FastTrack or "Where's Waldo?"

A C library to find visual landmarks in real-time

João P. Hespanha

August 6, 2007*†

Abstract

This document provides a guide to use the `FastTrack` library. The core of this library is a couple of functions that search for visual landmarks in gray-level images. These visual landmarks consist of "Square-in-Square" (SiS) shapes with specific geometric ratios between the sizes of the squares.

The `FastTrack` functions were constructed to pick out SiS landmarks, regardless of lighting conditions and viewing direction. They are also very fast, taking only a few milliseconds to process a full image. This makes `FastTrack` attractive for robot navigation applications that need to process images at frame rate.

The `FastTrack` library also provides a few auxiliary functions to aid in the development and debug of real-time image processing in the X-Windows environment.

Examples are provided to illustrate the use of the library.

Figure 1: Square-in-square landmarks. Where's Waldo?

*Major revision from a version dated October 30, 2004.
†Minor revision from a version dated September 5, 2006

Contents

1 Introduction

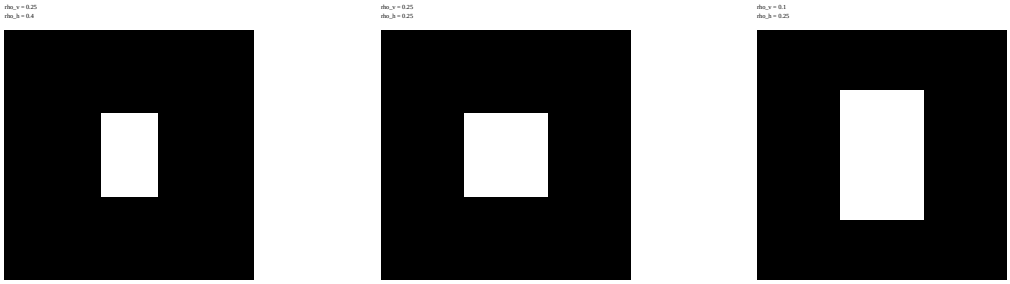

Figure 2: Square-in-square landmarks with different cross-rations

This library provides functions to search for visual landmarks in gray-level images. The visual landmarks considered here consist of "square-in-square" (SiS) shapes like the ones in Figure 2. Each SiS is characterized by a vertical and a horizontal cross ratio as shown in figure 3. These cross ratios have the crucial property that they remain invariant under perspective (and projective) transformations and therefore specific SiSs can be identified *regardless of the viewing direction*.

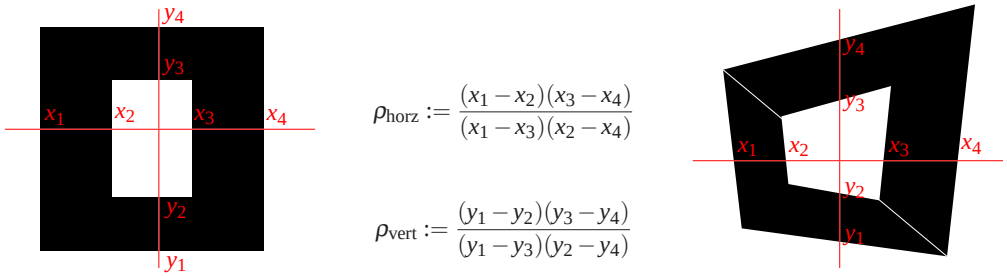

Figure 3: Horizontal and vertical cross-rations. Cross ratios are invariant with respect to projective transformations and therefore remain the same under different viewing directions of the SiS.

1.1 The good news...

View direction invariance As mentioned before, cross ratios are invariant under projective transformations and therefore specific SiSs can be identified from their cross-rations regardless of the viewing direction.

Lighting invariance The SiS detection is based on edge detection in the sense that the shape of the SiS is completely determined by the edges that define the SiS. Therefore SiS detection is very robust with respect to changes in lighting condition (including brightness, contrast, and color balance). As long as the edges that define the SiS can be detected, the SiS can be found by `FastTrack` .

Clutter Even in very cluttered images, it is very unlikely to find edges that

1. match the black-white-black pattern of an SiS both in the vertical and horizontal direction; and
2. simultaneously match the vertical and horizontal cross ratios of the SiS of interest.

This makes it very unlikely to get "false positives" in the detection of SiSs.

Speed The `FastTrack` functions that find SiSs have been highly optimized for speed, and even in fairly slow processors can find SiSs at frame-rate.

1.2 The bad news...

Tilted SiSs Although cross ratios are invariant under any projective transformation (and therefore retain their values under any rotation and translation of the target), the `FastTrack` function that looks for SiSs scans the image vertically and horizontally to look for cross rations. In the event that there is no horizontal scan line that intersects the four "vertical" SiS edges — or alternatively, if there is no vertical scan line that intersects the four "horizontal" SiS edges — then the SiS will not be detected. Figure 4 illustrates this difficulty.

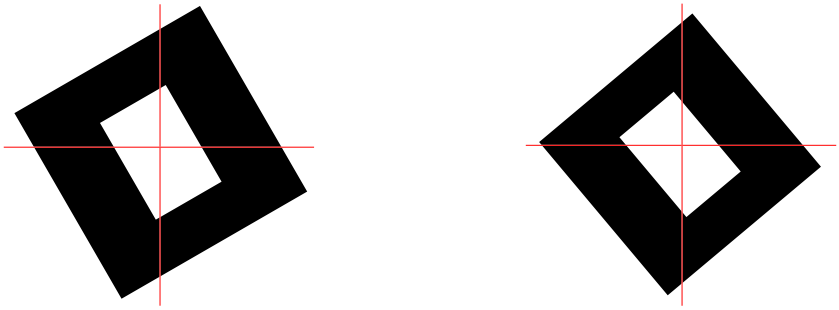

Figure 4: The left-hand side SiS can still be detected since (i) there exists an horizontal scan line that intersects the four parallel "vertical" edges and (ii) there exists a vertical scan line that intersects the four parallel "horizontal" edges. The right-hand side SiS will *not* be detected because there is no vertical scan line that intersects the four "horizontal" edges.

Blur Image blur due to poor focus or camera/SiS motion will make the edges hard to detect and SiS detection may fail. To mitigate this problem one can do the following:

- Keep the camera on focus. Small apertures enlarge the depth of field and keep better focus.
- Use high shutter speeds. A fast shutter will freeze motion and avoid blur.
- Avoid interlaced images. Interlaced images capture even and odd lines at different time instants separated by around 17ms (for a 60Hz frame rate). For horizontally moving objects, this often destroys vertical edges.

Searching for multiple SiSs in the same image The current `FastTrack` SiS detection functions seek for a specific SiS, with given vertical and horizontal cross-rations. If one wants to find multiple SiSs in the same image, one much call the SiS detection function multiples times. This is inefficient since at least some of the image processing could be re-used. In the future, this additional efficiency improvement will be implemented.

1.3 The rest of this document

Section 2 describes the basic types used by `FastTrack` to store images.

Section 3 describes the `FastTrack` image processing functions. The two key functions for SiS detection are `HVEdgeCross` and `VHEdgeCross`. However, to better understand the parameters that these functions need, we also describe the functions used to perform convolution and edge detection.

The remaining sections describe several `FastTrack` auxiliary functions to aid in the development and debug of real-time image processing in the X-Windows environment:

- Section 4 describes functions to capture images using the Video4Linux2 device.
- Section 5 describes functions to load and save images to files using a wide variety of formats. These functions use the GDK library.
- Section 6 describes functions to display images in the X-Windows environment. These functions use the GTK library.
- Section 7 describes a Perl script that can be used to print SiS with appropriate cross-ratios.
- The Appendix lists `FastTrack` default parameters and presents a few examples that illustrate the use of the library.

2 Image types and conversion

`FastTrack` stores images in various C structures. The following types are defined in `FastTrack.h`.

1. `FT_4GrayImage_T`, `FT_8GrayImage_T`, `FT_16GrayImage_T` correspond to images stored as a linear array of 4/8/16bit gray-level integers.
2. `FT_8RLEGrayImage_T` correspond to images stored as a Run Length Encoding (RLE) array of 8bit gray levels;

Most computations are done for images stored as `FT_16GrayImage_T` to avoid integer overflow. This type is defined as follows

```
typedef struct {
  unsigned short width;          // image width in pixels
  unsigned short height;         // image height in pixels
  unsigned short type;           // ``magic'' byte which indicates the
                                 // type of structure (\texttt{FT\_16BITGRAY}
                                 // for \texttt{FT\_16GrayImage\_T})
  int            pixelslength;   // size in bytes of the following array
  signed short pixels[FT_MAX_IMAGEWIDTH*FT_MAX_IMAGEHEIGHT]; // image data
} FT_16GrayImage_T;
```

Any `FastTrack` function that creates an `FT_GrayImage_T` fills out all these fields automatically.

The remaining types `FT_4GrayImage_T`, `FT_8GrayImage_T`, `FT_8RLEGrayImage_T` only differ by the type and size of the field `pixels`. Their values for the `type` field are `FT_4BITGRAY`, `FT_8BITGRAY`, and `FT_8BITRLEGRAY`. These types are mostly useful to store and transmit images because they require a smaller number of bytes (cf. conversion functions below).

`ConvertAnyto8()`

```
void ConvertAnyto8(FT_16GrayImage_T *Input,FT_8GrayImage_T *Output);
void Convert16to8(FT_16GrayImage_T *Input,FT_8GrayImage_T *Output);
void Convert4to8(FT_4GrayImage_T *Input,FT_8GrayImage_T *Output);
void Convert8RLEto8(FT_8RLEGrayImage_T *Input,FT_8GrayImage_T *Output);
```

These functions convert the image referenced by `Input` into a `FT_8GrayImage_T` object and stores it in the object referenced by `Output`. The original image can be stored in any of the following formats: `FT_4GrayImage_T`, `FT_16GrayImage_T`, or `FT_8RLEGrayImage_T`. One can always use `ConvertAnyto8` to do any of these conversions. This function actually checks the type of the `Input` image an then calls the appropriate conversation function.

`Convert16to4()`

```
void Convert16to4(FT_16GrayImage_T *Input,FT_4GrayImage_T *Output);
```

These functions convert the image referenced by `Input` into a `FT_4GrayImage_T` object and stores it in the object referenced by `Output`. The original image is assumed to be stored in the `FT_16GrayImage_T` format. This amounts to a very simple form of compression.

`Convert16to8RLE()`

```
void Convert16to8RLE(FT_16GrayImage_T *Input,FT_8RLEGrayImage_T *Output,int threshold);
```

These functions convert the image referenced by `Input` into a `FT_8RLEGrayImage_T` object and stores it in the object referenced by `Output`. The original image is assumed to be stored in the `FT_16GrayImage_T` format. Gray values that differ by less than `threshold` are assumed equal for RLE. In general, this achieves significant compression when `threshold` is high.

3 Image processing functions

3.1 Convolutions

`HorizConv(), VertConv()`

```
void HorizConv(
   FT_16GrayImage_T *Input,   // original image                          (input)
   FT_16GrayImage_T *Output,  // output image                            (output)
   int hlength,               // horizontal length of (half) template   (input)
   int vlength,               // vertical length of template            (input)
   int FirstColumn,           // region of interest                      (input)
   int FirstRow,
   int LastColumn,
   int LastRow);

void VertConv(
   FT_16GrayImage_T *Input,   // original image                          (input)
   FT_16GrayImage_T *Output,  // output image                            (output)
   int vlength,               // vertical length of (half) template     (input)
   int hlength,               // horizontal length of template          (input)
   int FirstColumn,           // region of interest                      (input)
   int FirstRow,
   int LastColumn,
   int LastRow);
```

These functions preform a convolution on a specified region of the image referenced by `Input` and stores the result in the object referenced by `Output`. The convolution is only performed on the region specified by `FirstColumn`, `FirstRow`, `LastColumn`, `LastRow`. Rows and columns are numbered starting at zero. If possible, this region will be "enlarged" to make sure that boundary effects do not introduce errors in the rows and columns specified by these variables.

With `HorizConv()` the image is convolved with the following horizontal step template:

$$\underbrace{\left.\begin{array}{cccccccc} -1 & -1 & \cdots & -1 & +1 & +1 & \cdots & +1 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & \ldots & \vdots \\ -1 & -1 & \cdots & -1 & +1 & +1 & \cdots & +1 \end{array}\right\}}_{\texttt{hlength}\ \textit{elements}\qquad\qquad\texttt{hlength}\ \textit{elements}} \texttt{vlength}\ \textit{elements}$$

The resulting image essentially contains a filtered version of the *derivative of the gray-levels along each row* of the original image.

With `VertConv()` the image is convolved with the following vertical step template:

$$\left.\begin{array}{ccc} -1 & \cdots & -1 \\ \vdots & & \vdots \\ -1 & \cdots & -1 \end{array}\right\} \texttt{vlength}\ \textit{elements}$$

$$\underbrace{\left.\begin{array}{ccc} +1 & \cdots & +1 \\ \vdots & & \vdots \\ +1 & \cdots & +1 \end{array}\right\}}_{\texttt{hlength}\ \textit{elements}} \texttt{vlength}\ \textit{elements}$$

The resulting image essentially contains a filtered version of the *derivative of the gray-levels along each column* of the original image.

Parameter selection Large values for `vlength` and `hlength` will generally reduce the effect of noise but they may make the algorithm "miss" small edges. Also, a large value for `vlength` in `HorizConv()` will result in poor detection of edges that are not quite vertical or that are only a few pixels tall. Analogous problems arise for large values of `hlength` in `VertConv()`.

Computational complexity The computational complexity of `HorizConv()` and `VertConv()` is proportional to the size of the region:

$$O\Big((\texttt{LastColumn} - \texttt{FirstColumn}) \times (\texttt{LastRow} - \texttt{FirstRow})\Big).$$

In particular, the complexity does not increase significantly as one increases the size of the template defined by `vlength` or `hlength`.

Future work `HorizConv()` could probably be significantly optimized for `vlength`=1, and `VertConv()` for `hlength`=1.

3.2 Edge detection

`HEdgeCross(), VEdgeCross()`

```
int HEdgeCross(
   FT_16GrayImage_T *Grad,     // horizontal gradient                  (input)
   FT_16GrayImage_T *Grad2,    // horizontal "2nd" gradient            (input)
   FT_16GrayImage_T *Output,   // output image                         (output)
   int FirstColumn,            // region of interest                   (input)
   int FirstRow,
   int LastColumn,
   int LastRow,
   int EdgeThreshold,          // edge detection threhold              (input)
   float Cross,               // horizontal cross-ratio               (input)
   float CrossThreshold,      // horizontal cross-ratio threshold     (input)
   int   *bestrow,             // row of vertical edges                (output)
   float *bestedges            // vertical edges (array of 4)          (output)
   );

int VEdgeCross(
   FT_16GrayImage_T *Grad,     // vertical gradient                    (input)
   FT_16GrayImage_T *Grad2,    // vertical "2nd" gradient              (input)
   FT_16GrayImage_T *Output,   // output image                         (output)
   int FirstColumn,            // region of interest
   int FirstRow,
   int LastColumn,
   int LastRow,
   int EdgeThreshold,          // edge detection threhold              (input)
   float Cross,               // vertical cross-ratio                 (input)
   float CrossThreshold,      // vertical cross-ratio threshold       (input)
   int   *bestcolumn,          // column of vertical edges             (output)
   float *bestedges            // horizontal edges (array of 4)        (output)
   );
```

`HEdgeCross()` searches for *vertical edges* matching certain geometric conditions on a specified region of the image whose first horizontal derivative is referenced by `Grad` and the second horizontal derivative by `Grad2`.

`VEdgeCross()` searches for *horizontal edges* matching certain geometric conditions on a specified region of the image whose first vertical derivative is referenced by `Grad` and the second vertical derivative by `Grad2`.

The search is only performed on the region specified by `FirstColumn`, `FirstRow`, `LastColumn`, `LastRow`. Rows and columns are numbered starting at zero.

The function reports four consecutive edges that satisfy the following conditions:

1. The gradient `*Grad` exceeds `EdgeThreshold` at the four edges.
2. The four edges are of the type "light-to-dark," "dark-to-light," "light-to-dark," "dark-to-light," i.e., something of the type  for `HEdgeCross()` or  for `VEdgeCross()`;
3. Denoting by x_1, x_2, x_3, x_4 the columns of the edges, we must have

$$|\rho - \texttt{Cross}| \leq \texttt{CrossThreshold},$$

where ρ is the following projective-invariant cross-ratios:

$$\rho := \frac{(x_1 - x_2)(x_3 - x_4)}{(x_1 - x_3)(x_2 - x_4)}.$$

Among all the edges that satisfy these conditions, the function returns the edge quadruple that minimizes

$$|\rho - \texttt{Cross}|$$

For `HEdgeCross()`, the *row* where these edges were found is reported in `*bestrow` and the *columns* of the four edges are reported in `*bestedges`.

For `VEdgeCross()`, the *column* where these edges were found is reported in `*bestcolumn` and the *rows* of the four edges are reported in `*bestedges`.

The values returned in `*bestedges` corresponds to interpolated zeros of the seconds derivative `*Grad2`. In general these are not integers, providing sub-pixel accuracy.

Computational complexity The computational complexity of `HEdgeCross()` and `VEdgeCross()` is proportional to the size of the region:

$$O\Big((\texttt{LastColumn} - \texttt{FirstColumn}) \times (\texttt{LastRow} - \texttt{FirstRow})\Big).$$

Debug options The `*Output` image is only used for debug purposes. Its size and type is adjusted to match that of the input `*Grad` image. All edges that satisfy 1–3 above are marked in this image.

When `#define OUTPUT_EDGES` is set in `FastTrack.c`, all edges are marked in this image. Different gray-levels are used for positive edges, negative edges, edges with incorrect cross-ratio, and edges with correct cross ratio. Only the latter are marked when `OUTPUT_EDGES` is undefined.

Usage Typically, the image inputs to `HEdgeCross()` are computed using `HorizConv()` and the image inputs to `VEdgeCross()` are computed using `VertConv()`, as shown in the following example.

```
FT_16GrayImage_T Image,HGrad,HGrad2;
int row;
float vedges[4];

// ...  Assign value to Image ...

HorizConv(&Image,&HGrad,
          TEMPLATE_LONG_SIZE,TEMPLATE_SHORT_SIZE,
          MINCOL,MINROW,MAXCOL,MAXROW);
HorizConv(&HGrad,&HGrad2,
          TEMPLATE_LONG_SIZE,TEMPLATE_SHORT_SIZE,
          MINCOL,MINROW,MAXCOL,MAXROW);
if (!HEdgeCross(&HGrad,&HGrad2,&VEdge2,
                MINCOL,MINROW,MAXCOL,MAXROW,
                EDGETHRESHOLD,
                CROSS,CROSSTHRESHOLD,
                &row,vedges)) {
  // ... process edges ...
} else {
  fprintf(stderr,"Cross ratio not found!\n");
}
```

```
FT_16GrayImage_T Image,VGrad,VGrad2;
int col;
float hedges[4];

// ...  Assign value to Image ...

VertConv(Image,&VGrad,
         TEMPLATE_LONG_SIZE,TEMPLATE_SHORT_SIZE,
         MINCOL,MINROW,MAXCOL,MAXROW);
VertConv(&VGrad,&VGrad2,
         TEMPLATE_LONG_SIZE,TEMPLATE_SHORT_SIZE,
         MINCOL,MINROW,MAXCOL,MAXROW);
if (!VEdgeCross(&VGrad,&VGrad2,&VEdge,
                MINCOL,MINROW,MAXCOL,MAXROW,
                EDGETHRESHOLD,
                CROSS,CROSSTHRESHOLD,
                &col,hedges)) {
  // ... process edges ...
} else {
  fprintf(stderr,"Cross ratio not found!\n");
}
```

3.3 SiS detection

`HVEdgeCross()`, `VHEdgeCross()`

```
int HVEdgeCross(
   FT_16GrayImage_T *Image,    // original image                       (input)
   FT_16GrayImage_T *Output,   // output image                         (output)
   int FirstColumn,            // region of interest                   (input)
   int FirstRow,
   int LastColumn,
   int LastRow,
   int LengthGradDir,          // length of (half) template in grad dir (input)
   int LengthAveDir,           // length of template in averaging dir   (input)
   int EdgeThreshold,          // edge detection threhold               (input)
   float HCross,               // horizontal cross-ratio                (input)
   float HCrossThreshold,      // horizontal cross-ratio threshold      (input)
   float VCross,               // vertical cross-ratio                  (input)
   float VCrossThreshold,      // vertical cross-ratio threshold        (input)
   float *bestvm,              // slopes of vertical edges (array of 4) (output)
   float *bestvb,              // origin of vertical edges (array of 4) (output)
   float *besthm,              // slopes of horiz. edges (array of 4)   (output)
   float *besthb               // origin of horiz. edges (array of 4)   (output)
   );

int VHEdgeCross(
   FT_16GrayImage_T *Image,    // original image                       (input)
   FT_16GrayImage_T *Output,   // output image                         (output)
   int FirstColumn,            // region of interest                   (input)
   int FirstRow,
   int LastColumn,
   int LastRow,
   int LengthGradDir,          // length of (half) template in grad dir (input)
   int LengthAveDir,               // length of template in averaging dir   (input)
   int EdgeThreshold,          // edge detection threhold               (input)
   float HCross,               // horizontal cross-ratio                (input)
   float HCrossThreshold,      // horizontal cross-ratio threshold      (input)
   float VCross,               // vertical cross-ratio                  (input)
   float VCrossThreshold,      // vertical cross-ratio threshold        (input)
   float *bestvm,              // slopes of vertical edges (array of 4) (output)
   float *bestvb,              // origin of vertical edges (array of 4) (output)
   float *besthm,              // slopes of horiz. edges (array of 4)   (output)
   float *besthb               // origin of horiz. edges (array of 4)   (output)
   );
```

Both these functions search SiSs matching appropriate cross-ratios on a specified region of the image referenced by `Input` and stores the result in the object referenced by `Output`. The search is only performed on the region specified by `FirstColumn`, `FirstRow`, `LastColumn`, `LastRow`. Rows and columns are numbered starting at zero. The two function only differ by the order in which computations are performed. The choice of one over the other may result in a smaller or larger computation time. This issue is further discussed in the discussion below on computational complexity.

The functions report the position of a SiS shape as in Figure 3. The four vertical edges must satisfy the conditions used by `HEdgeCross()` to search for edges with parameters `EdgeThreshold`, `HCross`, `HCrossThreshold`. The derivatives are computed using `HorizConv()` with `hlength=LengthGradDir` and `vlenght=LengthAveDir`.

The four horizontal edges must satisfy the conditions used by `VEdgeCross()` to search for edges with parameters `EdgeThreshold`, `VCross`, `VCrossThreshold`. The derivatives are computed using `VertConv()` with `vlength=LengthGradDir` and `hlenght=LengthAveDir`.

In case no appropriate SiS is found, the function returns a nonzero value. Otherwise, the function returns zero and reports the positions of the four horizontal and the four vertical lines that compose the square. These are returned in

1. `bestvm[4]` and `bestvb[4]` for the vertical edges (from left to right). The equation for the `i`th edge is

$$column = \texttt{bestvm[i]} * row + \texttt{bestvb[i]}, \qquad \texttt{i} \in \{0,1,2,3\}.$$

2. `besthm[4]` and `besthb[4]` for the horizontal edges (from top to bottom). The equation for the *i*th edge is

$$row = \texttt{besthm[i]} * column + \texttt{besthb[i]}, \qquad \texttt{i} \in \{0,1,2,3\}.$$

Computational complexity `HVEdgeCross()` calls `HorizConv()` twice for the whole region (to obtain 1st and 2nd horizontal derivatives). For each quadruple of consecutive vertical edges that have satisfy 1–3 in `HEdgeCross()`, `VertConv()` is called twice for a small window around the edges found (two obtain 1st and 2nd vertical derivatives) and edges are searched on these windows.

`VHEdgeCross()` calls `VertConv()` twice for the whole region (two obtain 1st and 2nd horizontal derivatives). For each quadruple of consecutive vertical edges that have satisfy 1–3 in `VEdgeCross()`, `HorizConv()` is called twice for a small window around the edges found (two obtain 1st and 2nd vertical derivatives) and edges are searched on these windows.

In general one should use `HVEdgeCross()` instead of `VHEdgeCross()` when the number of vertical edges is smaller than the number of horizontal edges, because this will require the analysis of a smaller number of the small windows. However, the difference in computation time may not be significant.

The overall computational complexity of both `HVEdgeCross()` and `VHEdgeCross()` is proportional to the size of the region:

$$O\Big((\texttt{LastColumn} - \texttt{FirstColumn}) \times (\texttt{LastRow} - \texttt{FirstRow})\Big).$$

Debug options The `*Output` image is only used for debug purposes. Its size and type is adjusted to match that of the input `*Grad` image.

When `#define OUTPUT_CORRECT_CR` is set in `FastTrack.c`, horizontal edges that satisfy 1–3 in `HEdgeCross()`/`VEdgeCross(` are marked in the image.

When `#define OUTPUT_EDGES` is set in `FastTrack.c`, all edges are marked in this image. Different gray-levels are used for positive edges, negative edges, edges with incorrect cross-ratio, and edges with correct cross ratio.

When `#define OUTPUT_EXTENDED_EDGES` is set in `FastTrack.c`, edges that satisfy 1–3 in `HEdgeCross()`/`VEdgeCross()` are extended vertically and horizontally and then marked in the image.

The content of the `*Output` image is not cleared so, in practice, the edges are marked on top of the previous image in `*Output`. Typically, `*Output` should either be cleared or the original image should copied into `*Output` before calling these functions .

Lines2Points()

```
void Lines2Points(
   float *vm,  // slopes of vertical edges       (array of 4) (input)
   float *vb,  // origin of vertical edges       (array of 4) (input)
   float *hm,  // slopes of horiz. edges         (array of 4) (input)
   float *hb,  // origin of horiz. edges         (array of 4) (input)
   float *px,  // x coordinates of intersections (array of 8) (output)
               // big square clock-wise from top-left,
               // followed by small square
   float *py   // y coordinates of intersections (array of 8) (output)
   );          // same order as px
```

This function converts the outputs `bestvm`, `bestvb`, `besthm`, `besthd` of `HVEdgeCross()` and `VHEdgeCross()` into the x and y coordinates of the 8 corners that define the SiS found. The *x* and *y* corner coordinates are returned in the 8-element arrays `px` and `py`, respectively. The first four elements of each array correspond to the outer square, clockwise starting at the top-left corner, and the last four elements of each array correspond to the inner square, also clockwise starting at the top-left corner.

Usage The following example illustrates the combined use of `HVEdgeCross()`, `VHEdgeCross()`, and `Lines2Points()`.

```
FT_16GrayImage_T Image,Edge;
FT_8GrayImage_T  Edge8;
float vm[4],vb[4],hm[5],hb[4];
float px[8],py[8];

// ...  Assign value to Image ...

memcpy(Edge.pixels,Image.pixels,
       Image.pixelslength);
if (!HVEdgeCross(&Image,&Edge,
                 MINCOL,MINROW,MAXCOL,MAXROW,
                 TEMPLATE_LONG_SIZE,
                 TEMPLATE_SHORT_SIZE,
                 EDGE_THRESHOLD,
                 HCROSS,HCROSSTHRESHOLD,
                 VCROSS,VCROSSTHRESHOLD,
                 vm,vb,hm,hb)) {
  Lines2Points(vm,vb,hm,hb,px,py);
  Convert16to8(&Edge,&Edge8);
  DisplayNewImage(&Edge8,"HVEdgeCross");
} else {
  fprintf(stderr,"Cross ratio not found!\n");
}
```

```
FT_16GrayImage_T Image,Edge;
FT_8GrayImage_T  Edge8;
float vm[4],vb[4],hm[5],hb[4];
float px[8],py[8];

// ...  Assign value to Image ...

memcpy(Edge.pixels,Image.pixels,
       Image.pixelslength);
if (!VHEdgeCross(&Image,&Edge,
                 MINCOL,MINROW,MAXCOL,MAXROW,
                 TEMPLATE_LONG_SIZE,
                 TEMPLATE_SHORT_SIZE,
                 EDGE_THRESHOLD,
                 HCROSS,HCROSSTHRESHOLD,
                 VCROSS,VCROSSTHRESHOLD,
                 vm,vb,hm,hb)) {
  Lines2Points(vm,vb,hm,hb,px,py);
  Convert16to8(&Edge,&Edge8);
  DisplayNewImage(&Edge8,"VHEdgeCross");
} else {
  fprintf(stderr,"Cross ratio not found!\n");
}
```

4 Image capture functions (Video4Linux2)

v4l2Init()

```
int v4l2Init(
  char* deviceName,           // device name, typically "/dev/video0" (input)
  int videoSource,            // index of the video source (0)        (input)
  v4l2_std_id videoStandard,  // video standard (V4L2_STD_NTSC_M)     (input)
  int pixelFormat,            // desired pixel format                 (input)
  enum v4l2_field fieldOrder, // desired field order                  (input)
  int width,                  // window capture width                 (input)
  int height,                 // window capture height                (input)
  int nBuffers);              // desired # of buffers for stream capture (input)
```

This command initializes a Video4Linux2 capture device and starts continuous (stream) capture. The function returns file handle `fd` to be used by the remaining image capture functions.

The relevant input parameters are as follows:

1. `deviceName`: device file name, typically `/dev/video0`.
2. `videoSource`: the channel of video to be captured (for card with multiple inputs), typically `0`.
3. `videoStandard`: the color standard of the video signal, typically `V4L2_STD_NTSC_M`.
4. `pixelFormat`: the format in which pixels are stored in memory, typically `V4L2_PIX_FMT_GREY` for grey-level images or `V4L2_PIX_FMT_BGR24` for color images.

   More options can be found at `http://v4l2spec.bytesex.org/spec-single/v4l2.html#RGB-FORMATS`, but not all are supported by all devices.
5. `fieldOrder`: which fields should be stored in memory for multi-field video signals, typically `V4L2_FIELD_INTERLACED` if one wants to capture both fields or `V4L2_FIELD_BOTTOM` or `V4L2_FIELD_TOP` if one only wants to capture one field.

   More options can be found at `http://v4l2spec.bytesex.org/spec-single/v4l2.html#V4L2-FIELD`, but not all are supported by all devices.

   Note that if one uses `V4L2_FIELD_BOTTOM` or `V4L2_FIELD_TOP`, the total number of lines available is half the one available to `V4L2_FIELD_INTERLACED`.
6. `width`, `height`: size of image to be captured, typically 640×480 for interlaced video and 320×240 for a single field.
7. `nBuffers`: number of buffers used for continuous capture, typically 4–8.

   The device fills buffer as new images arrives but will not overwrite a buffer until it is read by `v4l2GetImage()` or `v4l2GetSubImage()`. If not more free buffers are available, images will be dropped. A large number of buffers will make image drops less likely.

   There is usually a limit on the maximum number of buffers that a device can use.

v4l2SetParameters()

```
void v4l2SetParameters(
     int fd,          // device file handle for ioctl
     int brightness,  // desired brightness, i.e., the black level
     int contrast,    // desired brightness, i.e., luma gain
     int saturation,  // desired color saturation, i.e., chroma gain
     int hue);        // desired hue or color balance
                      // all parameters from 0..65535
```

This function sets the image capture parameters. Before returning all buffers are cleared. The device handle `fd` to be used is obtained from `v4l2Init()`. All parameters take values between 0 and 65535. 32767 is a typically value for all parameters.

v4l2GetImage()

```
int v4l2GetImage(
  int fd,                   // device file handle for ioctl            (input)
  FT_16GrayImage_T *Output, // Image buffer to be filled               (output)
  struct timeval *tstamp,   // time stamp with start-of-capture time   (output)
  int *sequence,            // frame sequence number                  (output)
  struct timeval *timeout,  // timeout for waiting for new frame (input/output)
                            // outputs time left before timeout
  int *mnptr,               // minimum grey level (0..255)            (output)
                            // not computed if NULL (saves time)
  int *mxptr,               // maximum grey level (0..255)            (output)
                            // not computed if NULL (which saves time)
  int lastPicFlag)          // when nonzero, gets the most recent     (input)
                                       image available
                            // when zero, gets the oldest image not yet read
```

This function waits for the next image to become available (if none is currently available) and stores it in the FT_16GrayImage_T object referenced by Output.

When the lastPicFlag is nonzero, and several images are available, then it returns the most recent one, discarding all others. Otherwise, it return the oldest images that has not yet been read and keeps the others for subsequent calls to v4l2Get(Sub)Image().

It takes as input the timeout value timeout that specifies the maximum amount of time that the function will wait for a new image. This time can be zero, which means that the function will not wait and only produce an image if one is already available. Upon return timeout indicates how much time was left before the timeout.

The function returns:

1. -1 if the return was caused a timeout (no image stored in Output)
2. -2 if the device was busy (no image stored in Output)
3. the number of the buffer from which the message was retrieved (Output contains one image).

The following parameters are also returned by reference:

1. tstamp time at which image capture started as returned by gettimeofday()
2. sequence the video sequence number (incremented by one for each frame). When frames are dropped sequence will show this by skipping values.
3. mnptr, mxptr minimum and maximum grey level in the image (normalized from 0–255.

   These pointers can be set to NULL to avoid this computation and speed-up the function.

v4l2GetSubImage()

```
int v4l2GetSubImage(
  int fd,                   // device file handle for ioctl            (input)
  FT_16GrayImage_T *Output, // Image buffer to be filled               (output)
  struct timeval *tstamp,   // time stamp with start-of-capture time   (output)
  int *sequence,            // frame sequence number                  (output)
  struct timeval *timeout,  // timeout for waiting for new frame (input/output)
                            // outputs time left before timeout
  int FirstColumn,          // region of interest                      (input)
  int FirstRow,
  int LastColumn,
  int LastRow,
  int *mnptr,               // minimum grey level (0..255)             (output)
                            // not computed if NULL (saves time)
  int *mxptr,               // maximum grey level (0..255)             (output)
                            // not computed if NULL (which saves time)
  int lastPicFlag)          // when nonzero, gets the most recent      (input)
                                        image available
                            // when zero, gets the oldest image not yet read
```

This function is similar to v4l2GetImage() except only the region of the image specified by FirstColumn, FirstRow, LastColumn, LastRow is copied to Output.

If one wants to capture the whole image it is preferable to use v4l2GetImage(), which will be faster. However, to capture a small subset of the image v4l2GetSubImage() will be faster.

v4l2End()

```
void v4l2End(int fd);       // device file handle for ioctl
```

This command stops the continuous (stream) capture and closes the Video4Linux2 capture device.

5 Image load & save functions

LoadImage()

```
void LoadImage(char *filename,FT_16GrayImage_T *GrayImage);
```

This function reads an image from a file named `filename` and stores it in the `FT_16GrayImage_T` object referenced by `GrayImage`. This object must have enough space for the image to fit. `LoadImage` simply checks if the image width and height to be loaded does not exceed `FT_MAX_IMAGEWIDTH` and `FT_MAX_IMAGEHEIGHT`, respectively.

This function uses `gdk_pixbuf_new_from_file` from the GDK [1] library and supports multiple file formats (PNG, XPM, JPEG, TIFF, PNM, RAS, BMP, GIF). Color images are converted to gray-levels using some unspecified algorithm.

SavePGM8

```
void SavePGM8(char *filename, FT_8GrayImage_T *GrayImage);
```

This function saves the image referenced by `GrayImage` into a file named `filename`. The image is saved using the PGM RAW format, using 8bits per pixel. The image must be stored as a `FT_8GrayImage_T` object.

6 Image display functions (X Windows)

6.1 GDK

Basic image display in X-Windows is implemented using the GTK [2] library. For these functions to operate, the functions `gtk_init()` and gdk_rgb_init() must be called at the beginning of `main()`. The function `gtk_main()` gives control to `GDK`. This function should be called at the end of `main()` because it does not return.

Usage

```
int main(int argc, char *argv[])
{
  //... variable definition ...

  gtk_init(&argc, &argv);
  gdk_rgb_init();

  //... remaining of main, including additional processing of argc and argv

  gtk_main();
}
```

6.2 Window creation and update

DisplayNewImage()

```
GtkWidget *DisplayNewImage(FT_8GrayImage_T *Gray8Image,char *windowname);
```

This function creates a new window and displays in it the image stored in the `FT_8GrayImage_T` object referenced by `Gray8Image`. This object is expected to remain in memory until the window is closed. Changes in the object will translate into changes in the image displayed when the window is redrawn. The window will have the title given in `windowname`. The function return a handle to the image to be used by `UpdateImageDisplay()`. Automatically calls `ProcessXEvents()`.

```
filename "../images/board2.dib", width = 320, height = 186
Horizontal-Vertical detection (HVEdgeCross)
( 75.61, 39.85)------------------------------( 95.16, 44.05)
       |      ( 79.94, 49.17)---( 86.73, 47.57)      |
       |      ( 79.05, 54.81)---( 85.51, 54.98)      |
( 70.25, 59.06)------------------------------( 90.04, 63.37)
Vertical-Horizontal detection (VHEdgeCross)
( 75.61, 39.85)------------------------------( 95.16, 44.05)
       |      ( 79.94, 49.17)---( 86.73, 47.57)      |
       |      ( 79.05, 54.81)---( 85.51, 54.98)      |
( 70.25, 59.06)------------------------------( 90.04, 63.37)
```

Table 1: Standard output generated by the demonstration program `file2FT.c`, described in Section A.2

`UpdateImageDisplay()`

```
void UpdateImageDisplay(GtkWidget *drawarea);
```

This function informs X Windows that the window should be redrawn. Typically called when the image being displayed was changed and one wants this to be reflected in the image currently displayed in the screen. Automatically calls `ProcessXEvents()`.

`ProcessXEvents()`

```
void ProcessXEvents();
```

This function gives X Windows the opportunity to redraw any window that needs to be updated.

7 Printing targets

`maketarget.pl`

```
maketarget.pl -rho_h "0.25" -rho_v "0.25" -o target
```

The Perl script `maketarget.pl` can be used to produce post-script files with targets with desired vertical and horizontal cross-rations. By default the postscript file is save as `target.ps` and both the vertical and horizontal cross-ratios are equal to 0.25. However, these can be changed with appropriate switches: `-rho_h`, `-rho_v`, and `-o`.

A Appendix

A.1 `FastTrack` default parameters

The following parameters are defined in `FastTrack.h` to allocate space for various buffers:

```
#define FT_MAX_IMAGEWIDTH  640      // Maximum image width in pixels
#define FT_MAX_IMAGEHEIGHT 480      // Maximum image height in pixels
#define FT_MAX_RLELENGHT   64000    // Maximum length for RLE images
```

A.2 `FastTrack` Example 1 – Image from file

The program `file2FT.c` in Table 2 uses the `FastTrack` library to find a square in an image contained in a file. This program would be compiled using the texttttMakefile in Table 3 and it should be executed at the command line using `file2FT images/board2.dib`. Upon execution, the content of Table 1 appears in the standard output and the three windows in Figure 5 are displayed.

```
#include "FastTrack.h"

#define TEMPLATE_LONG_SIZE  3   // in the direction of derivative
#define TEMPLATE_SHORT_SIZE 2   // for averaging

#define EDGE_THRESHOLD 100      // threshold for edge detection

#define CRATIO_H .25            // Cross-rations being searched for
#define CRATIO_V .25            // (horizontal and vertical)

#define CRATIO_THRESHOLD .02    // Threshold to reject the cross ratio

FT_16GrayImage_T Image,VHOutput,HVOutput;
FT_8GrayImage_T  Image8,VHOutput8,HVOutput8;

int main(int argc, char *argv[])
{
  float vm[4],vb[4],hm[5],hb[4],px[8],py[8];
  ////// initialize gtk //////////////////////////////////////////////
  gtk_init(&argc, &argv);
  gdk_rgb_init();
  ////// process argv to get filename ////////////////////////////////
  if (argc != 2) {
    fprintf(stderr,"%s : wrong number of arguments (1 expected, %d found)\n",argv[0],argc-1);
    exit(1); }
  ////// LoadImage() /////////////////////////////////////////////////
  LoadImage(argv[1],&Image);
  fprintf(stderr,"filename \"%s\", width = %d, height = %d\n",
          argv[1],Image.width,Image.height);
  ////// call HVEdgeCross() //////////////////////////////////////////
  memcpy(HVOutput.pixels,Image.pixels,Image.pixelslength); //edges on original image
  fprintf(stderr,"Horizontal-Vertical detection (HVEdgeCross)\n");
  if (!HVEdgeCross(&Image,&HVOutput,0,0,Image.width-1,Image.height-1,
                   TEMPLATE_LONG_SIZE,TEMPLATE_SHORT_SIZE,EDGE_THRESHOLD,
                   CRATIO_H,CRATIO_THRESHOLD,
                   CRATIO_V,CRATIO_THRESHOLD,
                   vm,vb,hm,hb)) {
    Lines2Points(vm,vb,hm,hb,px,py);
    gettimeofday(&t1,NULL);
    printf("(%6.2f,%6.2f)------------------------------(%6.2f,%6.2f)\n",px[0],py[0],px[1],py[1]);
    printf("       |      (%6.2f,%6.2f)---(%6.2f,%6.2f)      |\n",px[4],py[4],px[5],py[5]);
    printf("       |      (%6.2f,%6.2f)---(%6.2f,%6.2f)      |\n",px[7],py[7],px[6],py[6]);
    printf("(%6.2f,%6.2f)------------------------------(%6.2f,%6.2f)\n",px[3],py[3],px[2],py[2]);
  } else {
    fprintf(stderr,"Cross ratio not found!\n"); }
  ////// call VHEdgeCross() //////////////////////////////////////////
  memset(VHOutput.pixels,0,Image.pixelslength); // edges on black image
  fprintf(stderr,"Vertical-Horizontal detection (VHEdgeCross)\n");
  if (!VHEdgeCross(&Image,&VHOutput,0,0,Image.width-1,Image.height-1,
                   TEMPLATE_LONG_SIZE,TEMPLATE_SHORT_SIZE,EDGE_THRESHOLD,
                   .25,CRATIO_THRESHOLD,
                   .25,CRATIO_THRESHOLD,
                   vm,vb,hm,hb)) {
    Lines2Points(vm,vb,hm,hb,px,py);
    gettimeofday(&t1,NULL);
    printf("(%6.2f,%6.2f)------------------------------(%6.2f,%6.2f)\n",px[0],py[0],px[1],py[1]);
    printf("       |      (%6.2f,%6.2f)---(%6.2f,%6.2f)      |\n",px[4],py[4],px[5],py[5]);
    printf("       |      (%6.2f,%6.2f)---(%6.2f,%6.2f)      |\n",px[7],py[7],px[6],py[6]);
    printf("(%6.2f,%6.2f)------------------------------(%6.2f,%6.2f)\n",px[3],py[3],px[2],py[2]);
  } else {
    fprintf(stderr,"Cross ratio not found!\n"); }
  ////// Display all images //////////////////////////////////////////
  Convert16to8(&Image,&Image8);
  DisplayNewImage(&Image8,"Original Image");
  Convert16to8(&HVOutput,&HVOutput8);
  DisplayNewImage(&HVOutput8,"HVEdgeCross (full image)");
  Convert16to8(&VHOutput,&VHOutput8);
  DisplayNewImage(&VHOutput8,"VHEdgeCross (full image)");
  ////// gtk_main() loop /////////////////////////////////////////////
  gtk_main();
}
```

Table 2: Demonstration program `file2FT.c` described in Section A.2

Figure 5: Windows displayed by the demonstration program `file2FT.c` described in Section A.2

```
##############################################################################
# Linux with GTK+ version 1.2 & glib version 1.2
# Needs: 'yum install gtk+-devel'
#        'yum install gdk-pixbuf-devel'
##############################################################################
#CFLAGS = -DGDK12 -I/usr/include/atk-1.0 -I/usr/include/pango-1.0/ \
#         -I/usr/lib/glib/include -I/usr/include/glib-1.2/ \
#         -I/usr/include/gtk-1.2 -I/usr/lib/gtk/include/ \
#         -I/usr/include/gdk-pixbuf-1.0 -L/usr/X11R6/lib -O3
#LIBS = -lX11 -lgdk -lgtk -lgdk_pixbuf
##############################################################################

##############################################################################
# Linux with GTK+ version 2.0 & glib version 2.0
# Needs: 'yum install gtk2-devel'
##############################################################################
CFLAGS = -DGDK20 -I/usr/include/atk-1.0 -I/usr/include/pango-1.0/ \
         -I/usr/include/glib-2.0/ -I/usr/lib/glib-2.0/include \
         -I/usr/include/gtk-2.0 -I/usr/lib/gtk-2.0/include/ \
         -L/usr/X11R6/lib -O3
LIBS = -lX11 -lgdk-x11-2.0 -lgtk-x11-2.0 -lgdk_pixbuf-2.0
##############################################################################

file2FT: file2FT.c FastTrack.c FastTrack.h
        gcc $(CFLAGS) file2FT.c FastTrack.c -o file2FT $(LIBS)
```

Table 3: `Makefile` for the demonstration program `file2FT.c` described in Section A.2

A.3 `FastTrack` Example 2 – Image from camera

The following program `v4l2FT.c` uses the library to find a square in an image obtained from a Video4Linux2 capture device. The brightness and contrast are automatically adjusted. The program should be executed with the command line `v4l2FT`.

```
#include "FastTrack.h"

#define TEMPLATE_LONG_SIZE  3   // in the direction of derivative
#define TEMPLATE_SHORT_SIZE 2   // for averaging

#define EDGE_THRESHOLD 150
#define CRATIO_H .25
#define CRATIO_V .25
#define CRATIO_THRESHOLD .02

FT_16GrayImage_T SubImage,Image,HVOutput,VHOutput;
FT_8GrayImage_T  SubImage8,Image8,HVOutput8,VHOutput8;

int main(int argc, char *argv[])
{
  int width=320,height=240;
  float vm[4],vb[4],hm[4],hb[4];
  int vidfd,i,j,mn,mx,rc,sequence;
  struct timeval timeout,tstamp,t0;
  float dt
  gettimeofday(&t0,NULL);

  ////// initialize gtk ///////////////////////////////////////////
  gtk_init(&argc, &argv);
  gdk_rgb_init();

  // Initialize device and start capture
  vidfd=v4l2Init("/dev/video0",          // device
                 0,                      // source
                 V4L2_STD_NTSC_M,        // standard
                 V4L2_PIX_FMT_GREY,      // pixel format
                 V4L2_FIELD_BOTTOM,         // field order
                 width,                  // window capture width
                 height,                 // window capture height
                 4);                       // number of buffers

  ////// adjust brightness and contrast
  int contrast=32767/4, brightness=32767*2;
  for (i=0;i<100;i++) {
    v4l2SetParameters(vidfd,brightness,contrast,
                      32767,         // saturation
                      32767);        // hue

    timeout.tv_sec = 2;timeout.tv_usec = 0;
    rc=v4l2GetSubImage(vidfd,           // device file handle for ioctl
                    &SubImage,          // Image
                    &tstamp,         // time stamp with start-of-capture time
                    &sequence,       // frame sequence number
                    &timeout,        // maximum waiting time
                    width/4,height/4,3*width/4,3*height/4,
                    &mn,             // minimum grey level
                    &mx);            // maximum grey level

    if (mx>250)      brightness=brightness/1.05;
    else if (mx<240) brightness=brightness*1.02;
    else break;
    if (brightness>65535) {
      brightness=65535;
      contrast=contrast*1.02; }
  }

  printf("v4l2 Sub Image: min = %d, max = %d (brightness= %5d, %contrast = %5d)\n",
         mn,mx,brightness,contrast);
```

```
  ////// v4l2GetImage() ////////////////////////////////////////////////////
  for (i=0;i<10;i++) {
    timeout.tv_sec = 2;timeout.tv_usec = 0;
    rc=v4l2GetImage(vidfd,           // device file handle for ioctl
                    &Image,          // Image
                    &tstamp,         // time stamp with start-of-capture time
                    &sequence,       // frame sequence number
                    &timeout,        // maximum waiting time
                    NULL,NULL);
    if (rc<0) fprintf(stderr,"v4l2GetImage: rc = %d\n",rc);

    dt=1000*(tstamp.tv_sec-t0.tv_sec)+.001*(tstamp.tv_usec-t0.tv_usec);
    fprintf(stdout,"rc %d seq %5d tstamp = %6.2fms wait = %6.2fms now = %6.2fms\n",rc,sequence,dt);
    fflush (stdout);}

  ////// call HVEdgeCross() ////////////////////////////////////////
  fprintf(stderr,"Horizontal-Vertical detection (HVEdgeCross)\n");
  memcpy(HVOutput.pixels,Image.pixels,Image.pixelslength);
  if (!HVEdgeCross(&Image,&HVOutput,0,0,Image.width-1,Image.height-1,
                   TEMPLATE_LONG_SIZE,TEMPLATE_SHORT_SIZE,EDGE_THRESHOLD,
                   CRATIO_H,CRATIO_THRESHOLD,
                   CRATIO_V,CRATIO_THRESHOLD,
                   vm,vb,hm,hb)) {
  } else { fprintf(stderr,"Cross ratio not found!\n"); }

  ////// call VHEdgeCross() ////////////////////////////////////////
  fprintf(stderr,"Vertical-Horizontal detection (VHEdgeCross)\n");
  memset(VHOutput.pixels,0,Image.pixelslength);
  if (!VHEdgeCross(&Image,&VHOutput,0,0,Image.width-1,Image.height-1,
                   TEMPLATE_LONG_SIZE,TEMPLATE_SHORT_SIZE,EDGE_THRESHOLD,
                   CRATIO_H,CRATIO_THRESHOLD,
                   CRATIO_V,CRATIO_THRESHOLD,
                   vm,vb,hm,hb)) {
  } else { fprintf(stderr,"Cross ratio not found!\n"); }

  Convert16to8(&SubImage,&SubImage8);
  DisplayNewImage(&SubImage8,"Sub Image");
  Convert16to8(&Image,&Image8);
  SavePGM8("frame0.pgm",&Image8);
  DisplayNewImage(&Image8,"Original Image");
  Convert16to8(&HVOutput,&HVOutput8);
  DisplayNewImage(&HVOutput8,"HVEdgeCross (full image)");
  Convert16to8(&VHOutput,&VHOutput8);
  DisplayNewImage(&VHOutput8,"VHEdgeCross (full image)");

  ////// gtk_main() loop ///////////////////////////////////////////
  gtk_main();
}
```

A.4 Undocumented functions

EdgeVExtend(), EdgeHExtend()

```
void EdgeVExtend(
   FT_16GrayImage_T *Grad,     // horizontal gradient                  (input)
   FT_16GrayImage_T *Grad2,    // horizontal "2nd" gradient            (input)
   FT_16GrayImage_T *Output,   // output image                         (output)
   int FirstColumn,            // region of interest                   (input)
   int FirstRow,
   int LastColumn,
   int LastRow,
   int EdgeThreshold,          // edge detection threhold              (input)
   int   row,                  // row of vertical edge to extend       (input)
   float edge,                 // vertical edge (column) to extend     (input)
   float *m,                   // slope of vertical edge               (output)
   float *b,                   // origin of vertical edge              (output)
   int *minrow,                // row where edge starts                (output)
   int *maxrow,                // row where edge ends                  (output)
   int *mincolumn,             // column where edge starts             (output)
   int *maxcolumn              // column where edge ends               (output)
   );

void EdgeHExtend(
   FT_16GrayImage_T *Grad,     // vertical gradient                    (input)
   FT_16GrayImage_T *Grad2,    // vertical "2nd" gradient              (input)
   FT_16GrayImage_T *Output,   // output image                         (output)
   int FirstColumn,            // region of interest                   (input)
   int FirstRow,
   int LastColumn,
   int LastRow,
   int EdgeThreshold,          // edge detection threhold              (input)
   int   column,               // column of horizontal edge to extend  (input)
   float edge,                 // horizontal edge (row) to extend      (input)
   float *m,                   // slope of horizontal edge             (output)
   float *b,                   // origin of horizontal edge            (output)
   int *minrow,                // row where edge starts                (output)
   int *maxrow,                // row where edge ends                  (output)
   int *mincolumn,             // column where edge starts             (output)
   int *maxcolumn              // column where edge ends               (output)
   );
```

The function `EdgeVExtend()` expands vertically vertical edges previously found by `HEdgeCross()`.

Usage

```
FT_16GrayImage_T Image,HGrad,HGrad2,Edge;
int row;
int thisminrow,thismaxrow,thismincol,thismaxcol;
float vedges[4],vm[4],vb[4];

// ...  Assign value to Image ...

HorizConv(&Image,&HGrad,
          TEMPLATE_LONG_SIZE,TEMPLATE_SHORT_SIZE,
          MINCOL,MINROW,MAXCOL,MAXROW);
HorizConv(&HGrad,&HGrad2,
          TEMPLATE_LONG_SIZE,TEMPLATE_SHORT_SIZE,
          MINCOL,MINROW,MAXCOL,MAXROW);
if (!HEdgeCross(&HGrad,&HGrad2,&VEdge2,
                MINCOL,MINROW,MAXCOL,MAXROW,
                EDGETHRESHOLD,
                HCROSS,HCROSSTHRESHOLD,
                VCROSS,VCROSSTHRESHOLD,
                &row,vedges)) {
  EdgeVExtend(&HGrad,&HGrad2,&Edge,
              MINCOL,MINROW,MAXCOL,MAXROW,
              EDGE_THRESHOLD,
              row,vedges[0],vm,vb,
              &thisminrow,&thismaxrow,
              &thismincol,&thismaxcol);
  EdgeVExtend(&HGrad,&HGrad2,&Edge,mincol,
              MINROW,MAXCOL,MAXROW,
              EDGE_THRESHOLD,
              row,vedges[1],vm+1,vb+1,
              &thisminrow,&thismaxrow,
              &thismincol,&thismaxcol);
  EdgeVExtend(&HGrad,&HGrad2,&Edge,
              MINCOL,MINROW,MAXCOL,MAXROW,
              EDGE_THRESHOLD,
              row,vedges[2],vm+2,vb+2,
              &thisminrow,&thismaxrow,
              &thismincol,&thismaxcol);
  EdgeVExtend(&HGrad,&HGrad2,&Edge,
              MINCOL,MINROW,MAXCOL,MAXROW,
              EDGE_THRESHOLD,
              row,vedges[3],vm+3,vb+3,
              &thisminrow,&thismaxrow,
              &thismincol,&thismaxcol);
}
```

```
FT_16GrayImage_T Image,VGrad,VGrad2,Edge;
int col;
int thisminrow,thismaxrow,thismincol,thismaxcol;
float hedges[4],hm[4],hb[4];

// ...  Assign value to Image ...

VertConv(Image,&VGrad,
         TEMPLATE_LONG_SIZE,TEMPLATE_SHORT_SIZE,
         MINCOL,MINROW,MAXCOL,MAXROW);
VertConv(&VGrad,&VGrad2,
         TEMPLATE_LONG_SIZE,TEMPLATE_SHORT_SIZE,
         MINCOL,MINROW,MAXCOL,MAXROW);
if (!VEdgeCross(&VGrad,&VGrad2,&VEdge,
                MINCOL,MINROW,MAXCOL,MAXROW,
                EDGETHRESHOLD,
                HCROSS,HCROSSTHRESHOLD,
                VCROSS,VCROSSTHRESHOLD,
                &col,hedges)) {
  EdgeHExtend(&VGrad,&VGrad2,&Edge,
              MINCOL,MINROW,MAXCOL,MAXROW,
              EDGE_THRESHOLD,
              col,hedges[0],hm,hb,
              &thisminrow,&thismaxrow,
              &thismincol,&thismaxcol);
  EdgeHExtend(&VGrad,&VGrad2,&Edge,mincol,
              MINROW,MAXCOL,MAXROW,
              EDGE_THRESHOLD,
              col,hedges[1],hm+1,hb+1,
              &thisminrow,&thismaxrow,
              &thismincol,&thismaxcol);
  EdgeHExtend(&VGrad,&VGrad2,&Edge,
              MINCOL,MINROW,MAXCOL,MAXROW,
              EDGE_THRESHOLD,
              col,hedges[2],hm+2,hb+2,
              &thisminrow,&thismaxrow,
              &thismincol,&thismaxcol);
  EdgeHExtend(&VGrad,&VGrad2,&Edge,
              MINCOL,MINROW,MAXCOL,MAXROW,
              EDGE_THRESHOLD,
              col,hedges[3],hm+3,hb+3,
              &thisminrow,&thismaxrow,
              &thismincol,&thismaxcol);
}
```

References

[1] *GDK Reference Manual*. Available at `http://developer.gnome.org/doc/API/gdk/`.

[2] *GTK+ Reference Manual*. Available at `http://developer.gnome.org/doc/API/2.0/gtk/`.

Index