define maxlength 5000

The maximum length of certain strings, set high enough to be safe from buffer overflows but low enough to not waste memory.

Enumeration Type Documentation

enum tags

A set of values which represent the possible types of tags. This is handy when trying to classify a string as either a tag or a value so that it can be appropriately parsed. The separate values are intended to distinguish general tags which compose the majority of XML data read in, such as <thumb> or <motion>, with unusual tags like <letter>.

Enumerator:

TAG_LETTER
TAG_OPEN
TAG_CLOSE
TAG_NONE

enum values

Enumerator:

VALUE_TEXT
VALUE_TAGS
VALUE_NONE

Variable Documentation

char* content

A variable which describes whether or not the runtime log file is accessible. If at any point the program cannot access it, then this flag is set to true to prevent any further errors. The contents of the XML file, slowly dropped as it is parsed. Each time the next tag is found, this pointer is moved just slightly ahead, until the length of this string is less than or equal to two (the point after which no legal tag is possible).

FILE* infile

The input file which contains data about various hand positions in XML format. This is the file that will be parsed by this program. If desired, the program will parse multiple XML files serially (this will be specified in the parsing command).