```
/*****************************************************************************************************/
/*                 L I B V I S I O N   v2.00                                                         */
/*                                                                                                   */
/* Philippe LECLERCQ                                                                                 */
/* -----------------                                                                                 */
/*                  November 29th 2000                                                               */
/*                                                                                                   */
/*                                                                          lecle-p@ee.uwa.edu.au */
/*****************************************************************************************************/

/*****************************************************************************************************/
/* Header file for the LibVision v2.00                                                               */
/* Declaration of constants, structures and user functions                                           */
/*****************************************************************************************************/

/****h* LibVision/LibVision
 *DESCRIPTION
 *
 * All the 'LibVision' components (#define, structures and functions) begin with VIS to avoid
 * confusion with any other modules.
 *
 * Only the values in the header files may need to be changed. The values in the program source
 * are based on the values in the header and should not be changed.
 *
 * * ************** *
 * * #define values *
 * * ************** *
 *
 * Library default informations
 * ----------------------------
 *   #define VIS_MSB         5   : Number of most significant bits used for the 3D tables
 *   #define VIS_NONE        0   : No colour class
 *   #define VIS_CONFLICT    255 : Colour-conflict class
 *   #define VIS_SAT_THRES   32  : Threshold for limit saturation
 * @ RGB Values are coded on 8 bits, but for a RGB 3D space, ((2^8)^3)=16MB...
 * @ VIS_MSB defines how many MSB should be used : as an exemple for VIS_MSB=5 -> ((2^5)^3)=32kB.
 * @ VIS_NONE and VIS_CONFLICT define the two defaults values for the no-colour class and the colour-conflict
 * class.
 * @ VIS_SAT_THRES defines the default saturation threshold to define colours within the RGB space avoiding
 * defining white shades as colours.
 *
 * Defaults for the algorithm
 * --------------------------
 *   #define VIS_ALGO_DEPTH 1                  : One pixel around
 *   #define VIS_ALGO_FAST  1                  : Use fast
 *   #define VIS_ALGO_STEP  2                  : Every 2nd pixel
 *   #define VIS_ALGO_LOOP  (VIS_ALGO_STEP-1) : defined internally (in LibVision.c)
 * @ These values define the way to process the image. Some function are optimised for certain parameters.
 * Look the VIS_Algo structure for more informations on these values.
 *
 * Image size information
 * ----------------------
 *   #define VIS_IMROW imagerows    : number of rows
 *   #define VIS_IMCOL imagecolumns : number of columns
 * @ These two values are used to avoid the ANSI C warning about variable-size variables definition.
 *
 * Maximum number of classes to be processed
 * -----------------------------------------
 *   #define VIS_MAX_CLASS 10 : No more than 10 different classes
 * @ This value is used to avoid the ANSI C warning about variable-size variables definition.
 * It defines the maximum number of colour classes to be processed by the library.
 *
 * LCD Display informations, used by VIS_ComputeBW
 * -----------------------------------------------
 *   #define VIS_LCD_DISPLAY_ROW  64
 *   #define VIS_LCD_DISPLAY_COL  128
 *   #define VIS_LCD_DISPLAY_SIZE ((VIS_LCD_DISPLAY_ROW*VIS_LCD_DISPLAY_COL)>>3)
 * @ These values are used to define the LCD display properties. The VIS_LCD_DISPLAY_SIZE defines the 1D
 * memory length of a complete B/W image for the LCD display.
 *
 * Algorithm information, ONLY depending on VIS_MSB in the .h, (declared in LibVision.c)
 * -------------------------------------------------------------------------------------
 *   #define VIS_LSB      (8-VIS_MSB)  : Less significant bits
 *   #define VIS_VALUES   (1<<VIS_MSB) : Number of intervals
 *   #define VIS_SHIFT    (1<<VIS_LSB) : Shift for loops
 * @ These values are used internally and are based on the VIS_MSB value. They are used to define internal
 * values to deal with the RGB 3D colour space.
 *
 * Distances lookup tables : normal table, border table, angle tables
 * ------------------------------------------------------------------
 *   #define VIS_ROW    0
 *   #define VIS_COL    1
 *   #define VIS_ALPHA  0
 *   #define VIS_BETA   1
 *   #define VIS_NORMAL 0
 *   #define VIS_BORDER 1
 * @ These values are used internally to access the right part of the distance tables.
 * Look the VIS_DistanceTables for more informations.
 *
 * * ***************** *
 * * typedef structures *
 * * ***************** *
 *
 * RGB space
 * ---------
 *   typedef BYTE VIS_RGBSpace[1<<VIS_MSB][1<<VIS_MSB][1<<VIS_MSB];
 * @ Definition of a RGB 3D colour space type. Each dimension is depending on the number of MSB used.
 *
 * RGB -> Hue conversion table
 *   typedef BYTE VIS_HueTable[1<<VIS_MSB][1<<VIS_MSB][1<<VIS_MSB];
 * @ Definition of a RGB->Hue 3D conversion table. Each dimension is depending on the number of MSB used.
 *
 * Median hue and automatic range structure
 * ----------------------------------------
 *   typedef struct {
 *     int Hue;   : Median hue value
 *     int Range; : Range around the median value
 * } VIS_DefineHue;
 * @ Used to get the 'median hue' and the 'hue range' from the function 'VIS_MedianHue'.
 *
 * Image processing struct : table -> undenoised class, index, X and Y_list denoised data
 * --------------------------------------------------------------------------------------
 *   typedef struct {
 *     BYTE *table;  : Binary table
 *     BYTE *X_list; : X positions list
 *     BYTE *Y_list; : Y positions list
 *     int  index;   : Number of elements in X_list and Y_list
 *   } VIS_Process;
 * @ The VIS_Process structure is used internally and contains a binary picture (table) of the colour-class,
 * a pixel list of the denoised binary picture (X and Y list) and the number of pixel in these lists (index).
 * 'index' is used to look for the limit pixels of the object in the denoised pixel lists.
 *
 * Algorithm options struct
 * ------------------------
 *   typedef struct {
 *     BYTE depth; : denoise one dot around
 *     BYTE fast;  : use fast algorithm, only for depth=1
 *     BYTE step;  : process every dots or two dots only
 *     BYTE loop;  : shift for loops, modified internally loop=step-1
 *   } VIS_Algo;
 * @ These values enable to choose between several optimised functions for image processing.
 * 'depth' indicates on how many pixels the "erosion" is applied.
 * 'fast'  indicates the use of an easier loop if the "erosion" is applied on only 1 pixel (depth==1).
 * 'step'  is either every pixels or every second pixels.
 * 'loop'  is used internally to define a shift value depending on 'step'.
 *
 * Object structure : class and limits
 * -----------------------------------
 *   typedef struct {
 *     BYTE class;                   : Class number of the object
 *     BYTE top, bot, right, left; : Coordinates of the found object
 *   } VIS_Object;
 * @ This structure is created by the user to get back the result. If an object has been found, 'class' will
 * contain the class number, i.e. VIS_NONE if no object from the current colour has been found.
 * 'top', 'bot', 'right' and 'left' contain the pixel limits of the found object in the image.
 *
 * Camera angles and positions offsets to calibrate
 * ------------------------------------------------
 *   typedef struct {
 *     int   alpha;  : Servo alpha offset
 *     int   height; : Camera height
 *     int   lenght; : Camera distance to the front of the robot
 *     float angle;  : Lens angle in radians
 *     float beta;   : Camera beta offset
 *   } VIS_CameraOffset;
 * @ This structure defines the camera properties and position.
 * 'alpha'  gives the servo position when the camera looks straight forward,
 * 'beta'   gives the angle offset between the camera middle axis and the horizontal axis,
 * 'lenght' gives the distance offset from the camera to the robot front,
 * 'height' gives the camera heigth to the ground,
 * 'angle'  defines the camera view angle.
 * @ These values are used to calibrate the ball distance to the camera and the angle positions.
 *
 * Camera offset default informations
 * ----------------------------------
 *   static const int   VIS_CAM_HEIGHT[2][5]={{-1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1}};
 *   static const int   VIS_CAM_LENGHT[2][5]={{-1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1}};
 *   static const BYTE  VIS_CAM_ALPHA[2][5]={{-1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1}};
 *   static const float VIS_CAM_BETA[2][5]={{-1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1}};
 *   static const float VIS_CAM_ANGLE[2][5]={{0.85, 0.85, 0.85, 0.85, 0.85}, {0.85, 0.85, 0.85, 0.85, 0.85}};
 * @ These values are used as defaults for the camera calibration (VIS_CameraOffset).
 * @ Angles informations have to be set up in radians.
 *
 * Distances lookup tables : normal table, border table, angle tables
 * ------------------------------------------------------------------
 *   typedef struct {
 *     short Dist_dist[2][imagerows][imagecolumns];     : Direct relative distance, border or normal
 *     short Dist_coord[2][2][imagerows][imagecolumns]; : Relative distance in x and y, border or normal
 *     float Dist_angle[2][imagerows][imagecolumns];    : Alpha and beta angles
 *   } VIS_DistanceTables;
 * @ This structure defines the distance table and is a global variable.
 * 'dist'  defines the direct distance between the robot and the ball,
 * 'coord' defines the side and depth coordinates of the ball compared to the camera,
 * 'angle' defines the object angle position compared to the camera.
 *
 * Distance structure
 * ------------------
 *   typedef struct {
 *     short d_row; : depth distance
 *     short d_col; : side distance
 *     short dist;  : direct distance : dist^2=d_row^2+d_col^2
 *     float alpha; : sideways angle
 *     float beta;  : up-down angle
 *   } VIS_Distance;
 * @ The distance structure is for the user to get the ball position using VIS_GetPosition.
 * @ The distance is only valid for the ball, but the angle position can be used for any object.
 *
 * * **************************************** *
 * * Global variables (defined in LibVision.c) *
 * * **************************************** *
 *
 *   VIS_HueTable       HueTable;       : Global hue-table conversion
 *   BYTE               HueTableOK=0;   : Flag if hue-table is initialised or not
 *   VIS_Algo           AlgoInfo;       : Global algorithm information
 *   BYTE               AlgoInfoOK=0;   : Flag if the algorithm informations are initialised or not
 *   VIS_CameraOffset   CamInfo;        : Camera setup for distance
 *   BYTE               CamInfoOK=0;    : Flag of the camera informations are initialised or not
 *   VIS_DistanceTables DistTables;     : Distance tables
```

```c
 *   BYTE               DistTablesOK=0; : Flag if the distance tables are initialised or not
 * @ These are the global variables declared in the LibVision.c program. They are the basic informations the
 * library needs to work.
 ****/

   /* C Libraries */
#include <eyebot.h>
#include <math.h>

   /* Camera offset default informations */
static const int   VIS_CAM_HEIGHT[2][5]={{59, 62, 55, 52, -1}, {-1, -1, -1, -1, -1}};
static const int   VIS_CAM_LENGHT[2][5]={{30, 33, 38, 20, -1}, {-1, -1, -1, -1, -1}};
static const BYTE  VIS_CAM_ALPHA[2][5]={{118, 139, 132, -1, -1}, {-1, -1, -1, -1, -1}};
static const float VIS_CAM_BETA[2][5]={{0.147, 0.026, 0.037, -1, -1}, {-1, -1, -1, -1, -1}};
static const float VIS_CAM_ANGLE[2][5]={{0.85, 0.85, 0.85, 0.85, 0.85}, {0.85, 0.85, 0.85, 0.85, 0.85}};

   /* Algorithm informations */
#define VIS_MSB         5   /* Most significant bits used for the 3D tables */
#define VIS_NONE        0   /* No colour class */
#define VIS_CONFLICT    255 /* Colour-conflict class */
#define VIS_SAT_THRES   32  /* Threshold for limit saturation */
#define VIS_BALL_RADIUS 21  /* ball radius */

    /* Defaults for the algorithm */
#define VIS_ALGO_DEPTH 1 /* "erosion" one dot around */
#define VIS_ALGO_FAST  1 /* Use fast function */
#define VIS_ALGO_STEP  2 /* Process every second pixel */
/* VIS_ALGO_LOOP is defined in the LibVision.c and is based on VIS_ALGO_STEP */

    /* Image size information */
#define VIS_IMROW imagerows    /* number of camera image rows */
#define VIS_IMCOL imagecolumns /* number of camrea image columns */

    /* Maximum number of classes to be processed */
#define VIS_MAX_CLASS 10 /* No more than 10 different classes, avoiding ANSI C warning */

    /* LCD Display informations, used by VIS_ComputeBW */
#define VIS_LCD_DISPLAY_ROW  64
#define VIS_LCD_DISPLAY_COL  128
#define VIS_LCD_DISPLAY_SIZE ((VIS_LCD_DISPLAY_ROW*VIS_LCD_DISPLAY_COL)>>3)

    /* RGB space */
typedef BYTE VIS_RGBSpace[1<<VIS_MSB][1<<VIS_MSB][1<<VIS_MSB];

    /* RGB -> Hue conversion table */
typedef BYTE VIS_HueTable[1<<VIS_MSB][1<<VIS_MSB][1<<VIS_MSB];

    /* Median hue and automatic range structure */
typedef struct {
  int Hue;   /* Median hue value */
  int Range; /* Range around the median value */
} VIS_DefineHue;

    /* Image processing struct : table -> undenoised class, index, X and Y_list denoised data */
typedef struct {
  BYTE *table;  /* Binary table */
  BYTE *X_list; /* X positions list */
  BYTE *Y_list; /* Y positions list */
  int  index;   /* Number of elements in X_list and Y_list */
} VIS_Process;

    /* Algorithm options struct */
typedef struct {
  BYTE depth; /* denoise one dot around */
  BYTE fast;  /* use fast algorithm, only for depth=1 */
  BYTE step;  /* process every dots or two dots only */
  BYTE loop;  /* shift for loops, modified internally loop=step-1 */
} VIS_Algo;

    /* Object structure : class and limits */
typedef struct {
  BYTE class;                /* Class number of the object */
  BYTE top, bot, right, left; /* Coordinates of the found object */
} VIS_Object;

    /* Camera angles and positions offsets to calibrate */
typedef struct {
  int   alpha;   /* Servo alpha offset */
  int   height;  /* Camera height */
  int   lenght;  /* Camera distance to the middle of the robot */
  float angle;   /* Lens angle in radians */
  float beta;    /* Camera beta offset */
} VIS_CameraOffset;

   /* Distances lookup tables : normal table, border table, angle tables */
typedef struct {
  short Dist_dist[2][imagerows][imagecolumns];     /* Direct relative distance, border or normal */
  short Dist_coord[2][2][imagerows][imagecolumns]; /* Relative distance in x and y, border or normal */
  float Dist_angle[2][imagerows][imagecolumns];    /* Alpha and beta angles */
} VIS_DistanceTables;

   /* Distance structure */
typedef struct {
  short d_row; /* depth distance */
  short d_col; /* side distance */
  short dist;  /* direct distance : dist^2=d_row^2+d_col^2 */
  float alpha; /* sideways angle */
  float beta;  /* up-down angle */
} VIS_Distance;

/*********************************************************************************************************/
/* LibVision functions                                                                                   */
/*********************************************************************************************************/
    /* Converts (R,G,B) triplet into Hue value */
BYTE VIS_RGB2Hue(BYTE R, BYTE G, BYTE B);
    /* Converts (R,G,B) triplet into Saturation value */
BYTE VIS_RGB2Sat(BYTE R, BYTE G, BYTE B);
    /* Converts (R,G,B) triplet into Intensity value */
BYTE VIS_RGB2Int(BYTE R, BYTE G, BYTE B);
    /* Fills the Hue table for faster conversion */
void VIS_FillHueTable(void);
    /* Fills the RGB space cube with a colour class */
void VIS_FillRGBSpace(BYTE HueMin, BYTE HueMax, BYTE Saturation, BYTE ColourClass, VIS_RGBSpace *RGBSpace);
    /* Fills a white class in the RGB space. The shade between black and white is defined by the intensity */
void VIS_FillWhiteClass(BYTE SatMin, BYTE SatMax, BYTE IntMin, BYTE IntMax,
                        BYTE ColourClass, VIS_RGBSpace *RGBSpace);
    /* Denoise every pixels */
int  VIS_ReduceNoise(VIS_Process *Process, int len);
    /* Denoise every pixels f i.e. no for, one pixel around */
int  VIS_ReduceNoisef(VIS_Process *Process);
    /* Denoise 2 i.e. every two pixels */
int  VIS_ReduceNoise2(VIS_Process *Process, int len);
    /* Denoise 2 i.e. every two pixels f i.e. fast (no for, one pixel around) */
int  VIS_ReduceNoise2f(VIS_Process *Process);
    /* Find the limits of the segmented objects */
void VIS_FindLimits(int* pTop, int* pBot, int* pRight, int* pLeft, VIS_Process *Process);
    /* Build a binary map for a specific colour class */
void VIS_FindClass(colimage *Pic, VIS_RGBSpace *RGBSpace, VIS_Process *Process, BYTE ColourClass);
    /* Build a binary map for an several colour classes */
void VIS_FindClasses(BYTE number, colimage *Pic, VIS_RGBSpace *RGBSpace, VIS_Process *Process,
                     BYTE *ColourList);
    /* Finds median hue and the hue range out of the middle part of the picture */
VIS_DefineHue VIS_MedianHue2(colimage *Pic, int size);
    /* Fills the hue conversion table and initialise the algorithm defaults */
void VIS_Init(void);
    /* Clears the RGB space */
void VIS_ColClear(VIS_RGBSpace *RGBSpace);
    /* Initialise a new colour class */
void VIS_ColInit(VIS_RGBSpace *RGBSpace, BYTE ColourClass);
    /* Finds the specified colour class */
int  VIS_ColFindOne(colimage *Pic, VIS_RGBSpace *RGBSpace, VIS_Object *Object, BYTE ColourClass);
    /* Finds several colour classes pur in a list */
int  VIS_ColFind(int number, colimage *Pic, VIS_RGBSpace *RGBSpace,
                 VIS_Object *Object, BYTE *ColourList);
    /* Camera angle offset */
void VIS_CamCal(VIS_RGBSpace *RGBSpace, BYTE ColourClass);
    /* Modify camera offset values */
int  VIS_ModifyCam(VIS_CameraOffset *NewCam);
    /* Modify algorithm values */
int  VIS_ModifyAlgo(BYTE NewDepth, BYTE NewFast, BYTE NewStep);

/*********************************************************************************************************/
/* Distance functions                                                                                    */
/*********************************************************************************************************/
    /* Check the CamInfo and initialise distances lookup tables */
BYTE VIS_InitDistance(void);
    /* Returns the distance of the ball by looking into the lookup tables */
int  VIS_GetPosition(VIS_Object *Object, VIS_Distance *ObjDist);

/*********************************************************************************************************/
/* Display functions                                                                                     */
/*********************************************************************************************************/
    /* BW LCD picture initialisation */
int  VIS_InitBwimg(BYTE *bwimg);
    /* Get a 2D binary array and make a linear BW array for LCD display */
void VIS_ComputeBW(BYTE *bwimg, BYTE *temp);
    /* Draw object limits and center on bw picture */
void VIS_DrawLimits(BYTE *bwimg, VIS_Object *Object, int value);
    /* Draw borders in a 1D array for LCD display */
void VIS_DrawBorder(BYTE *bwimg, int rows, int cols);
    /* Mark object position by drawing a vertical and horizontal white line through it */
void VIS_MarkObject(image greyimg, int x_middle, int y_middle, int object_size);
    /* Display error messages */
void VIS_ErrorDisplay(char title[16], int error, char msg[16]);
    /* Initialise the progress show of a loop (clear screen, display title,...) */
void VIS_InitShowProgress(char title[16]);
    /* Displays the progress of a loop */
void VIS_ShowProgress(int actual, int max);
    /* Displays the actual offset of the camera */
void VIS_ShowOffset(int alpha_offset, float beta_offset);

/*********************************************************************************************************/
/* Camera functions                                                                                      */
/*********************************************************************************************************/
    /* Initialisation of the camera. 10 retries */
int  VIS_InitCam(int cammode);
    /* Let time to camera for autobrightness */
void VIS_AutobrightnessDelay(int number);
    /* Returns a robot team and ID */
int  VIS_TeamRobot(void);
```

```c
/*****************************************************************************************************/
/*                  L I B V I S I O N   v2.00                                                        */
/*                                                                                                   */
/* Philippe LECLERCQ                                                                                 */
/* -----------------                                                                                 */
/*                   November 29th 2000                                                              */
/*                                                                                                   */
/*                                                                           lecle-p@ee.uwa.edu.au */
/*****************************************************************************************************/

/*****************************************************************************************************/
/* LibVision v2.00 functions                                                                         */
/*****************************************************************************************************/

#include "/home/post/lecle-p/LibVision/LibVision.h"
/* #include "LibVision.h" */

   /* Library informations, ONLY depending on VIS_MSB in the .h */
#define VIS_LSB    (8-VIS_MSB)  /* Less significant bits */
#define VIS_VALUES (1<<VIS_MSB) /* Number of intervals */
#define VIS_SHIFT  (1<<VIS_LSB) /* Shift for loops */

   /* Algorithm default value (internal) : always Algo.step-1 */
#define VIS_ALGO_LOOP (VIS_ALGO_STEP-1)

   /* Distances lookup tables : normal table, border table, angle tables */
#define VIS_ROW    0
#define VIS_COL    1
#define VIS_ALPHA  0
#define VIS_BETA   1
#define VIS_NORMAL 0
#define VIS_BORDER 1

   /* Default value for VIS_RGB2Hue() function */
#define VIS_NO_HUE 255

   /* Minimum function */
#define MIN(a,b) (a<b?a:b)

   /* Maximum function */
#define MAX(a,b) (a>b?a:b)

   /* Global variables */
VIS_HueTable       HueTable;       /* Global hue-table conversion */
BYTE               HueTableOK=0;   /* Flag if hue-table is initialised or not */
VIS_Algo           AlgoInfo;       /* Global algorithm information */
BYTE               AlgoInfoOK=0;   /* Flag if the algorithm informations are initialised or not */
VIS_CameraOffset   CamInfo;        /* Camera setup for distance */
BYTE               CamInfoOK=0;    /* Flag of the camera informations are initialised or not */
VIS_DistanceTables DistTables;     /* Distance tables */
BYTE               DistTablesOK=0; /* Flag if the distance tables are initialised or not*/

   /* convert radian to degrees */
static double rad2deg(radians rad) { return 180*rad/M_PI;}

/*****************************************************************************************************/
/* LibVision functions                                                                               */
/*****************************************************************************************************/
/****f* LibVision/VIS_RGB2Hue
 * SYNOPSIS
 *   BYTE VIS_RGB2Hue(BYTE R, BYTE G, BYTE B)
 * DESCRIPTION
 *   Converts (R,G,B) triplet into Hue value between 0 and 252.
 * SEE ALSO
 *   VIS_RGB2Sat VIS_RGB2Int
 *
 * Thomas Braunl, UWA 1998.
 ****/
BYTE VIS_RGB2Hue(BYTE R, BYTE G, BYTE B)
{
  BYTE hue=0;
  BYTE max=MAX(R, MAX(G, B));
  BYTE min=MIN(R, MIN(G, B));
  BYTE delta=max-min;

  if (2*delta<=max) hue=VIS_NO_HUE;
  else {
    if (R==max) hue=42+42*(G-B)/delta; /* 1*42 */
    else if (G==max) hue=126+42*(B-R)/delta; /* 3*42 */
    else if (B==max) hue=210+42*(R-G)/delta; /* 5*42 */
    /* now: hue is in range [0..252] */
  }
  return hue;
}

/****f* LibVision/VIS_RGB2Sat
 * SYNOPSIS
 *   BYTE VIS_RGB2Sat(int R, int G, int B)
 * DESCRIPTION
 *   Converts (R,G,B) triplet into Saturation value between 0 and 255.
 * SEE ALSO
 *   VIS_RGB2Hue VIS_RGB2Int
 ****/
BYTE VIS_RGB2Sat(BYTE R, BYTE G, BYTE B)
{
  if ((R+G+B)==0) return 255; /* Avoid division by 0 */
  else return ((BYTE)((255-(765/(R+G+B)))*MIN(MIN(R, G), B)));
}

/****f* LibVision/VIS_RGB2Int
 * SYNOPSIS
 *   BYTE VIS_RGB2Int(int R, int G, int B)
 * DESCRIPTION
 *   Converts (R,G,B) triplet into Intensity value between 0 and 255.
 * SEE ALSO
 *   VIS_RGB2Hue VIS_RGB2Sat
 ****/
BYTE VIS_RGB2Int(BYTE R, BYTE G, BYTE B)
{
  return ((BYTE)((R+G+B)/3));
}

/****f* LibVision/VIS_FillHueTable
 * SYNOPSIS
 *   void VIS_FillHueTable(void)
 * DESCRIPTION
 *   Fills the RGB to Hue conversion table for faster conversion. Sets the HueTableOK flag to 1.
 *   The RGB->Hue conversion table is a global variable.
 * SEE ALSO
 *   VIS_FillRGBSpace VIS_FillWhiteClass
 ****/
void VIS_FillHueTable(void)
{
  int  max_loop=VIS_VALUES*VIS_VALUES; /* Last index for loop progress display */
  BYTE R_index, G_index, B_index;

  /* Initialise progress display */
  VIS_InitShowProgress("   RGB -> Hue");

  for (R_index=0; R_index<=VIS_VALUES; R_index++) {
    for (G_index=0; G_index<=VIS_VALUES; G_index++) {
      /* Display progress */
      VIS_ShowProgress(G_index+(R_index<<VIS_MSB),max_loop);

      for (B_index=0; B_index<=VIS_VALUES; B_index++) {
        /* Fill in table */
        HueTable[R_index][G_index][B_index]=VIS_RGB2Hue(R_index<<VIS_LSB, G_index<<VIS_LSB,
                                                        B_index<<VIS_LSB);
      }
    }
  }
  HueTableOK=1; /* Conversion table filled */
}

/****f* LibVision/VIS_FillRGBSpace
 * SYNOPSIS
 *   void VIS_FillRGBSpace(BYTE HueMin, BYTE HueMax, BYTE Saturation,
 *                         BYTE ColourClass, VIS_RGBSpace *RGBSpace)
 * DESCRIPTION
 *   Fills the RGB space cube with a colour class, using a saturation threshold.
 *   The saturation threshold can be the default defined one (VIS_SAT_THRES).
 *   'ColourClass' is defined by the user and is the reference to the colour-class.
 * SEE ALSO
 *   VIS_ColInit VIS_ColClear VIS_ColFind VIS_FillHueTable VIS_FillWhiteClass
 ****/
void VIS_FillRGBSpace(BYTE HueMin, BYTE HueMax, BYTE Saturation, BYTE ColourClass, VIS_RGBSpace *RGBSpace)
{
  int  max_loop=VIS_VALUES*VIS_VALUES; /* Last index for loop progress display */
  BYTE R_index, G_index, B_index;
  BYTE RGBValue, HueValue;
  BYTE test=0; /* HueMin<HueMax */

  /* Initialise progress display */
  VIS_InitShowProgress("    Fill RGB");

  /* Check if HueMin<HueMax */
  if (HueMin>HueMax) {test=1;}

  /* Loops */
  for (R_index=0; R_index<VIS_VALUES; R_index++) {
    for (G_index=0; G_index<VIS_VALUES; G_index++) {
      /* Display progress */
      VIS_ShowProgress(G_index+(R_index<<VIS_MSB),max_loop);

      for (B_index=0; B_index<VIS_VALUES; B_index++) {
        /* Init inner loop variables */
        RGBValue=(*RGBSpace)[R_index][G_index][B_index];
        HueValue=HueTable[R_index][G_index][B_index];

        /* Check the Saturation, to avoid white */
        if (VIS_RGB2Sat(R_index, G_index, B_index)>Saturation) {

          /* Sat>Threshold -> Check for any colour class conflict */
          if ((RGBValue==VIS_CONFLICT) || (RGBValue!=VIS_NONE)) {
            (*RGBSpace)[R_index][G_index][B_index]=VIS_CONFLICT; /* YES, use default colour conflict class */
            LCDSetPrintf(3, 0, "    Conflict"); /* Display if there was a class conflict */
          }
          else {
            /* NO conflict, check if corresponding Hue is within the limits */
            if (test) { /* Is HueMin<HueMax ? */
              if (((HueValue>=HueMin) && (HueValue<=252)) || (/* (HueValue>=0) && */ (HueValue<=HueMax))) {
                (*RGBSpace)[R_index][G_index][B_index]=ColourClass; /* YES, use the given class number */
                LCDSetPrintf(3, 0, "Col"); /* Display when it finds class */
                LCDSetPrintf(3, 12, "   ");
              }
            }
            else {
              if ((HueValue<=HueMax) && (HueValue>=HueMin)) {
                (*RGBSpace)[R_index][G_index][B_index]=ColourClass; /* YES, use the given class number */
                LCDSetPrintf(3, 0, "Col"); /* Display when it finds class */
                LCDSetPrintf(3, 12, "   ");
              }
            }
          }
        }
        else { /* Display when it's saturated */
          LCDSetPrintf(3, 12, "SAT");
          LCDSetPrintf(3, 0, "   ");
```

```c
        }
      }
    }
  }
}

/****f* LibVision/VIS_FillWhiteClass
 * SYNOPSIS
 *   void VIS_FillWhiteClass(int SatMin, int SatMax, BYTE IntMin, BYTE IntMax,
 *                           BYTE ColourClass, VIS_RGBSpace *RGBSpace)
 * DESCRIPTION
 *   Fills a white class in the RGB space.
 *   'SatMin' and 'SatMax' define the saturation range, 'SatMax' may be VIS_SAT_THRES for instance.
 *   The shade between black and white is defined by the intensity range (IntMin, IntMax).
 *   'ColourClass' is defined by the user and is the reference to the colour-class.
 * SEE ALSO
 *   VIS_ColInit VIS_ColClear VIS_ColFind VIS_FillHueTable
 ****/
void VIS_FillWhiteClass(BYTE SatMin, BYTE SatMax, BYTE IntMin, BYTE IntMax,
                        BYTE ColourClass, VIS_RGBSpace *RGBSpace)
{
  int  max_loop=VIS_VALUES*VIS_VALUES; /* Last index for loop progress display */
  BYTE R_index, G_index, B_index;
  BYTE RGBValue, SatValue, IntValue;

  /* Initialise progress display */
  VIS_InitShowProgress("   White Class");

  for (R_index=0; R_index<VIS_VALUES; R_index++) {
    for (G_index=0; G_index<VIS_VALUES; G_index++) {
      /* Display progress */
      VIS_ShowProgress(G_index+(R_index<<VIS_MSB),max_loop);

      for (B_index=0; B_index<VIS_VALUES; B_index++) {
        /* Init inner loop variables */
        RGBValue=(*RGBSpace)[R_index][G_index][B_index];
        SatValue=VIS_RGB2Sat(R_index, G_index, B_index);
        IntValue=VIS_RGB2Int(R_index, G_index, B_index);

        /* Check the Saturation and Intensity values */
        if ((SatValue<=SatMax) && (SatValue>=SatMin) && (IntValue<=IntMax) && (IntValue>=IntMin)) {

          /* Check for any colour class conflict */
          if ((RGBValue==VIS_CONFLICT) || (RGBValue!=VIS_NONE)) {
            (*RGBSpace)[R_index][G_index][B_index]=VIS_CONFLICT; /* YES, use default colour conflict class */
            LCDSetPrintf(3, 0, "    Conflict"); /* Display if there was a class conflict */
          }
          else {
            /* NO conflict, check if corresponding Hue is within the limits */
            if ((SatValue<=SatMax) && (SatValue>=SatMin)) {
              (*RGBSpace)[R_index][G_index][B_index]=ColourClass; /* YES, use the given class number */
              LCDSetPrintf(3, 0, "Col"); /* Display when it finds class */
            }
          }
        }
        else { LCDSetPrintf(3, 12, "SAT");} /* Display when it's saturated */
      }
    }
  }
}

/****f* LibVision/VIS_ReduceNoise
 * SYNOPSIS
 *   int VIS_ReduceNoise(VIS_Process *Process, int len)
 * DESCRIPTION
 *   Denoise every pixels with a depth of 'len'.
 * SEE ALSO
 *   VIS_ReduceNoisef VIS_ReduceNoise2 VIS_ReduceNoise2f
 ****/
int VIS_ReduceNoise(VIS_Process *Process, int len)
{
  int  l_index=0;
  int  row;
  int  imgcols=imagecolumns;
  BYTE *temp=Process->table, *X=Process->X_list, *Y=Process->Y_list;

  for (row=len+1; row<=(imagerows-len-2); row++) {
    int col;
    for (col=len+2; col<=(imgcols-len-2); col++) {
      int res=0, depth;
      for (depth=1; depth<=len; depth++) {
        int pos=row*imgcols+col;
        res+=(temp[pos-imgcols*depth] & temp[pos+depth] & temp[pos+imgcols*depth] & temp[pos-depth]);
      }
      if (res==len) { X[l_index]=row; Y[l_index]=col; l_index++;}
    }
  }
  return l_index-1;
}

/****f* LibVision/VIS_ReduceNoisef
 * SYNOPSIS
 *   int VIS_ReduceNoisef(VIS_Process *Process)
 * DESCRIPTION
 *   Denoise 'f' every pixels using the fast loop (i.e. one pixel around).
 * SEE ALSO
 *   VIS_ReduceNoise VIS_ReduceNoise2 VIS_ReduceNoise2f
 ****/
int VIS_ReduceNoisef(VIS_Process *Process)
{
  int  l_index=0;
  int  row;
  int  imgcols=imagecolumns;
  BYTE *temp=Process->table, *X=Process->X_list, *Y=Process->Y_list;

  for (row=2; row<=imagerows-1-2; row++) {
    int col;
    for (col=3; col<=imgcols-1-2; col ++) {
      int pos=row*imgcols+col;
      if (temp[pos-imgcols] & temp[pos+1] & temp[pos+imgcols] & temp[pos-1])
        { X[l_index]=row; Y[l_index]=col; l_index++;}
    }
  }
  return l_index-1;
}

/****f* LibVision/VIS_ReduceNoise2
 * SYNOPSIS
 *   int VIS_ReduceNoise2(VIS_Process *Process, int len)
 * DESCRIPTION
 *   Denoise '2' i.e. every two pixels.
 * SEE ALSO
 *   VIS_ReduceNoise VIS_ReduceNoisef VIS_ReduceNoise2f
 ****/
int VIS_ReduceNoise2(VIS_Process *Process, int len)
{
  int  l_index=0;
  int  row_index;
  int  imrows=(imagerows>>1), imcols=(imagecolumns>>1);
  BYTE *temp=Process->table, *X=Process->X_list, *Y=Process->Y_list;

  for (row_index=(len+1); row_index<=imrows-len-1; row_index++) {
    int row=(row_index<<1);
    int col_index;
      for (col_index=len+1; col_index<=imcols-len-1; col_index++) {
      int res=0, depth;
      for (depth=1<<1; depth<=(len<<1); depth+=2) {
        int pos=row_index*imcols+col_index;
        res+=(temp[pos-imcols*depth] & temp[pos+depth] & temp[pos+imcols*depth] & temp[pos-depth]);
      }
      if (res==len) {
        int col=(col_index<<1);
        X[l_index]=row; Y[l_index]=col; l_index++;
      }
    }
  }
  return l_index-1;
}

/****f* LibVision/VIS_ReduceNoise2f
 * SYNOPSIS
 *   int VIS_ReduceNoise2f(VIS_Process *Process)
 * DESCRIPTION
 *   Denoise '2f' i.e. every two pixels and fast (i.e. one pixel around).
 * SEE ALSO
 *   VIS_ReduceNoise VIS_ReduceNoisef VIS_ReduceNoise2
 ****/
int VIS_ReduceNoise2f(VIS_Process *Process)
{
  int  l_index=0;
  int  row_index;
  int  imrows=(imagerows>>1), imcols=(imagecolumns>>1);
  BYTE *temp=Process->table, *X=Process->X_list, *Y=Process->Y_list;

  for (row_index=1; row_index<=imrows-2; row_index++) {
    int row=(row_index<<1);
    int col_index;
    for (col_index=3; col_index<=imcols-2; col_index++) {
      int pos=row_index*imcols+col_index;
      if (temp[pos-imcols] & temp[pos+1] & temp[pos+imcols] & temp[pos-1]) {
        int col=(col_index<<1);
        X[l_index]=row; Y[l_index]=col; l_index++;
      }
    }
  }
  return l_index-1;
}

/****f* LibVision/VIS_FindLimits
 * SYNOPSIS
 *   void VIS_FindLimits(int* pTop, int* pBot, int* pRight, int* pLeft, VIS_Object *Object)
 * DESCRIPTION
 *   Find the limits of the segmented objects.
 * SEE ALSO
 *   VIS_FindLimits VIS_FindClass VIS_FindClasses VIS_ColFindOne VIS_ColFind
 ****/
void VIS_FindLimits(int* pTop, int* pBot, int* pRight, int* pLeft, VIS_Process *Process)
{
  int  i;
  int  right=*pRight, left=*pLeft;
  int  l_index=Process->index;
  BYTE *Y=Process->Y_list, *X=Process->X_list;

  /* Look for right and left values */
  for (i=l_index; i>=0; i--) { /*for (i=0; i<=l_index; i++) {*/
    int y_value=Y[i];
    if      (y_value<left) { left=y_value;}   /*if (Y[i]<*pLeft) { *pLeft=Y[i];}*/
    else if (y_value>right) { right=y_value;} /*if (Y[i]>*pRight) { *pRight=Y[i];}*/
  }
  /* Assign values */
  *pTop=X[0]; *pBot=X[l_index]; /* Values are inserted from TOP to BOTTOM in denoising function... */
  *pLeft=left; *pRight=right;
}

/****f* LibVision/VIS_FindClass
 * SYNOPSIS
 *   void VIS_FindClass(colimage *Pic, VIS_RGBSpace *RGBSpace, VIS_Object *Object, int ColourClass)
 * DESCRIPTION
```

```c
 *   Build a binary map for a specific colour class.
 *   Input  : a colimage, the RGBSpace to use, and the ColourClass to look for.
 *   Output : a VIS_Object structure containing the object informations.
 * SEE ALSO
 *   VIS_FindLimits VIS_ColFindOne VIS_ColFind VIS_FindClass VIS_FindClasses
 ****/
void VIS_FindClass(colimage *Pic, VIS_RGBSpace *RGBSpace, VIS_Process *Process, BYTE ColourClass)
{
  int           loop=AlgoInfo.loop; /* AlgoInfo is global */
  int           loop_offset=2>>loop;
  int           imrows=(imagerows>>loop), imcols=(imagecolumns>>loop);
  int           row_index=0; /* Processed picture index */
  unsigned char *table=Process->table;

  /* row_index and col_index : processed picture indexes (i.e. object indexes) */
  int offset=0; /* Object table linear index */
  for (row_index=0; row_index<=imrows-loop_offset; row_index++) {
    int row=(row_index<<loop);   /* Camera picture index */
    int col_index=0;             /* Processed picture index */
    for (col_index=loop_offset+1; col_index<=imcols-loop_offset; col_index++) {
      int col=(col_index<<loop); /* Camera picture index */

      /* Get class of the current pixel */
      unsigned char PixelClass=(*RGBSpace)[(*Pic)[row][col][0]>>VIS_LSB]
                                          [(*Pic)[row][col][1]>>VIS_LSB]
                                          [(*Pic)[row][col][2]>>VIS_LSB];

      /* Fill in the binary tables (undenoised) for the object */
      table[offset++]=(PixelClass==ColourClass); /* Fills table with 0 or 1 and update 'offset' */
    }
  }
}

/****f* LibVision/VIS_FindClasses
 * SYNOPSIS
 *   void VIS_FindClasses(BYTE number, colimage *Pic, VIS_RGBSpace *RGBSpace, VIS_Process *Process,
 *                        BYTE *ColourList)
 * DESCRIPTION
 *   Build a binary map for several colour classes.
 *   'Process' and 'ColourClass' are arrays having 'number' elements.
 * SEE ALSO
 *   VIS_FindLimits VIS_ColFindOne VIS_ColFind VIS_FindClass
 ****/
void VIS_FindClasses(BYTE number, colimage *Pic, VIS_RGBSpace *RGBSpace, VIS_Process *Process,
                     BYTE *ColourList)
{
  int loop=AlgoInfo.loop; /* AlgoInfo is global */
  int loop_offset=2>>loop;
  int imrows=(imagerows>>loop), imcols=(imagecolumns>>loop);
  int row_index=0; /* Processed picture index */
  int class;

  /* row_index and col_index : processed picture indexes (i.e. object indexes) */
  int offset=0; /* Object table linear index */
  for (row_index=0; row_index<=imrows-loop_offset; row_index++) {
    int row=(row_index<<loop);   /* Camera picture index */
    int col_index=0;             /* Processed picture index */
    for (col_index=loop_offset+1; col_index<=imcols-loop_offset; col_index++) {
      int col=(col_index<<loop); /* Camera picture index */

      /* Get class of the current pixel */
      unsigned char PixelClass=(*RGBSpace)[(*Pic)[row][col][0]>>VIS_LSB]
                                          [(*Pic)[row][col][1]>>VIS_LSB]
                                          [(*Pic)[row][col][2]>>VIS_LSB];

      /* Look through all the classes */
      for (class=0; class<=number; class++) {
        /* Fill in the binary tables (undenoised) for the object */
        Process[class].table[offset]=(PixelClass==ColourList[class]);
      }
      offset++; /* Update offset value */
    }
  }
}

/****f* LibVision/VIS_MedianHue
 * SYNOPSIS
 *   VIS_DefineHue VIS_MedianHue(colimage *Pic, int size)
 * DESCRIPTION
 *   Finds median hue and the hue range out of the middle part of the picture.
 *   'size' can not be more than 10.
 *   Returns a VIS_DefineHue structure containing the median Hue (VIS_DefineHue.Hue) and
 * the Hue range (VIS_DefineHue.Range).
 *   Both structure elements are set to -1 if an error occured.
 * SEE ALSO
 *
 ****/
VIS_DefineHue VIS_MedianHue(colimage *Pic, int size)
{
  int           Row, Col, RowStart, ColStart, RowPos, ColPos, Pos, Index;
  int           MedIndex=(size*size)>>1; /* (size*size)/2 we order only half the values */
  int           min, minIndex=0;
  int           minRange=255, maxRange=0;
  int           Array[10*10]; /* Limits size to 10 to get rid of the ANSI C variable-size warning */
  VIS_DefineHue Result;

  /* Check 'size' */
  if (size>10) {
    VIS_ErrorDisplay(" VIS_MedianHue", -1, "   size>10");
    Result.Hue=-1; Result.Range=-1;
    return Result;
  }

  /* Start indexes */
  RowStart=(imagerows-size)>>1; /* RowStart=(height/2)-(size/2)*/
  ColStart=(imagecolumns-size)>>1; /* ColStart=(width/2)-(size/2)*/

  /* Filling "Unsorted" table */
  Index=0;
  for (Row=0; Row<size; Row++) {
    for (Col=0; Col<size; Col++) {
      RowPos=RowStart+Row; ColPos=ColStart+Col; /* Picture indexes */
      Array[Index++]=VIS_RGB2Hue((*Pic)[RowPos][ColPos][0], /* Filling "Unsorted" table */
                                 (*Pic)[RowPos][ColPos][1],
                                 (*Pic)[RowPos][ColPos][2]);
    }
  }
  /* Loop finding the minRange and maxRange values */
  for (Index=0; Index<=MedIndex; Index++) {
    /* Sub-loop looking for the left local min */
    minRange=255; maxRange=0; /* Init. values */

    for (Pos=0; Pos<size*size; Pos++) {
      if (Array[Pos]<minRange) { minRange=Array[Pos];}
      if (Array[Pos]>maxRange) { maxRange=Array[Pos];}
    }
  }

  /* Loop finding the median value */
  for (Index=0; Index<=MedIndex; Index++) {
    /* Sub-loop looking for the left local min */
    min=255; /* Init. value */

    for (Pos=0; Pos<size*size; Pos++) {
      if (Array[Pos]<min) { /* Looking for min */
        min=Array[Pos]; /* YES, update min value and min position */
        minIndex=Pos;
      }
    }
    if (Index!=MedIndex) { /* Last Value ? Do not overwrite last value ! */
      Array[minIndex]=255; /* NO, Update "Unsorted" table*/
    }
  }

  /* Returning median value */
  Result.Hue=Array[minIndex];
  Result.Range=MAX(Result.Hue-minRange, maxRange-Result.Hue);

  return Result;
}

/****f* LibVision/VIS_Init
 * SYNOPSIS
 *   void VIS_Init(void)
 * DESCRIPTION
 *   Fills the hue conversion table and initialise the algorithm and camera offset defaults.
 * SEE ALSO
 *
 ****/
void VIS_Init(void)
{
  int RobotId, team, id; /* Robot identification */

  /* Fill the global hue table conversion */
  VIS_FillHueTable(); /* HueTableOK is set up by VIS_FillHueTable */

  /* Initialise the default algorithm values */
  AlgoInfo.depth=VIS_ALGO_DEPTH;
  AlgoInfo.fast=VIS_ALGO_FAST;
  AlgoInfo.step=VIS_ALGO_STEP;
  AlgoInfo.loop=VIS_ALGO_LOOP;
  AlgoInfoOK=1; /* Flag for Algorithm info structure initialised */

  /* Initialise the default camera offset values */
  RobotId=VIS_TeamRobot(); team=((int)(RobotId/10)); id=RobotId-10*team;
  CamInfo.angle=VIS_CAM_ANGLE[team][id];
  CamInfo.alpha=VIS_CAM_ALPHA[team][id];
  CamInfo.beta=VIS_CAM_BETA[team][id];
  CamInfo.height=VIS_CAM_HEIGHT[team][id];
  CamInfo.lenght=VIS_CAM_LENGHT[team][id];
  if ((CamInfo.angle==-1)||(CamInfo.alpha==-1)||(CamInfo.beta==-1)||
      (CamInfo.height==-1)||(CamInfo.lenght==-1)) {
    VIS_ErrorDisplay("VIS_Init", -1, "CamInfo incomp");
    CamInfoOK=0; /* One value is not set up properly... */
  }
  else { CamInfoOK=1;} /* Flag for camera offset structure initialised */
}

/****f* LibVision/VIS_ColClear
 * SYNOPSIS
 *   void VIS_ColClear(VIS_RGBSpace *RGBSpace)
 * DESCRIPTION
 *   Clears the RGB space: fills it with the VIS_NONE value.
 * SEE ALSO
 *   VIS_ColInit VIS_ColFind
 ****/
void VIS_ColClear(VIS_RGBSpace *RGBSpace)
{
  int  max_loop=VIS_VALUES*VIS_VALUES; /* Last index for loop progress display */
  BYTE R_index, G_index, B_index;

  /* Initialise progress display */
  VIS_InitShowProgress("   Clear RGB");

  for (R_index=0; R_index<VIS_VALUES; R_index++) {
    for (G_index=0; G_index<VIS_VALUES; G_index++) {
      /* Display progress */
      VIS_ShowProgress(G_index+(R_index<<VIS_MSB), max_loop);
```

```
      for (B_index=0; B_index<VIS_VALUES; B_index++) {
        /* Clear RGB space */
        (*RGBSpace)[R_index][G_index][B_index]=VIS_NONE;
      }
    }
  }
}

/****f* LibVision/VIS_ColInit
 * SYNOPSIS
 *   void VIS_ColInit(VIS_HueTable *HueTable, VIS_RGBSpace *RGBSpace, int ColourClass)
 * DESCRIPTION
 *   Initialise a new colour class.
 * SEE ALSO
 *   VIS_ColClear VIS_ColFind
 ****/
void VIS_ColInit(VIS_RGBSpace *RGBSpace, BYTE ColourClass)
{
  int           size=4, Key;
  BYTE          HueMin, HueMax;
  image         greyimg;
  colimage      Pic;
  VIS_DefineHue HueInfo;

  /* Offset calibration display */
  LCDClear();
  LCDSetPrintf(0, 0, "* Colour cal. *");
  LCDSetPrintf(1, 0, "---------------");
  LCDSetPrintf(2, 0, "  Align objet");
  LCDSetPrintf(3, 0, "  Choose size");
  LCDSetPrintf(4, 0, " with : + or -");
  LCDSetPrintf(5, 0, "Cal : calibrate");
  LCDMenu(" ", " ", " ", "Nxt");

  /* Wait for "Nxt" */
  KEYWait(KEY4);

  /* Initialise display */
  LCDClear();
  LCDSetPrintf(0, 11, "Size");
  LCDSetPrintf(2, 11, "Hue");
  LCDSetPrintf(4, 11, "Range");
  LCDMenu(" - ", " + ", " ", "Cal");
  Key=0;

  /* Waiting for "Col" */
  while (Key!=KEY4) {

    /* Get a picture */
    CAMGetColFrame(&Pic, 0);

    /* Display... */
    IPColor2Grey(&Pic, &greyimg);
    VIS_MarkObject(greyimg, imagerows/2-1,  imagecolumns/2-1, 0);
    LCDPutGraphic(&greyimg);

    /* Getting hue */
    HueInfo=VIS_MedianHue(&Pic, size);

    /* Reading a key from keyboard */
    Key=KEYRead();

    /* Changing 'size' value */
    if (Key==KEY1) { size--;}
    if (Key==KEY2) { size++;}
    if (size>10) { size=1;}
    if (size<1) { size=10;}

    /* Display information on LCD */
    LCDSetPrintf(1,11, " %d ", size);
    LCDSetPrintf(3,11, " %d ", HueInfo.Hue);
    LCDSetPrintf(5,11, " %d ", HueInfo.Range);
  }

  /* Displaying result, ask for HueStep */
  LCDClear();
  LCDSetPrintf(0,0, "* Colour cal. *");
  LCDSetPrintf(1,0, "---------------");
  LCDSetPrintf(2,0, "  Hue : %d ", HueInfo.Hue);
  LCDSetPrintf(4,0, "  Adjust Step");
  LCDSetPrintf(5,0, "  with + or -");
  LCDMenu("-", "+", " ", "Nxt");
  Key=0;

  /* Waiting for "Nxt" */
  while (Key!=KEY4) {

    /* Display information on LCD */
    LCDSetPrintf(3, 0, "  Step : %d ", HueInfo.Range);

    /* Reading a key from keyboard */
    Key=KEYRead();

    /* Changing Range value and displaying it */
    if (Key==KEY1) { HueInfo.Range--;}
    if (Key==KEY2) { HueInfo.Range++;}
    if (HueInfo.Range>252) { HueInfo.Range=0;}
    if (HueInfo.Range<0) { HueInfo.Range=252;}
  }

  /* Determine HueMin and HueMax */
  if (HueInfo.Hue-HueInfo.Range<0) {HueMin=252+HueInfo.Hue-HueInfo.Range;}
  else {HueMin=HueInfo.Hue-HueInfo.Range;}
  if (HueInfo.Hue+HueInfo.Range>252) {HueMax=HueInfo.Hue+HueInfo.Range-252;}
  else {HueMax=HueInfo.Hue+HueInfo.Range;}

  /* Filling RGB space with found Hue */
  VIS_FillRGBSpace(HueMin, HueMax, VIS_SAT_THRES, ColourClass, RGBSpace);
}

/****f* LibVision/VIS_ColFindOne
 * SYNOPSIS
 *   void VIS_ColFindOne(colimage *Pic, VIS_RGBSpace *RGBSpace, VIS_Object *Object, BYTE ColourClass)
 * DESCRIPTION
 *   Finds the specified colour class.
 *   Returns the ColourClass found : either ColourClass, either VIS_NONE.
 * SEE ALSO
 *   VIS_ColInit VIS_ColClear VIS_ColFind
 ****/
int VIS_ColFindOne(colimage *Pic, VIS_RGBSpace *RGBSpace, VIS_Object *Object, BYTE ColourClass)
{
  VIS_Process Process; /* Processing structure */
         BYTE table[VIS_IMROW*VIS_IMCOL];
         BYTE X_list[VIS_IMROW*VIS_IMCOL];
         BYTE Y_list[VIS_IMROW*VIS_IMCOL];

  /* Process structure init */
  Process.table=&table[0];
  Process.X_list=&X_list[0];
  Process.Y_list=&Y_list[0];

  /* Find colour in image */
  VIS_FindClass(Pic, RGBSpace, &Process, ColourClass);

  /* Reduce noise function selection : normal, fast, every 2nd pixels... */
  if      (AlgoInfo.fast && AlgoInfo.loop)  { Process.index=VIS_ReduceNoise2f(&Process);}
  else if (AlgoInfo.fast && !AlgoInfo.loop) { Process.index=VIS_ReduceNoisef(&Process);}
  else if (!AlgoInfo.fast && AlgoInfo.loop) { Process.index=VIS_ReduceNoise2(&Process, AlgoInfo.depth);}
  else                                      { Process.index=VIS_ReduceNoise(&Process, AlgoInfo.depth);}

  /* If an object has been found */
  if (Process.index>=0) {
    /* Find limits of object */
    int offset=(AlgoInfo.depth)*(1<<AlgoInfo.loop);
    int top=imagerows+1,  bot=-1, right=-1, left=imagecolumns+1;
    VIS_FindLimits(&top, &bot, &right, &left, &Process);

    /* Update object structure */
    Object->class=ColourClass;
    Object->top=top-offset;
    Object->bot=bot+offset;
    Object->right=right+offset;
    Object->left=left-offset;
  }
  /* No object has been found */
  else {
    Object->class=VIS_NONE;
  }

  /* Return result */
  return Object->class;
}

/****f* LibVision/VIS_ColFind
 * SYNOPSIS
 *   int VIS_ColFind(int number, colimage *Pic, VIS_RGBSpace *RGBSpace,
 *                   VIS_Object *Object, BYTE *ColourList)
 * DESCRIPTION
 *   Finds several colour classes pur in a list.
 *   Returns -1 if 'number' is bigger than the maximum number of classes, 0 for no errors.
 * SEE ALSO
 *   VIS_ColInit VIS_ColClear VIS_ColFindOne
 ****/
int VIS_ColFind(int number, colimage *Pic, VIS_RGBSpace *RGBSpace, VIS_Object *Object, BYTE *ColourList)
{
  int         index;
  VIS_Process Process[VIS_MAX_CLASS]; /* Processing structure */
         BYTE table[VIS_MAX_CLASS][VIS_IMROW*VIS_IMCOL];
         BYTE X_list[VIS_MAX_CLASS][VIS_IMROW*VIS_IMCOL];
         BYTE Y_list[VIS_MAX_CLASS][VIS_IMROW*VIS_IMCOL];

  /* Check that number~>VIS_MAX_CLASS */
  if (number>VIS_MAX_CLASS) {
    VIS_ErrorDisplay("VIS_ColFind", -1, "VIS_MAX_CLASS");
    return -1;
  }

  /* Process structure init */
  for (index=0; index<=number; index++) {
    Process[index].table=&table[index][0];
    Process[index].X_list=&X_list[index][0];
    Process[index].Y_list=&Y_list[index][0];
  }

  /* Find colour in image */
  VIS_FindClasses(number, Pic, RGBSpace, &Process[0], ColourList);

  /* Reduce Noise, find limits */
  for (index=0; index<=number; index++) {

    /* Reduce noise function selection : normal, fast, every 2nd pixels... */
    if      (AlgoInfo.fast && AlgoInfo.loop)  { Process[index].index=VIS_ReduceNoise2f(&Process[index]);}
    else if (AlgoInfo.fast && !AlgoInfo.loop) { Process[index].index=VIS_ReduceNoisef(&Process[index]);}
    else if (!AlgoInfo.fast && AlgoInfo.loop) { Process[index].index=VIS_ReduceNoise2(&Process[index],
                                                                                     AlgoInfo.depth);}
    else                                      { Process[index].index=VIS_ReduceNoise(&Process[index],
                                                                                    AlgoInfo.depth);}
```

```
    /* If an object has been found */
    if (Process[index].index>=0) {
      /* Find limits of object */
      int offset=(AlgoInfo.depth)*(1<<AlgoInfo.loop);
      int top=imagerows+1,  bot=-1, right=-1, left=imagecolumns+1;
      VIS_FindLimits(&top, &bot, &right, &left, &Process[index]);

      /* Update object structure */
      Object[index].class=ColourList[index];
      Object[index].top=top-offset;
      Object[index].bot=bot+offset;
      Object[index].right=right+offset;
      Object[index].left=left-offset;
    }
    /* No object has been found */
    else {
      Object[index].class=VIS_NONE;
    }
  }
  /* No error */
  return 0;
}

/****f* LibVision/VIS_CamCal
 * SYNOPSIS
 *   void VIS_CamCal(VIS_RGBSpace *RGBSpace, BYTE ColourClass)
 * DESCRIPTION
 *   Camera angle offset
 * SEE ALSO
 *
 ****/
void VIS_CamCal(VIS_RGBSpace *RGBSpace, BYTE ColourClass)
{
  int         imgrows=imagerows, imgcols=imagecolumns;
  int         middle=0;
  int         distance=260; /* in mm */
  char        Key=0;
  BYTE        test=0;
  BYTE        alpha_offset=CamInfo.alpha;
  BYTE        bwimg[VIS_LCD_DISPLAY_SIZE]; /* Declaration of LCD display 1D array */
  BYTE        FoundColour;
  float       beta_offset=CamInfo.beta;
  float       beta_th=0, beta_r=0;
  colimage    ColPic;
  VIS_Object  Object;
  VIS_Process Process;
  ServoHandle panning_camera=0;

  /* Process structure declaration */
  BYTE table[VIS_IMROW*VIS_IMCOL]; /* Binary table */
  BYTE X_list[VIS_IMROW*VIS_IMCOL]; /* X positions list */
  BYTE Y_list[VIS_IMROW*VIS_IMCOL]; /* Y positions list */

  /* Process structure init */
  Process.table=&table[0];
  Process.X_list=&X_list[0];
  Process.Y_list=&Y_list[0];

  /* Ask for calibration */
  VIS_ShowOffset(alpha_offset, beta_offset);
  LCDMenu("No", " ", " ", "Yes");

  /* Wait for Yes or No */
  while (Key!=KEY1 && Key!=KEY4) {
    Key=KEYRead();
  }

  /* Yes, calibration... */
  if (Key==KEY4) {

    /*-------------------------*/
    /* Alpha offset calibration */
    /*-------------------------*/

    /* Panning camera servo init */
    panning_camera=SERVOInit(SERVO11);

    /* BW picture initialisation */
    VIS_InitBwimg(bwimg);

    /* Draw border of camera picture */
    VIS_DrawBorder(bwimg, imagerows, imagecolumns);

    /* Alpha offset display */
    LCDClear();
    LCDSetPrintf(0,11, "Alpha");
    LCDSetPrintf(2,11, "Mid.");
    LCDMenu(" ", " ", " ", "Stop");

    /* Pan from 0 to 255 until middle is found */
    alpha_offset=0; test=0; Key=0;

    while ((!test) && (Key=KEYRead()!=KEY4))
            {
      SERVOSet(panning_camera, alpha_offset);

      /* Getting image */
      OSWait(50); /* Avoid tilting camera problems... */
      CAMGetColFrame(&ColPic, 0);

      /* Process to find classes */
      FoundColour=VIS_ColFindOne(&ColPic, RGBSpace, &Object, ColourClass);

      /* Modify 'bwimg' if object found */
      if (FoundColour==ColourClass) {
        middle=(Object.right+Object.left)>>1;
        VIS_DrawLimits(bwimg, &Object, 1);
      }

      /* Display... */
      LCDPutImage(bwimg);

      /* Erase object after displaying */
      if (FoundColour==ColourClass) {
        VIS_DrawLimits(bwimg, &Object, 0);
      }

      /* Display values */
      LCDSetPrintf(1,11, "%d ", alpha_offset);
      LCDSetPos(3,11);
      if (FoundColour==ColourClass) { LCDPrintf("%d ", middle-(imgcols/2));}
      else { LCDPrintf("    ");}
      LCDSetPrintf(4,11, "%d ", Object.right);
      LCDSetPrintf(5,11, "%d ", Object.left);

      /* Test end of alignment */
      if (middle==(imgcols/2)) { test=1;}
      /* NOT aligned... Keep going... */
      else { alpha_offset++;}

      /* NOT found ? Try again ! */
      if (alpha_offset>255) { AUTone(110,500); alpha_offset=0;}
    }

    /* Wait for anykey */
    LCDMenu("  A", "N Y", "K E", "Y");
    KEYWait(ANYKEY);

    /* Panning camera servo release */
    SERVORelease(panning_camera);

    /*-------------------------*/
    /* Beta offset calibration */
    /*-------------------------*/

    /* Offset calibration display */
    LCDClear();
    LCDSetPrintf(0,0, "* Offset cal. *");
    LCDSetPrintf(1,0, "---------------");
    LCDSetPrintf(2,0, "  Align. ball");
    LCDSetPrintf(3,0, " Adj. distance");
    LCDSetPrintf(4,0, " with : + or -");
    LCDSetPrintf(5,0, "Cal : calibrate");
    LCDMenu(" ", " ", " ", "Nxt");

    /* Wait for "Nxt" */
    KEYWait(KEY4);

    /* Initialise display */
    LCDClear();
    LCDSetPrintf(0,11, "Dist\n");
    LCDSetPrintf(3,10, "Offset");
    LCDMenu(" - ", " + ", " ", "Cal");
    Key=0;

    /* BW picture initialisation */
    VIS_InitBwimg(bwimg);

    /* Draw border of camera picture */
    VIS_DrawBorder(bwimg, imagerows, imagecolumns);

    /* Waiting for "Cal" */
    while (Key!=KEY4) {

      /* Getting image */
      CAMGetColFrame(&ColPic, 0);

      /* Process to find classes */
      FoundColour=VIS_ColFindOne(&ColPic, RGBSpace, &Object, ColourClass);

      /* Display border of object class */
      if (FoundColour==ColourClass) {
        middle=(Object.right+Object.left)/2;
        VIS_DrawLimits(bwimg, &Object, 1);
      }
      else { middle=-1;}

      /* Display... */
      LCDPutImage(bwimg);

      /* Erase object in 'bwimg' */
      if (FoundColour==ColourClass) {
        VIS_DrawLimits(bwimg, &Object, 0);
      }

      /* Reading a key from keyboard */
      Key=KEYRead();

      /* Changing distance value and displaying it */
      if (Key==KEY1) { distance-=10;}
      if (Key==KEY2) { distance+=10;}

      /* Process angles, offset and distance */
      beta_th=atan( (float) (CamInfo.height-VIS_BALL_RADIUS)/distance);
      beta_r=( (float) 0.75*CamInfo.angle/2)*
           ((( (float) (Object.top+Object.bot)/2)-( (float) imgrows/2))/
             ( (float) imgrows/2));
```

```
      beta_offset=beta_th-beta_r;

      /* Display information on LCD */
      LCDSetPrintf(1,11, "%d cm", (int) (distance/10));
      LCDSetPrintf(4,11, "%1.2f ", beta_offset);
    }

    /* Modifiy values */
    CamInfo.alpha=alpha_offset;
    CamInfo.beta=beta_offset;

    /* Check if now all informations are filled */
    if ((CamInfo.angle==-1)||(CamInfo.alpha==-1)||(CamInfo.beta==-1)||
                     (CamInfo.height==-1)||(CamInfo.lenght==-1)) {
      VIS_ErrorDisplay("VIS_Init", -1, "CamInfo incomp");
      CamInfoOK=0; /* One value is not set up properly... */
    }
    else { CamInfoOK=1;} /* Flag for camera offset structure initialised */

    /* Angle have been changed, need to recalculate distance table */
    DistTablesOK=0;

    /* Display new values */
    VIS_ShowOffset(alpha_offset, beta_offset);
    LCDMenu(" ", " ", " ", "Nxt");

    /* Wait for Nxt */
    KEYWait(KEY4);
  }
}

/****f* LibVision/VIS_ModifyCam
 * SYNOPSIS
 *   void VIS_ModifyCam(VIS_CamOffset *NewCam)
 * DESCRIPTION
 *   Modify camera offset values.
 *   Input  : a new 'VIS_CamOffset' structure.
 *           'VIS_CamOffset' is 5 values : angle and beta as floats
 *                                          alpha as BYTE
 *                                          height and lenght as int
 *   Output : 0 for no errors, 1 is the NewCam is not valid.
 * SEE ALSO
 *   VIS_ModifyAlgo
 ****/
int VIS_ModifyCam(VIS_CameraOffset *NewCam)
{
  if ((NewCam->angle==-1)||(NewCam->alpha==-1)||(NewCam->beta==-1)||
     (NewCam->height==-1)||(NewCam->lenght==-1)) {
    VIS_ErrorDisplay("   ModifyCam", -1, "NewCam invalid");
    CamInfoOK=0;
    return -1;
  }
  else {
    CamInfo.angle=NewCam->angle;
    CamInfo.alpha=NewCam->alpha;
    CamInfo.beta=NewCam->beta;
    CamInfo.height=NewCam->height;
    CamInfo.lenght=NewCam->lenght;
    CamInfoOK=1;
    return 0;
  }
}

/****f* LibVision/VIS_ModifyAlgo
 * SYNOPSIS
 *   void VIS_ModifyAlgo(VIS_Algo *NewAlgo)
 * DESCRIPTION
 *   Modify algorithm values.
 *   Input  : a new 'VIS_algo' structure.
 *           'VIS_Algo' is 4 integers : depth, fast, step and loop (modified automatically).
 *           Program is checking error in 'VIS_Algo' values, display them and return an error code.
 *           If an error occurs, no changes are made.
 *   Output : an error code : ==0 -> no errors; >0 -> an error occured.
 * SEE ALSO
 *   VIS_ModifyCam
 ****/
int VIS_ModifyAlgo(BYTE NewDepth, BYTE NewFast, BYTE NewStep)
{
  int error=0;

  /* error checking */
  if ((NewStep!=1) && (NewStep!=2)) {
    error=100; VIS_ErrorDisplay("! Param. error !", error, "step=only 1 or 2"); return error;
  }
  if ((NewFast==1) && (NewDepth>1)) {
    error=110; VIS_ErrorDisplay("! Param. error !", error, "fast for depth=1"); return error;
  }

  /* Modify if no errors */
  if (!error) {
    /* Modify Algo_info struct */
    AlgoInfo.depth=NewDepth;
    AlgoInfo.fast=NewFast;
    AlgoInfo.step=NewStep;
    AlgoInfo.loop=NewStep-1;
  }

  /* No errors occured... */
  return error;
}

/*****************************************************************************************************/
/* Distance functions                                                                                */
/*****************************************************************************************************/

/****f* LibVision/VIS_InitDistance
 * SYNOPSIS
 *   BYTE VIS_InitDistance(void)
 * DESCRIPTION
 *   Check the CamInfo and initialise distances lookup tables.
 *   Output : returns 0 if VIS_struct is set up properly, >0 if not or if beta angle not set up.
 * SEE ALSO
 *   VIS_GetPosition()
 ****/
BYTE VIS_InitDistance(void)
{
  int   error=0;
  int   row=0, col=0;
  int   d_norm=0, d_bord=0;
  int   imgrows=imagerows, imgcols=imagecolumns;
  float alpha=0, beta=0;
  float temp, cos_alpha, sin_alpha;

  /* Check if the CamInfo struct has been filled or not */
  if (CamInfoOK) {

    /* Initialise show progress display */
    VIS_InitShowProgress("* Distance cal.\n");

    for (row=0; row<imgrows; row++) {
      beta=(0.75*(0.5*CamInfo.angle))*(row-(0.5*imgrows))/(0.5*imgrows);
      beta+=CamInfo.beta;

      temp=(float) (CamInfo.height-VIS_BALL_RADIUS);

      d_norm=(short) (temp/beta);
      d_bord=(short) ((temp-(0.5*VIS_BALL_RADIUS))/beta);

      for (col=0; col<imgcols; col++) {

        alpha=0.5*CamInfo.angle*(( (float) (2*col)/(imgcols))-1);
        cos_alpha=cos(alpha);
        sin_alpha=sin(alpha);

        DistTables.Dist_angle[VIS_ALPHA][row][col]=alpha;
        DistTables.Dist_angle[VIS_BETA][row][col]=beta;

        DistTables.Dist_dist[VIS_NORMAL][row][col]=d_norm;
        DistTables.Dist_dist[VIS_BORDER][row][col]=d_bord;

        DistTables.Dist_coord[VIS_NORMAL][VIS_ROW][row][col]=(short)d_norm*cos_alpha;
        DistTables.Dist_coord[VIS_NORMAL][VIS_COL][row][col]=(short)d_norm*sin_alpha;
        DistTables.Dist_coord[VIS_BORDER][VIS_ROW][row][col]=(short)d_bord*cos_alpha-VIS_BALL_RADIUS;
        DistTables.Dist_coord[VIS_BORDER][VIS_COL][row][col]=(short)d_bord*sin_alpha;
      }

      /* Still alive display... */
      VIS_ShowProgress(row,imgrows);
    }
    DistTablesOK=1; /* Distance lookup table initialsed */
  }
  else { /* CamInfo has not been defined */
    VIS_ErrorDisplay("   DistTables", -1, "CamInfo empty");
    error=-1;
  }
  return error;
}
/****f* LibVision/VIS_GetPosition
 * SYNOPSIS
 *   int VIS_GetPosition(VIS_Object *Object, VIS_Distance *ObjDist)
 * DESCRIPTION
 *   Returns the distance of the ball by looking into the lookup tables.
 *   A VIS_distance struct is made of 3 integers : d_row, d_col and dist.
 *   Where dist is the direct distance between the robot and the ball,
 *         (d_row,d_col) are the relative coordinates of the ball compared to the robot position,
 *   So that dist^2=d_row^2+d_col^2.
 *   d_row is the depth of the ball compared to the robot position,
 *   d_col is the side distance of the ball compared to the robot.
 *   Input  : an object structure (VIS_Object) and a VIS_Distance result structure.
 *   Output : 0 on success, -1 if distance table is not initialised.
 * SEE ALSO
 *   VIS_Init_distance()
 ****/
int VIS_GetPosition(VIS_Object *Object, VIS_Distance *ObjDist) {
  int       border=0;
  int       pcol=0, prow=0;
  int       error=0;

  /* Check if distance lookup table is updated */
  if (!DistTablesOK) {
    VIS_ErrorDisplay("  GetPosition", -1, "   DistTables");
    error=-1;
  }

  /* Look for ball position */
  if (Object->bot>=imagerows-5) {
    if (Object->left<=4) {
      prow=Object->top; pcol=Object->right;
      border=VIS_BORDER;
    }
    else if (Object->right>=imagecolumns-5) {
      prow=Object->top; pcol=Object->left;
      border=VIS_BORDER;
    }
    else {
      prow=Object->top; pcol=(Object->right+Object->left)>>1;
      border=VIS_BORDER;
    }
  }
```

```c
  else if (Object->left<=5) {
    prow=(Object->top+Object->bot)>>1; pcol=Object->right;
    border=VIS_NORMAL;
  }
  else if (Object->right>=imagecolumns-5) {
    prow=(Object->top+Object->bot)>>1; pcol=Object->left;
    border=VIS_NORMAL;
  }
  else {
    prow=(Object->top+Object->bot)>>1;
    pcol=(Object->right+Object->left)>>1;
    border=VIS_NORMAL;
  }

  /* Get values in the tables */
  ObjDist->d_row=DistTables.Dist_coord[border][VIS_ROW][(short) prow][(short) pcol];
  ObjDist->d_col=DistTables.Dist_coord[border][VIS_COL][(short) prow][(short) pcol];
  ObjDist->dist=DistTables.Dist_dist[border][(short) prow][(short) pcol];
  ObjDist->alpha=DistTables.Dist_angle[VIS_ALPHA][(short) prow][(short) pcol];
  ObjDist->beta=DistTables.Dist_angle[VIS_BETA][(short) prow][(short) pcol];

  return 0;
}

/*****************************************************************************************************************/
/* Display functions                                                                                             */
/*****************************************************************************************************************/
/****f* LibVision/VIS_InitBwimg
 * SYNOPSIS
 *   void VIS_InitBwimg(BYTE *bwimg)
 * DESCRIPTION
 *   BW LCD picture initialisation.
 *   Input  : 'Picture' structure.
 *   Output : 0 no errors, -1 NULL bwimg pointer.
 * SEE ALSO
 *   VIS_ComputeBW VIS_DrawLimits
 ****/
int VIS_InitBwimg(BYTE *bwimg)
{
  /* Init picture */
  if (NULL!=bwimg) {
    memset(bwimg, 0, VIS_LCD_DISPLAY_SIZE);
    return 0; /* Pointer was not NULL */
  }
  else { return -1;} /* Error pointer was NULL */
}

/****f* LibVision/VIS_ComputeBW
 * SYNOPSIS
 *   void VIS_ComputeBW(BYTE *bwimg, BYTE *temp)
 * DESCRIPTION
 *   Get a 2D binary array and make a linear BW array for LCD display.
 *   Input  : a BW linear array for LCD display (bwimg), a 2D binary array (temp).
 * SEE ALSO
 *   VIS_InitBwimg VIS_DrawLimits
 ****/
void VIS_ComputeBW(BYTE *bwimg, BYTE *temp)
{
  int row=0;
  int imgrows=VIS_LCD_DISPLAY_ROW, imgcols=VIS_LCD_DISPLAY_COL;
  int datasize=VIS_LCD_DISPLAY_SIZE;

  for (row=imgrows-1; row>=0; row--) { /*for (row=0; row<imagerows; row++) {*/
    int col;
    for (col=0; col<imgcols; col+=8) {
      int value=0x00;
      int i;
      for (i=7; i>=0; i--) { /*for (i=0; i<8; i++) {*/
        int offset=((row*imgcols)+(col+7-i)); /* [row][col+(7-i)] */
        if ((offset<datasize) && (temp[offset])) {
          value |= (1<<i);
        }
      }
      bwimg[(row<<4)+(col>>3)]=value; /* bwimg[row*128/8+col/8]=value; */
    }
  }
}

/****f* LibVision/VIS_DrawLimits
 * SYNOPSIS
 *   void VIS_DrawLimits(BYTE *bwimg, VIS_Object *Object, int value)
 * DESCRIPTION
 *   Draw object limits and center on bw picture.
 *   value==0 -> erase, value>0 -> draw.
 *   Input  : a BW linear array, the region to draw and a draw (i.e. >0) / erase (i.e. ==0) value.
 *   Output : N/A
 * SEE ALSO
 *   VIS_InitBwimg VIS_ComputeBW
 ****/
void VIS_DrawLimits(BYTE *bwimg, VIS_Object *Object, int value)
{
  int top=Object->top, bot=Object->bot;
  int right=Object->right, left=Object->left;

  if (value) { /* Display limits of object (value>0) */
    bwimg[(top<<4)+(right>>3)] |= 1<<(7-(right) % 8); /* bwimg[top*128/8+right/8] |= 1<<(7-(right) % 8); */
    bwimg[(top<<4)+(left>>3)] |= 1<<(7-(left) % 8);   /* bwimg[top*128/8+left/8] |= 1<<(7-(left) % 8); */
    bwimg[(bot<<4)+(right>>3)] |= 1<<(7-(right) % 8); /* bwimg[bot*128/8+right/8] |= 1<<(7-(right) % 8); */
    bwimg[(bot<<4)+(left>>3)] |= 1<<(7-(left) % 8);   /* bwimg[bot*128/8+left/8] |= 1<<(7-(left) % 8); */
    bwimg[((top+bot)<<3)+((right+left)>>4)] |= 1<<(7-((right+left)>>1) % 8);
    /* bwimg[((top+bot)/2)*128/8+((right+left)/2)/8] |= 1<<(7-((right+left)/2) % 8);*/
  }
  else { /* Erase limits of object (value==0) */
    bwimg[(top<<4)+(right>>3)] &= ~(1<<(7-(right) % 8)); /* bwimg[top*128/8+right/8]=~(1<<(7-(right) % 8)) */
    bwimg[(top<<4)+(left>>3)] &= ~(1<<(7-(left) % 8));  /* bwimg[top*128/8+left/8]=~(1<<(7-(left) % 8)) */
    bwimg[(bot<<4)+(right>>3)] &= ~(1<<(7-(right) % 8)); /* bwimg[bot*128/8+right/8]=~(1<<(7-(right) % 8)) */
    bwimg[(bot<<4)+(left>>3)] &= ~(1<<(7-(left) % 8));  /* bwimg[bot*128/8+left/8]=~(1<<(7-(left) % 8)) */
    bwimg[((top+bot)<<3)+((right+left)>>4)] &= ~(1<<(7-((right+left)>>1) % 8));
    /* bwimg[((top+bot)/2)*128/8+((right+left)/2)/8]=~(1<<(7-((right+left)>>1) % 8)) */
  }
}

/****f* LibVision/VIS_DrawBorder
 * SYNOPSIS
 *   void VIS_DrawBorder(BYTE *bwimg, int rows, int cols)
 * DESCRIPTION
 *   Draw borders in a 1D array for LCD display
 *   Input  : a 1D BW array, line and column of the border.
 *   Output : N/A
 * SEE ALSO
 *
 ****/
void VIS_DrawBorder(BYTE *bwimg, int rows, int cols)
{
  int i;

  for (i=cols; i>=0; i--) {                        /* for (i=0; i<=cols; i++) {*/
    bwimg[(rows<<4)+(i>>3)] |= 1<<(7-(i) % 8);     /* bwimg[rows*128/8+i/8] |= 1<<(7-(i) % 8); */
  }
  for (i=rows-1; i>=0; i--) {                      /* for (i=0; i<rows; i++) {*/
    bwimg[(i<<4)+(cols>>3)] |= 1<<(7-(cols) % 8); /* bwimg[i*128/8+cols/8] |= 1<<(7-(cols) % 8); */
  }
}

/****f* LibVision/VIS_MarkObject
 * SYNOPSIS
 *   void VIS_MarkObject(image greyimg, int x_middle, int y_middle, int object_size)
 * DESCRIPTION
 *   Mark object position by drawing a vertical and horizontal white
 *   line through it. Ball size is displayed by black lines.
 *   Birgit Graf, UWA 1998.
 ****/
/* -----------= colours =-------------- */
#define BLACK  0
#define WHITE 15

void VIS_MarkObject(image greyimg, int x_middle, int y_middle, int object_size)
{
  int row, column;

  /* mark object position: WHITE */
  for (row = 1; row < imagerows; row ++)
    greyimg[row][y_middle] = WHITE;

  for (column = 1; column < imagecolumns; column ++)
    greyimg[x_middle][column] = WHITE;

  /* mark object size: BLACK */
  for (row = x_middle - 2; row < x_middle + 3; row ++)
  {
    greyimg[row][y_middle - object_size/2] = BLACK;
    greyimg[row][y_middle + object_size/2] = BLACK;
  }

  for (column = y_middle - object_size / 2 + 1; column < y_middle +
         object_size / 2; column ++)
    greyimg[x_middle][column] = BLACK;
}

/****f* LibVision/VIS_ErrorDisplay
 * SYNOPSIS
 *   void VIS_ErrorDisplay(char title[16], int error, char msg[16])
 * DESCRIPTION
 *   Display error messages.
 *   Input  : a box title (16 char max.), an error code an error message (16 char max.).
 *   Output : N/A
 * SEE ALSO
 *
 ****/
void VIS_ErrorDisplay(char title[16], int error, char msg[16])
{
  AUTone(110, 500);
  LCDClear();
  LCDSetPrintf(0,0, title);
  LCDSetPrintf(1,0, "----------------");
  LCDSetPrintf(2,0, "  Error code :");
  LCDSetPrintf(3,0, "      %d", error);
  LCDSetPrintf(4,0, "   Message :");
  LCDSetPrintf(5,0, msg);
  LCDMenu("", "", "", "Cont");
  KEYWait(KEY4);
}

/****f* LibVision/VIS_InitShowProgress
 * SYNOPSIS
 *   void VIS_InitShowProgress(char title[16])
 * DESCRIPTION
 *   Initialise the progress show of a loop (clear screen, display title,...).
 *   Input  : a loop title (16 char max.).
 *   Output : N/A
 * SEE ALSO
 *
 ****/
void VIS_InitShowProgress(char title[16])
{
  LCDClear();
  LCDSetPrintf(0, 0, title);
```

```c
  LCDSetPrintf(2, 0, "I          I\n");
  LCDSetPrintf(2,13, "[ ]\n");
}

/****f* LibVision/VIS_ShowProgress
 * SYNOPSIS
 *   void VIS_ShowProgress(int actual, int max)
 * DESCRIPTION
 *   Displays the progress of a loop.
 *   Input  : the number of the actual loop and the total number of loops.
 *   Output : N/A
 * SEE ALSO
 *
 ****/
void VIS_ShowProgress(int actual, int max)
{
  int  col;
  char rotation[]={'|', '/', '-', '\\'};

  if (actual==0) {
    LCDSetPrintf(4,0, "  Please Wait");
  }
  col=(int) (actual/(max/10))+1;
  if (col>10) { col=10;}
  LCDSetChar(2,col, '.');
  LCDSetChar(2,14, rotation[actual % 4]);
}

/****f* LibVision/VIS_ShowOffset
 * SYNOPSIS
 *   void VIS_ShowOffset()
 * DESCRIPTION
 *   Displays the actual offset of the camera.
 *   'alpha' as a BYTE between 0 and 255, 'beta' as a float in degres.
 * SEE ALSO
 *
 ****/
void VIS_ShowOffset(int alpha_offset, float beta_offset)
{
  LCDClear();
  LCDSetPrintf(0,0, "* Offset cal. *");
  LCDSetPrintf(1,0, "---------------");
  LCDSetPrintf(3,0, "  Alpha: %d ", alpha_offset);
  LCDSetPrintf(4,0, " Beta : %1.3f d", rad2deg(beta_offset));
}

/***********************************************************************************************************/
/* Camera functions                                                                                        */
/***********************************************************************************************************/
/****f* LibVision/VIS_InitCam
 * SYNOPSIS
 *   int VIS_InitCam(int mode)
 * DESCRIPTION
 *   Initialisation of the camera. 10 retries.
 *   Input  : mode of the camera.
 *   Output : camversion or INITERROR.
 * SEE ALSO
 *   N/A
 ****/
int VIS_InitCam(int cammode)
{
  int  camversion=0;
  int  number=0;
  char rotation[]={'|', '/', '-', '\\'};

  LCDClear();
  LCDSetPrintf(0,0, "  Camera Init.");
  LCDSetPrintf(1,0, "  ------------");
  LCDSetPrintf(3,6, "[ ]");

  /* Wait for camera initialisation, get camera type */
  while ((INITERROR==(camversion=CAMInit(NORMAL))) && (number<=10) && (NOCAM!=camversion)){
    OSWait(CAMWAIT);
    LCDSetChar(3,7, rotation[(number++)%4]);
  }

  /* Bad initialisation after 10 times or no camera connected */
  if ((INITERROR==camversion) || (NOCAM==camversion)) { return camversion;}

  /* Camera initialised, return version */
  CAMMode(cammode);
  AUBeep();
  return camversion;
}

/****f* LibVision/VIS_AutobrightnessDelay
 * SYNOPSIS
 *   void VIS_AutobrightnessDelay(int number)
 * DESCRIPTION
 *   Let time to camera for autobrightness.
 *   Input  : number of waits.
 *   Output : N/A
 * SEE ALSO
 *   N/A
 ****/
void VIS_AutobrightnessDelay(int number)
{
  char     Key=0;
  colimage Pic;

  /* Initialise show progress display */
  VIS_InitShowProgress("*Autobrightness*\n");
  LCDMenu("No", " ", " ", "Yes");

  /* Wait for Yes or No */
  KEYGetBuf(&Key);
  while (Key!=KEY1 && Key!=KEY4) {
    KEYGetBuf(&Key);
  }
  AUBeep();

  /* Wait loop... */
  if (Key==KEY4) {
    int i;
    LCDMenu(" ", " ", " ", " ");
    for (i=0; i<number; i++) {
      /* Wait : i.e. take a picture */
      CAMGetColFrame (&Pic, 0);

      /* Still alive display */
      VIS_ShowProgress(i, number);
    }
  }
}

/****f* LibVision/VIS_TeamRobot
 * SYNOPSIS
 *   int VIS_TeamRobot(void)
 * DESCRIPTION
 *   Returns a robot team and ID.
 *   Team is 0 for Eyecam and 1 for Quickam, ID is between 1 and 5.
 *   Asks for robot ID if not between 1 and 5.
 *   Input  : N/A
 *   Output : a two digits integer xy where x is the team and y is the ID.
 * SEE ALSO
 *   N/A
 ****/
int VIS_TeamRobot(void)
{
  int  team=0;
  BYTE robot_id=0;
  char Key;

  robot_id=OSMachineID(); /* Get ID of the robot */
  team=CAMInit(NORMAL);   /* Quickcam : COLCAM=16, Eyecam : COLCAM+1 (i.e. 17) */

  if ((robot_id<1)||(robot_id>5)) {
    robot_id=1;
    VIS_ErrorDisplay("TeamRobot", -1, "bad robot_id");
    LCDClear();
    LCDSetPrintf(0, 0, "Enter Robot ID");
    LCDSetPrintf(3, 3, "Robot ID : %d", robot_id);
    LCDMenu("-", "+", "", "Cont");
    Key=0;
    while (Key!=KEY4) {
      Key=KEYRead();
      if (Key==KEY1) {robot_id--;}
      if (Key==KEY2) {robot_id++;}
      LCDSetPrintf(3, 3, "Robot ID : %d", robot_id);
    }
  }

  switch(team) {
    case COLCAM+1: /* EyeCam */
    case 32 :      /* Fixes the 32 instead of 17 (COLCAM+1) bug for EyeCam */
      return (robot_id-1);
      break;
    default :      /* QuickCam */
      return (10+robot_id-1);
      break;
  }
}
```

```c
#include "/home/post/lecle-p/LibVision/LibVision.h"
/* #include "LibVision.h" */

/* To remember the declared global variables (in LibVision.c) */
/* VIS_HueTable       HueTable;       */ /* Global hue-table conversion */
/* BYTE               HueTableOK=0;   */ /* Flag if hue-table is initialised or not */
/* VIS_Algo           AlgoInfo;       */ /* Global algorithm information */
/* BYTE               AlgoInfoOK=0;   */ /* Flag if the algorithm informations are initialised or not */
/* VIS_CameraOffset   CamInfo;        */ /* Camera setup for distance */
/* BYTE               CamInfoOK=0;    */ /* Flag of the camera informations are initialised or not */
/* VIS_DistanceTables DistTables;     */ /* Distance tables */
/* BYTE               DistTablesOK=0; */ /* Flag if the distance tables are initialised or not*/

int main(void)
{
  BYTE         ColourClass=1;               /* This example handles only one class */
  BYTE         FoundClass;                  /* Colour class found */
  BYTE         bwimg[VIS_LCD_DISPLAY_SIZE]; /* Declaration of LCD display 1D array */
  BYTE         draw;                        /* Flag if object drawn or not */
  int          error;                       /* Error number */
  int          timea, timeb;                /* Main function timers */
  char         Key;                         /* Keyboard... */
  colimage     ColPic;                      /* User's colour image */
  VIS_Object   Object;                      /* User's object */
  VIS_RGBSpace RGBSpace;                    /* User's RGB space */
  VIS_Distance ObjDist;                     /* User's distance structure */

  /******************/
  /* Initialisation */
  /******************/

  /* Camera Initialisation */
  error=VIS_InitCam(NORMAL);
  if (error==INITERROR) { VIS_ErrorDisplay("   Cam Init.", INITERROR, "");}

  /* Fill the hue conversion table and initialise the algorithm defaults */
  VIS_Init();

  /* Fill the user's RGB space with zeros */
  VIS_ColClear(&RGBSpace);

  /* Finding the median hue and filling the RGBSpace */
  VIS_ColInit(&RGBSpace, ColourClass);

  /* calibrate camera */
  VIS_CamCal(&RGBSpace, ColourClass);

  /* init distance table */
  VIS_InitDistance();

  /***************/
  /* Detect loop */
  /***************/

  /* Clear LCD display, display menu, clear the LCD 1D array and draw the camera picture borders */
  LCDClear();
  LCDMenu("", "", "", "END");

  /* Clear the LCD 1D array */
  error=VIS_InitBwimg(bwimg);
  if (error==-1) { VIS_ErrorDisplay(" bwimg pointer", error, "NULL");}

  /* Draw camera picture borders */
  VIS_DrawBorder(bwimg, 61, 81);

  /* Init variables... */
  draw=0; /* First loop, nothing has been drawn... */
  Key=0;

  /* Init display */
  LCDSetPrintf(0, 10, "Ball:\n");
  LCDSetPrintf(3, 10, "Time:\n");

  while (Key!=KEY4) {
    /* Erase object in LCD 1D array */
    if (draw) { VIS_DrawLimits(bwimg, &Object, 0);}

    /* Read Keyboard... */
    Key=KEYRead();

    /* Get a picture */
    CAMGetColFrame(&ColPic, 0);

    /* Find class */
    timea=OSGetCount();
    FoundClass=VIS_ColFindOne(&ColPic, &RGBSpace, &Object, ColourClass);
    timeb=OSGetCount();

    /* Draw limits of object if found */
    if (Object.class==ColourClass) { /* Could be 'if (FoundClass==ColourClass)' */
      VIS_DrawLimits(bwimg, &Object, 1); /* Draw limits in bwimg image */
      draw=1; /* Set flag that something has been drawn */
      VIS_GetPosition(&Object, &ObjDist);
      LCDSetPrintf(1, 11, "%3d  \n", ObjDist.d_row/10);
      LCDSetPrintf(2, 11, "%3d  \n", ObjDist.d_col/10);
    }
    else { /* Not object found, nothing drawn */
      draw=0;
      LCDSetPrintf(1, 11, "    ");
      LCDSetPrintf(2, 11, "    ");
    }

    /* Display timer */
    LCDSetPrintf(4, 11, "%d ", timeb-timea);

    /* Display bwimg (LCD 1D array) */
    LCDPutImage(bwimg);
  }
  return 0;
}
```