Systems Lab Research Project Code Analysis

The Applications of Image Processing
Techniques to Sign Language Recognition

Byron Hood
Version 0.1
October 31, 2007

Data Structure Documentation

`line` Struct Reference

Data Fields

1 int **start_x**
2 int **start_y**
3 int **end_x**
4 int **end_y**
5 double **slope**

Detailed Description

The structure in which the program stores information about a line as it goes about detecting lines in an image which has previously gone through edge-detection. The fields are optimized for later usability by the line-finding AI.

Field Documentation

int line::start_x

int line::start_y

int line::end_x

int line::end_y

double line::slope

`ts` Struct Reference

Data Fields

6 struct timeval **start**
7 struct timeval **end**
8 struct timeval **inactive**
9 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 ts::start [read]

The starting time of the timer.

struct timeval ts::end [read]

The stopping time of the timer.

struct timeval ts::inactive [read]

The amount of time a timer has been inactive.

int ts::status

The timer's status, such as "started" or "stopped"

Syslab Tech Project 2007 File Documentation

project/capture.c File Reference

```
#include <asm/types.h>
#include <errno.h>
#include <fcntl.h>
#include <linux/videodev.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include "timer.c"
```

Functions

10 int **save_pnm** (char ***buf**, int x, int y, int depth)
11 int **main** (int argc, char *argv[])

Variables

12 struct video_picture **grab_pic**
13 struct video_capability **grab_cap**
14 struct video_channel **grab_vid**
15 struct video_mmap **grab_buf**
16 struct video_mbuf **grab_vm**
17 unsigned int **grab_fd**
18 unsigned int **grab_size**
19 unsigned int **frame**
20 unsigned int **done** = 1
21 unsigned int **framecount** = 0
22 unsigned char * **grab_data**
23 unsigned char * **buf**

Function Documentation

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

The function which grabs frames and instructs the second method to save them. For the sake of simplicity, this has not received a header file yet (but will once I perfect it).

int save_pnm (char * *buf*, int *x*, int *y*, int *depth*)

A very simple function to save in the PNM standard formats, *.ppm and *.pgm (the first is color whereas the second is grayscale).

---

project/crop.c File Reference

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

Defines

24 #define **maxsize** 1000

Functions

25 int * **crop_rows** (int ***image**, int **rows**, int **cols**)
26 int * **crop_cols** (int ***image**, int **rows**, int **cols**)
27 int * **_crop_cols** (int **image**[maxsize][maxsize], int **rows**, int **cols**)
28 int **is_in** (int *arr, int len, int val)
29 void **seg** (int sig)
30 int **main** (int argc, char *argv[])

Variables

31 int **newrows**
32 int **newcols**

Define Documentation

#define maxsize 1000

Definition at line 22 of file crop.c.

Function Documentation

int * _crop_cols (int *image*[maxsize][maxsize], int *rows*, int *cols*)

Crop all of the unused columns in the image (those that are entirely black).

int * crop_cols (int * *image*, int *rows*, int *cols*)

An incomplete optimization of the above cropping mechanism.

int * crop_rows (int * *image*, int *rows*, int *cols*)

Crop all of the rows in an image (where the entire row is balck).

int is_in (int * *arr*, int *len*, int *val*)

If a value is in the first *len* elements of array *arr*.

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

Execute the program to crop an image down to size for optimization purposes.

---

project/edge_detect.c File Reference

```
#include <stdio.h>
#include <math.h>
#include <string.h>
```

Defines

33 #define **maxsize** 1000

Functions

34 void **edge_detect** (int *, int *, int, int)
35 void **__edge_detect__** (int **image1**[maxsize][maxsize], int **image2**[maxsize][maxsize], int **rows**, int **cols**)
36 void **seg** (int)

Variables

37 FILE * **thefile**

Define Documentation

#define maxsize 1000

The maximum dimensions of an image. Anything larger will be truncated. Therefore it is adivsable to use a 640x480 webcam.

Function Documentation

void __edge_detect__ (int *image1*[maxsize][maxsize], int *image2*[maxsize][maxsize], int *rows*, int *cols*)

An old, deprecated edge-detection function in there for backwards compatibility.

void edge_detect (int * *image1*, int * *image2*, int *rows*, int *cols*)

This runs the Robert's Cross edge detection algorithm on the image data in the first argument and stores the result in argument two. Robert's cross looks something like this:

value = sqrt((left - right)^2 + (top - bottom)^2)

Where left, right, top, and bottom refer to pixels to the left, to the right, above, and below the current pixel, respectively.

This method is highly effective and balances good edge detection with eliminating most of the unnecessary noise.

---

project/find_lines.c File Reference

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

Data Structures

38 struct **line**

Defines

39 #define **maxsize** 1000
40 #define **HIGHLIGHT_THRESHOLD** 20
41 #define **round**(x) ((x-floor(x) < 0.5) ? (int)floor(x) : (int)ceil(x))

Enumerations

42 enum **boundary** { **OUT_BOUNDS_ABOVE**, **OUT_BOUNDS_BELOW**, **IN_BOUNDS** }

Functions

43 int **check_bounds** (int, int, int)
44 void **branch_out** (int, int, int, **line** *)
45 int **detect_lines** (int rate, **line** lines[10000])
46 int **is_highlighted** (int)
47 void **print_lines** (int lc, **line** lines[10000])
48 int **main** (int argc, char *argv[])

Variables

49 enum **boundary b**
50 char * **infilename**
51 FILE * **infile**
52 int **image** [maxsize][maxsize]
53 int **rows**
54 int **cols**
55 int **maxpixel**
56 char **format** [3]

Define Documentation

#define HIGHLIGHT_THRESHOLD 20

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 ;-)

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 *row*, int *col*, int *rate*, line * *result*)

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 degrees). Best set to somewhere between 5 and 10. Returns: A very specialized array which contains the coordinates of the starting and ending points of the **line**. The array is four integers long.

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

Prototypes so that I can keep track of everything.

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 (int *rate*, line *lines*[10000])

The actual line detector. The general method is to iterate through angles at a given rate (best is 5-10 degrees) and look through the image for lines at that particular angle.

int is_highlighted (int *item*)

Whether or not a given point is “on” or “off”

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

Execute the line-finding program.

void print_lines (int *lc*, line *lines*[10000])

Print out any lines found.

---

project/log.c File Reference

Enumerations

57 enum { **LEVEL_DEBUG**, **LEVEL_OUTPUT**, **LEVEL_WARNING**, **LEVEL_ERROR**, **LEVEL_FATAL** }

Functions

58 void **die** (char *message)
59 void **debug** (char *message)
60 void **error** (char *message)
61 void **fatal** (char *message)
62 void **finish** ()
63 void **log_file** (int level, char *message)
64 void **out** (char *message)
65 void **sys_error** (int err, void(*func)(char *))

66 void **warn** (char *message)
67 void **warning** (char *message)

Variables

68 FILE * **logfile**
69 int **fileoff** = 0
70 static int **errorcount** = 0
71 static int **warncount** = 0

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

Definition at line 56 of file log.h.

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()** 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 *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 **warn()**, in case I attempt to call an alternate warning function.

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

---

project/main.c File Reference

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

Defines

72 #define **maxsize** 1000

Functions

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

Variables

74 char * **infilename**
75 char * **outfilename**
76 FILE * **infile**
77 FILE * **outfile**
78 int **image1** [maxsize][maxsize]
79 int **image2** [maxsize][maxsize]
80 int **rows**
81 int **cols**
82 int **maxpixel**
83 char **format** [3]

Define Documentation

#define maxsize 1000

The maximum dimensions of an image. Anything larger will be truncated. Therefore it is adivsable 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.

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.

---

project/timer.c File Reference

```
#include <stdio.h>
#include <time.h>
```

Data Structures

84 struct **ts**

Enumerations

85 enum { **TIMER_STARTED**, **TIMER_STOPPED**, **TIMER_RESTARTED**, **TIMER_NEW** }

Functions

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

Enumeration Type Documentation

anonymous enum

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

ts 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.

ts restart_timer (ts * *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.

long running_time (ts * *timer*)

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

ts start_timer (ts * *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.

ts stop_timer (ts * *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 (ts * *timer*)

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

project/xml_parse.c File Reference

Defines

92 #define **maxlength** 5000

Enumerations

93 enum **tags** { **TAG_LETTER**, **TAG_OPEN**, **TAG_CLOSE**, **TAG_NONE** }
94 enum **values** { **VALUE_TEXT**, **VALUE_TAGS**, **VALUE_NONE** }

Functions

95 int **is_tag** (char *test)
96 int **is_value** (char *test)
97 char * **next_tag** ()
98 char * **next_value** (char *tag)
99 char * **parsexmlfile** (char *filename)
100 void **set_letter** (char *tag, char *value)
101 int **strpos** (char *str, char ch)

Variables

102 FILE * **infile**
103 char * **content**

Define Documentation

#define maxlength 5000

The maximum length of certain strings, set high enoug hto 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

A set of values which represent possible types of tag values. They come in useful when classifying values between <hand> and <motion> type tags.

Enumerator:

VALUE_TEXT
VALUE_TAGS

VALUE_NONE

Function Documentation

int is_tag (char * *test*)

The function which tests a string for the `<' and `>' characters.

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). For those values, see the enum above.

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.

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 appropiate 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'

void set_letter (char * *tag*, char * *value*)

Propose a letter to the AI.

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.
