

## Image Processing Functions

<b>General Functions</b>	
<a href="#">mapcord</a>	Calculate image transformation co-ordinates
<a href="#">mapcoef</a>	Calculate image transformation coefficients
<a href="#">bmpread</a>	Read bmp file from disk
<a href="#">bmpwrite</a>	Write bmp file to disk
<a href="#">Code2RGB</a>	Convert color code array to RGB array
<a href="#">initcam</a>	Initialize camera
<b>Color Classification Functions</b>	
<a href="#">bmphdr</a>	Create Buffers and determine dimensions of Raw/Filtered image
<a href="#">delhdr</a>	Delete Buffers created by bmphdr()
<a href="#">GetImage</a>	Capture Image form Camera or File
<a href="#">GetColor</a>	Color Classification(Multi-thread).
<a href="#">GetColr</a>	Color Classification(Single-thread)
<a href="#">hsi</a>	Calculate the hue,relative saturation of a pixel
<b>Entity Classification Functions</b>	
<a href="#">GetEntity</a>	Entity Classification
<a href="#">MakEntity</a>	Create Entity of specified color from specified start point
<a href="#">delenty</a>	Delete dynamic arrays for GetEntity() outputs
<a href="#">makenty</a>	Create dynamic arrays for GetEntity() outputs
<a href="#">clrcmp</a>	Compare color of rectangular region w/ input
<a href="#">entity_wr</a>	Write Entity Classification o/p to disk file
<b>Object Classification Functions</b>	
<a href="#">robot_db</a>	Write Object Classification o/p to disk file
<a href="#">GetObject</a>	Field Object Classification
<a href="#">aread</a>	Classify Objects from AutoCad drawing
<a href="#">duprob</a>	Determine if object# is duplicated
<a href="#">emsg</a>	Object classification error codes
<a href="#">showerr</a>	Display message for to Object Classification error codes

## Data Structures Used

```
struct BMPF{           //Pixel data from BMP file
    COLOR *data;      //R,G,B pixel values of image
    DWORD w;         //pixel width of image
    DWORD h;         //pixel depth of image
};
//-----structures for Entity classification-GetEntity(), MakEntity()-----//

struct IMAGE{        //Array of Entities generated by GetEntity()
    DWORD *udp;      //udp[x=0 to ncc-1]. udp[x]=used depth of coli[x][]
    DWORD *adp;      //adp[x=0 to ncc-1]. adp[x]=depth of coli[x][]
    ENTITY **coli;   //coli[x=0 to ncc-1][y=0 to udp[x]]. ENTITY 'y' of color code(x)
    DWORD *num;      //num[x=0 to ncc-1]. num[x]=Number of pixels of each color code(x)
};

struct ENTITY{       //Properties of an Entity. Filled by MakEntity() & GetEntity()
    POINTI cg;       //CG coordinates
    DWORD vld;       //Entity is valid(most pixels have clrs color & pcnt is within limits)
    DWORD pcnt;      //body pixel count
    POINTI edge[4];  //extereme top(0),RH(1),bot(2) & LH(3) points
    int clrs;        //entity color code.
    POINTF tot;      //total of x,y coords of pixels in the entity
};
//-----structures for GetColor(),GetEntity()-----//

struct IMGF{         //o/p buffers-GetColor(),GetEntity()
    int *clrf;       //color coded o/p, index+1 in pst->r[],g[],b[] of corressponding b,g,r
    DWORD ht;        //pixel depth of scaled image
    DWORD wd;        //pixel width of scaled image
};

struct THDT{        //Data for GetColr()
    PSET *pst;       //Filter settings
    PSTG *ptg;       //Filter settings
    IMGF *rwi;       //Raw image i/p & filter o/p
    DWORD num;       //Number of threads
    DWORD ofs;       //Thread offset
};
//-----structures for GetObject(),aread()-----//

struct FOBJ{        //Properties of field object(element of rdb[])
    POINTF r_dp;     //CG of peripheral entitys-for object orientation
    POINTF r_cp;     //CG of center entity
    DWORD r_no;      //object number
    double r_dr;     //object(robot/ball) radius+robot exclusion zone(xzr)
    double r_db;     //object(robot/ball) radius+ball exclusion zone(xzb)
    double r_ad;     //object(robot/ball) radius
};

struct PEOB{        //peripheral "ENTITY"ies in object(element of pdb[])
    ENTITY *blobs;   //Array of peripheral "ENTITY"ies in object
    DWORD ncir;     //# peripheral entitys in object
};

struct FOBD{        //Field Object Details-o/p of GetObject(),aread(),mkrb()
    DWORD *orn;
    DWORD *nob;
    FOBJ *rdb;
    DWORD ard;
    DWORD urd;
};

nob  nob[i*]      -# of objects of type(i) found.
                Depth >= *ptg->ncl.(Depth= *ptg->ncl for robocp)

orn  orn[i*]      -Index in rdb[] of 1st element of Object type(i).
                Depth >= *ptg->ncl.(Depth= *ptg->ncl+1 for robocp,+1 for enemy goal)

rdb  rdb[]       -Field Object List. Filled by GetObject(),aread().

ard  ard         -Element count of rdb[].

urd  urd         -Element count of rdb[] relavent to current run of GetObject(),aread().

*i      : 0<=i< *ptg->ncl. Friend robot(0), Enemy robot(1), Ball(2)

ptg     : Input of type PSTG to GetObject(),aread()
```

## Settings-Additional

```
struct PSTG{
    DWORD ncc;
    DWORD ncl;
};
ncl    Number of Center Entity colors(Object types), width of pst->nob[],pst->np[]
ncc    # of colors/color segments in filtered o/p, color codes(1 to ncc) for Entities,
        # of elements at boundary 'm' in pst->H[],pst->S[],pst->I[]
```

## Settings-From Settings File

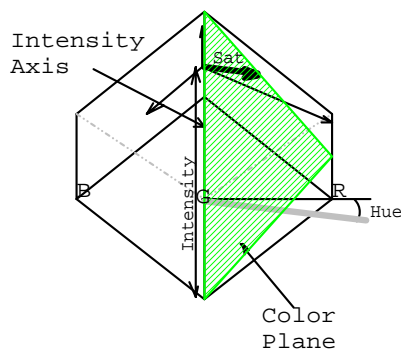
```
struct PSET{
    double *H;
    double *S;
    double *I;
    DWORD *pcnt;
    DWORD *r;
    DWORD *g;
    DWORD *b;
    char **fnm;
    DWORD *fop;
    DWORD *scale;
    DWORD *eclr;
    double *eint;
    DWORD *smth;
    DWORD *nob;
    double *pcd;
    DWORD *np;
};
H      H[i*]    -Hue of Color Segment 'j*', at boundary 'm*'
S      S[i*]    -Relative Saturation of Color Segment 'j*', at boundary 'm*'
I      I[i*]    -Intensity of Color Segment 'j*' at boundary 'm*'
pcnt   pcnt[i*] -Area Limit(in pixels) of Entity with color code 'j'+1', at boundary 'm*'.
r      r[j*]    -Red constituent of Color Segment 'j*'
g      g[j*]    -Green constituent of Color Segment 'j*'
b      b[j*]    -Blue constituent of Color Segment 'j*'
fnm    fnm[0]   -Pointer to name string of Color Classification o/p file.
        fnm[1]   -Pointer to name string of Entity Classification o/p file.
        fnm[2]   -Pointer to name string of Object Classification o/p file
        fnm[3]   -Pointer to name string of Smoothened o/p file.
        fnm[4]   -Pointer to name string of Bitmap image i/p file.
        fnm[5]   -Pointer to name string of AutoCad i/p file
fop    fop[0]   -Color Classification o/p to disk : Save(1)/No Save(0)
        fop[1]   -Entity Classification o/p to disk : Save(1)/No Save(0)
        fop[2]   -Object Classification o/p to disk : Save(1)/No Save(0)
        fop[3]   -Smoothened image o/p to disk : Save(1)/No Save(0)
        fop[4]   -Bitmap i/p is from : fnm[4](1)/camera(0)
        fop[5]   -Image processing i/p is from : AutoCad file fnm[5](1)/Bitmap(0)
scale  scale[0] -Division factor for rows & columns of Bitmap i/p:  $1 \leq scale[0] \leq (rwi \rightarrow w)/n$ 
eclr   eclr[0/1] -Color code for background/foreground intensity
eint   eint[0/1] -Intensity Limit for background/foreground intensity
smth   smth[0] -Smoothen filtered image in rwo->clrf[] i/p - Yes(1)/No(0)
nob    nob[k]   - $k = (0 * ptg \rightarrow ncl) + n$ : # of objects of type(n) expected.  $nob[k] \leq (np[k] + 1)^2 + 1$  (black).
         $k = (1 * ptg \rightarrow ncl) + n$ : Center Entity color code for Object type(n).
pcd    pcd[0/1] -Lower/Upper limit of Center to Peripheral Entity distance
np     np[n*]   -Number of Peripheral Colors(incl black) and Entities for Object type(n).
```

\* Note:-

m: Boundary-Lower(0)/Upper(1).  
j : Color Segment (0 to *ptg->ncc-1*) , color codes(j+1) in *rwo->clrf[]*: o/p of GetColor(), GetEntity()  
i :  $(m * ptg \rightarrow ncc) + j$ .  
n :  $0 \leq n < (ptg \rightarrow ncl)$ . friend-0, enemy-1, ball-2  
rwi : Object of type BMPF  
rwo: Object of type IMGF  
pst : Object of type PSET  
ptg : Object of type PSTG

# Color Classification

## COLOUR



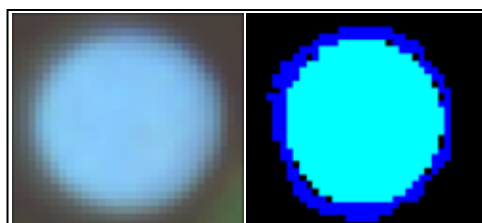
Defined by vectors(magnitude:0 to 255) in a color cube of R,G,B simple color constituents. Hold the color cube with the Intensity axis(connecting the two extreme vertices) perpendicular to the floor and the plane formed by the R,G,B vertices parallel to the floor, with the B and R vertices facing you.

The position(0-255) on the intensity axis, of the R,G and B vectors, is **Intensity(I)**(brightness). Length of the resultant is the **Saturation(S)**(white content or 'purity', analogous to the 'Contrast' of a TV picture). Angle of the resultant w/ the 'R' axis is the **Hue(H)**(Color). The **Color Plane** is formed by extending the intensity axis outwards to the edge of the cube, at the Hue angle. A color is specified by its (R,G,B) or (H,S,I) constituents.

## Noise

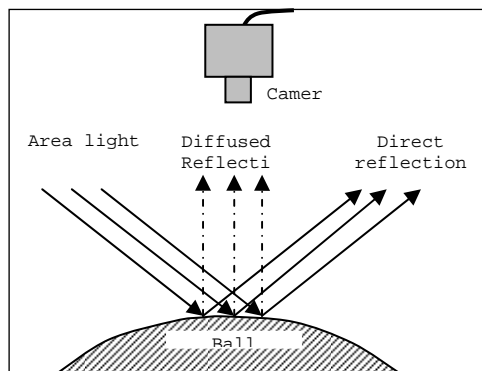
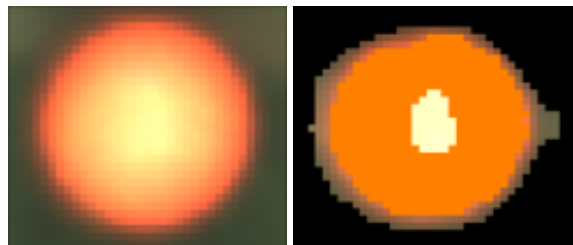
- Due to edge effect

Caused due to color mixing around the edges of blobs. Shades of grey are Neutral colors, they have no character of their own and absorb character from colors around them. This phenomenon affects blobs w/ grey backgrounds. Thus cyan mixes w/ black to give Periwinkle/lavender blue.

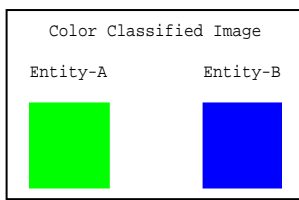
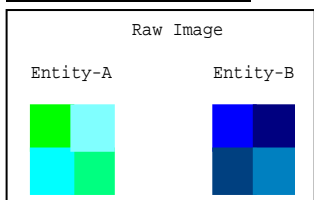


- Due to specular reflection

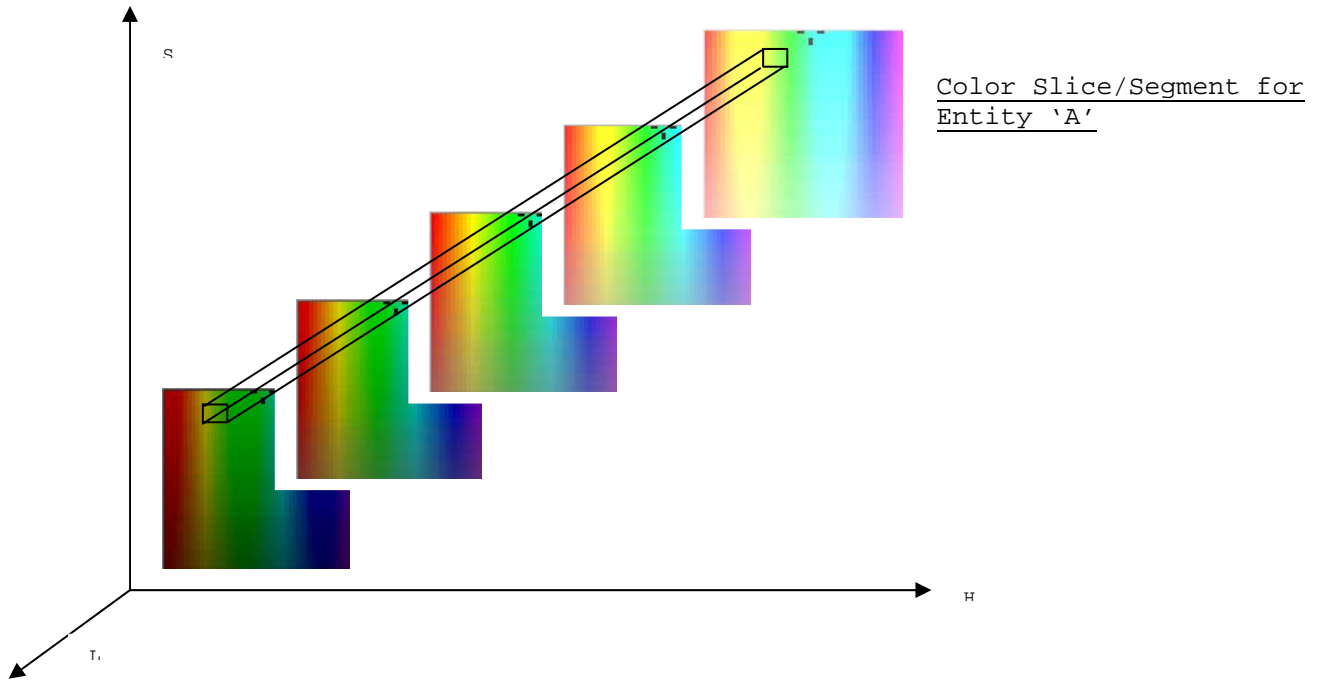
Affects curved surfaces. Below is the image of a ball, with a specular highlight at its centre and the Color Classified image. A less bright environment can help.



## Classification



Pixels of each entity in an image are in a range(slice/segment) of Hue(H), Saturation(S) & Intensity(I) values. Pixels within the specified ranges are converted to codes of pre-defined (pure) colors, by **color slicing/segmentation** in GetColor(). Its output is an array with the size of the scaled Raw image.



## Entity Classification

Color Classification results in a 2D array of codes of 'pure' colored pixels. The pixels are loose ie not identified with an Entity. Entity Classification by `GetEntity()`, identifies pixels to Entities.

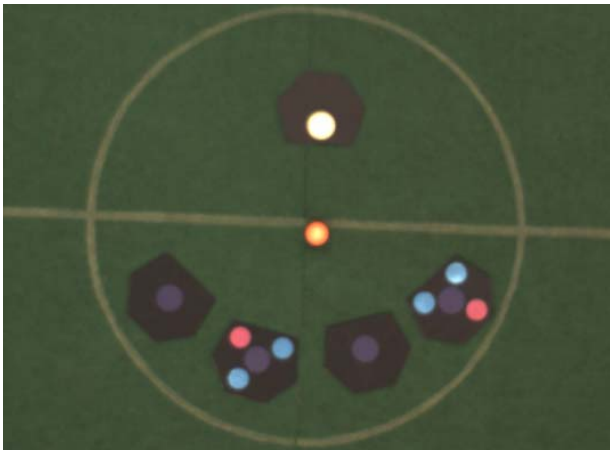


Figure 11.a: Raw image

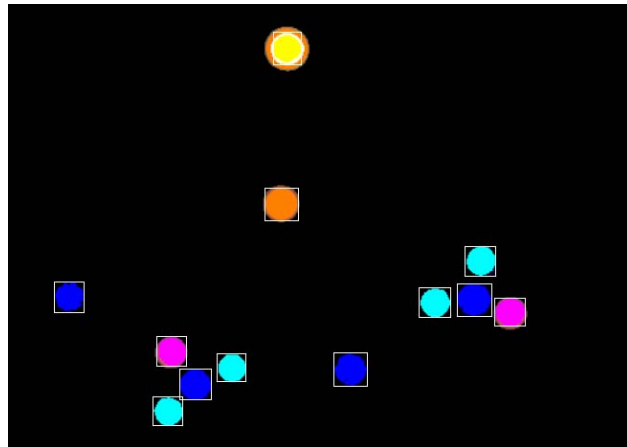
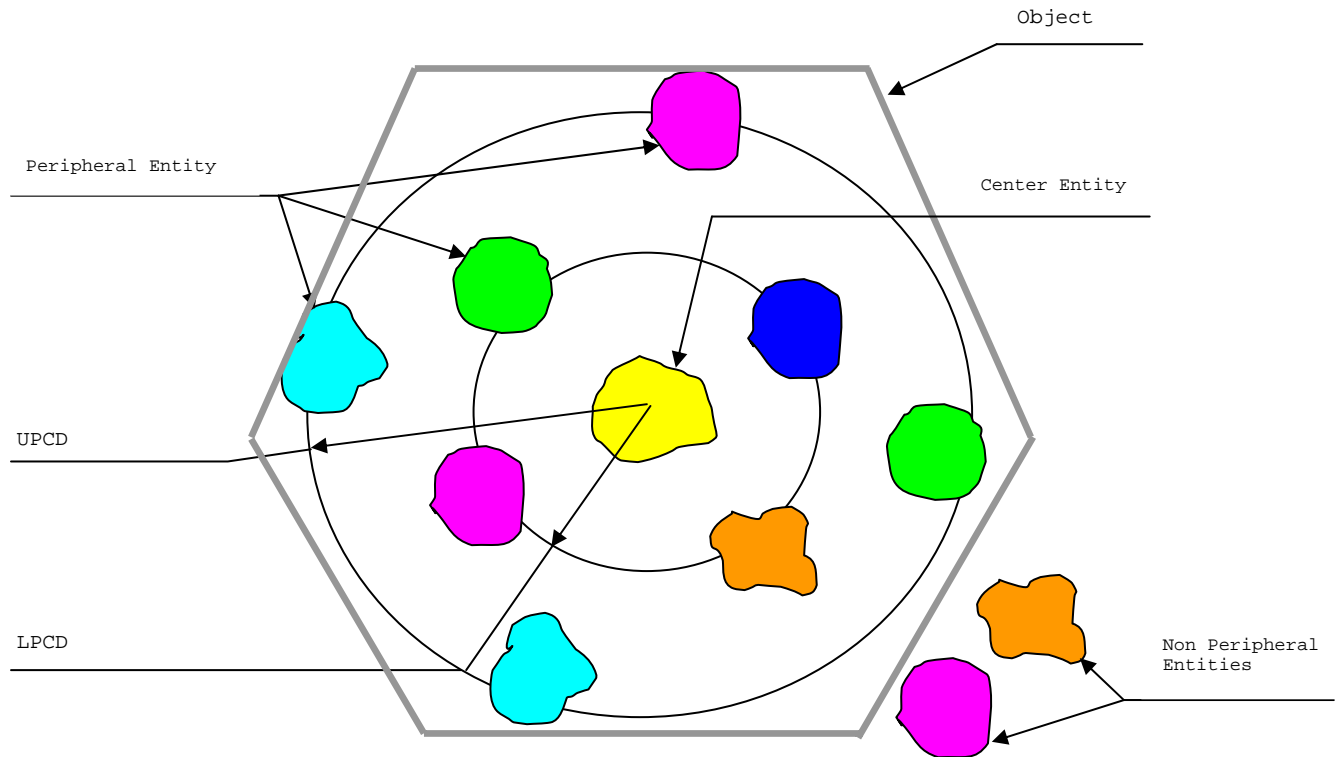


Figure 11.b: Color & Entity classified image.

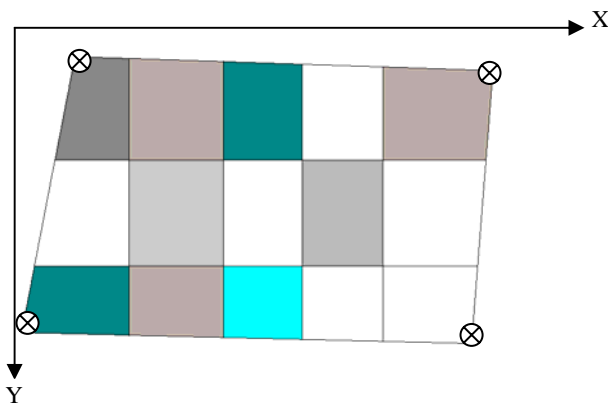
## Object Classification

After Color and Entity Classification a raster image is converted in to a two dimensional list of loose Entities, not identified with an Object. Object Classification by GetObject(), identifies Entities to Objects.

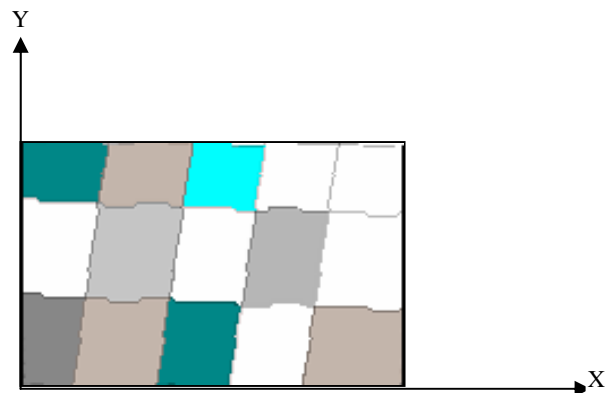


## Image Transformation

### Original Image



### Transformed Image



mapcoef() and mapcord() may be used to transform an image or specific points within it by skewing, stretching, rotating, mirroring or moving. Can be used on the output of Object Classification to correct object CG and orientation points for skew, stretch, mirroring and positioning problems in the raw image or AutoCad drawing inputs. mapcoef() is used to calculate transformation coefficients which are used by mapcord() to calculate a transformed point from a point in the original image or drawing.

## Color Classification Functions

### hsi

**void hsi(COLOR rgb, double \*ans)**

#### Description

Calculate the hue, relative saturation of a pixel from its RGB values

#### Parameters

*rgb* Red, Green & Blue constituents of pixel

#### Result

*ans[0]* Hue. MS-Paint value of Hue= $ans[0]*240/(2\pi)$

*ans[1]* Relative Saturation. MS-Paint value of saturation= $ans[1]*240$

*ans[2]* Intensity

*ans[3]* Absolute Saturation

### delhdr

**void delhdr(IMGF \*rwo, BMPF \*rwi);**

#### Description

Delete the dynamic buffers(*rwi->data[]* and *rwo->clrf[]*) created by *bmphdr()*

#### Parameters

*rwi->data[]* Array of 'COLOR' structures of the raw image

*rwo->clrf[]* Color Coded and scaled filter o/p

### bmphdr

**DWORD bmphdr(PSET \*pst, IMGF \*rwo, BMPF \*rwi, CArtCamSdk \*cam, DWORD scale, DWORD fi);**

#### Description

Create Buffers & determine dimensions for Raw & Color Classified image.

#### Parameters

*cam* pointer to CArtCamSdk class object, of camera functions.

*scale* Division factor for rows and columns of the i/p image to filtered image ( $\geq 1$ ).

*fi* 1: Raw image input is from *pst->fnm[4]*, 0: Raw image input is direct from camera

*pst* Members used: *fnm[4]*

#### Result

*rwi->h* Depth in pixels of unscaled i/p image

*rwi->w* Width in pixels of unscaled i/p image

*rwi->data[]* Array of 'COLOR' structures for i/p image. Number of elements =  $rwi->h * rwi->w$

*rwo->ht* Depth in pixels of scaled & filtered o/p image.  $ht = rwi->h / scale$ .

*rwo->wd* Width in pixels of scaled & filtered o/p image.  $wd = rwi->w / scale$ .

*rwo->clrf[]* Array for color Coded & scaled filter o/p. Number of integer elements =  $rwi->ht * rwi->wd$

#### Processing

If *fi=0*, *cam->Height()* and *cam->Height()* determine the dimensions of an image from the camera.

If *fi=1*, the dimensions of an image file are taken from the file header. Store these as the *h* and *w* and using *scale* to calculate *ht* and *wd*. Using *h*, *w*, *ht* and *wd* create empty dynamic buffers *data[]* and *clrf[]* which must be deleted by *delhdr()* when not required.

#### Returns

1- If successfully created otherwise 0.

# GetImage

```
void GetImage(char *fn, BMPF *rwi, CArtCamSdk *cam, DWORD fi);
```

## Description

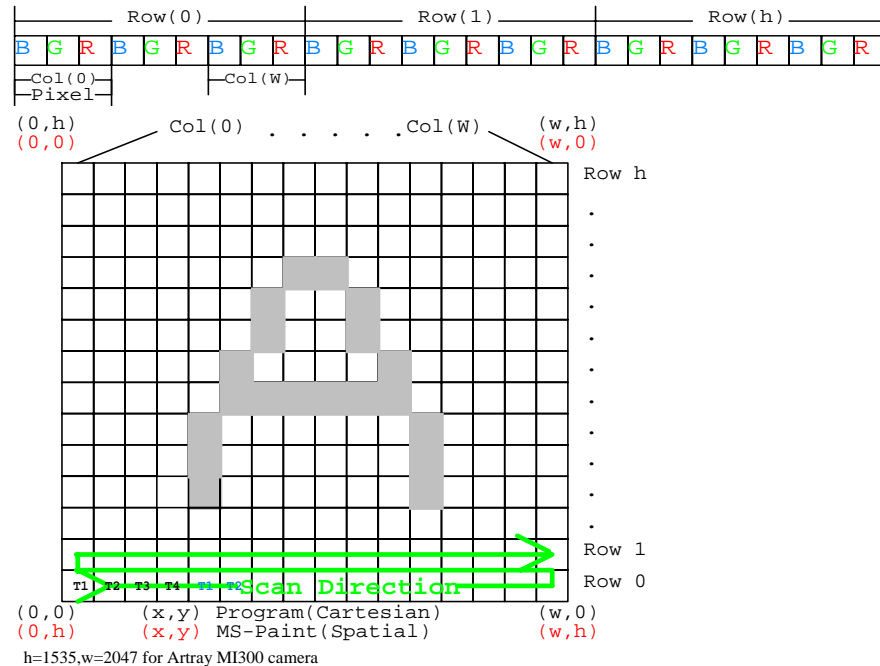
Read a 24 bit uncompressed windows bitmap image from camera( $fi=0$ ) or disk file( $fi=1$ ). Must be called once before GetImage() to determine the dimensions of the raw image and create the buffers to store it. When  $fi=0$  the **CArtCamSdk** class member function `cam->SnapShot()` is used to capture an image.

## Parameters

*cam* pointer to CArtCamSdk class object, of camera functions.  
*fi* Raw image input is from :*fn*(1), camera(0)  
*fn* Pointer to name and path string of bit map file to read  
*rwi* Address of variable w/ empty buffer & details of unscaled i/p image. Created by `bmphdr()`

## Result

*rwi->data[]* Array of 'COLOR' structures of the raw image. Number of elements= $rwi->h*rwi->w$ .



# GetColor

```
void GetColor(PSET *pst, PSTG *ptg, IMGF *rwo, BMPF *rwi, DWORD n);
```

## Description

Color Classify(filter) an image grabbed by GetImage()

## Parameters

*n* Number of GetColr() parallel threads(1 foreground and n-1 background threads).  
*pst* Members used: `scale[0],H[],S[],I[],r[],g[],b[],eclr[],eint[],fnm[0],fop[0]`  
*ptg* Members used: `ncc`  
*rwi* Addr of variable w/ pixel data & details of i/p image. Created by `bmphdr()`, filled by GetImage().  
*rwo* Addr of variable w/ empty buffer & details of result. Created by `bmphdr()`

## Result

*rwo->clrf[]* Color Coded and scaled Color Classified o/p. Number of elements= $rwo->ht*rwo->wd$

## Processing

Call `MakThread()` to create and execute n-1 background threads and 1 foreground thread of GetColr() to Color Classify the Raw image into a Color Coded and scaled output.

# GetColr

**void GetColr(void \*hMem);**

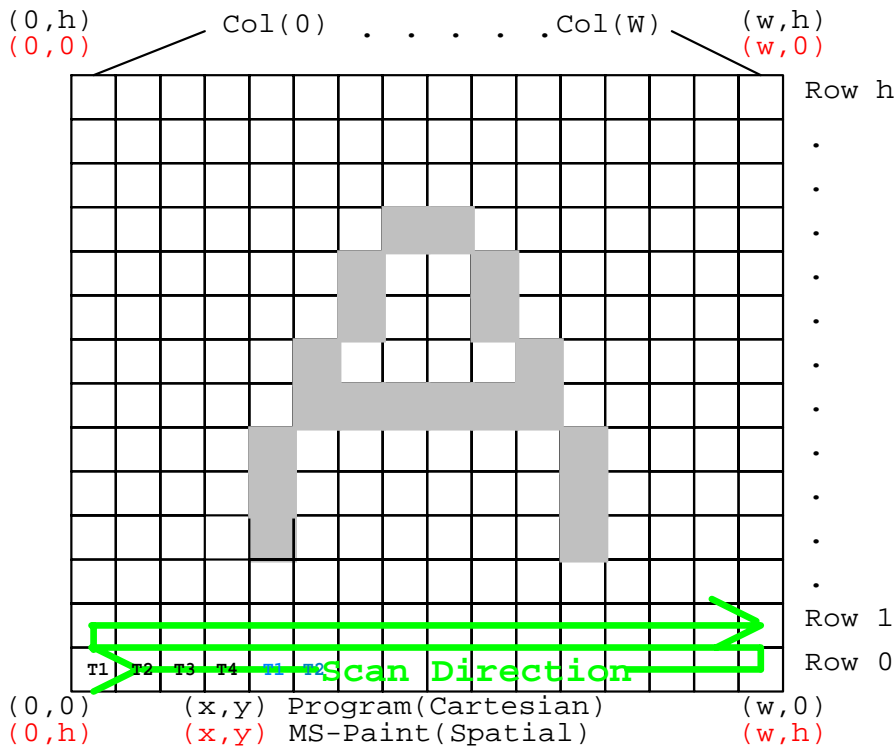
**Parameters**

*hMem* Address of a structure variable of type THDT with data for a single thread of Color Classification.

**Description**

Color Classifies a single thread of Raw image to a Color Coded and scaled output. The Raw image is scanned from the bottom left hand corner(0,0) towards the right and upwards to the top right hand corner. RGB of each pixel is converted to **H,S,I** and compared with the upper/lower thresholds for each complex color defined for the image. If a match is found the pixel is given the color code of the pure color defined for the segment otherwise it gets color code:0(black).

Pixels at the edge of the image are given color code zero, for Entity Classification to work correctly. Pixels w/ Intensity less than or equal to the **Background** limit or greater than or equal to the **Foreground** limit are given color codes defined for these limits(Pixels in overlapping limits get Foreground). Color codes selected for these Intensities should be 0(black) or different from those selected for segments, If not, they will be converted to Entities by Entity Classification.



**T1,2,3,4** - 1st pixel processed in parallel by threads 1 to 4(n)  
**T1,2,3,4** - 2nd pixel processed in parallel by threads 1 to 4(n)  
 n,h,w are passed in *hMem*

**Result**

*rwo*->*clrf*[ ] Color Coded and scaled Color- Classified o/p. for one thread. See GetColor().

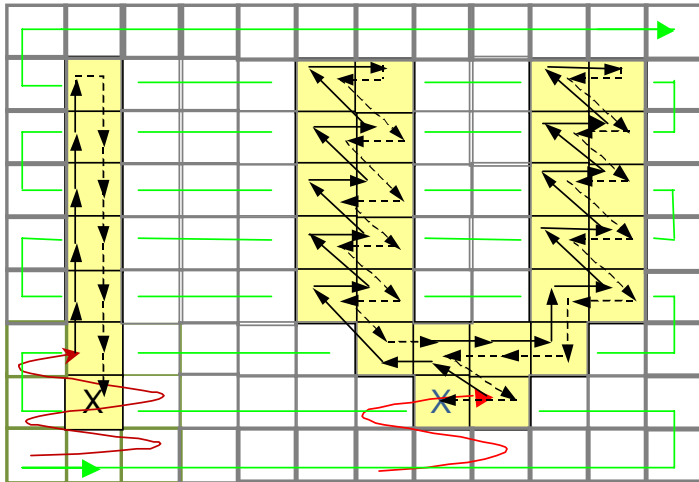
## Entity Classification Functions

# MakEntity

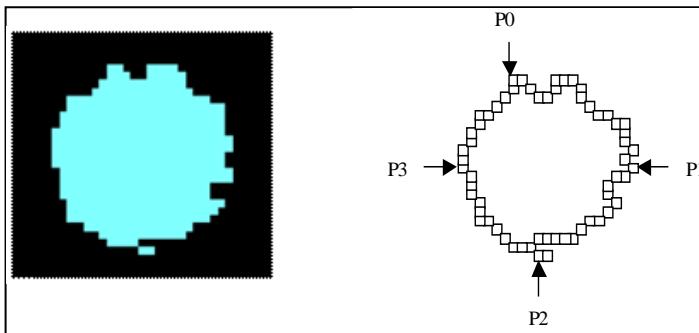
```
void MakEntity(ENTITY *sigma,DWORD x,DWORD y,DWORD fst,IMGF *rwo,DWORD smth);
```

### Description

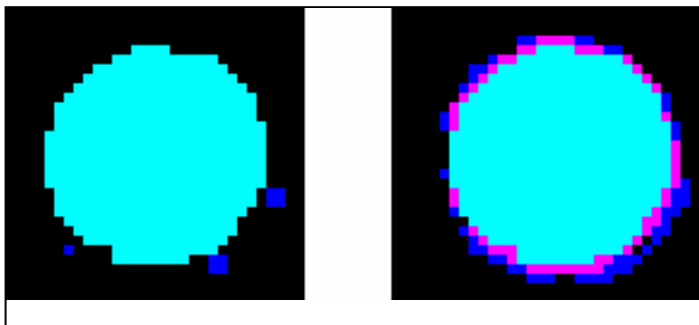
Called by GetEntity() to create an entity starting w/ 'Father' pixel(X) at (x,y). The color code of the pixel in *sigma->clrs* is negated, a Grid is placed on it and scanned for a pixel of the same color code(son). The first one encountered, it is made 'Father' and a recursive call to MakEntity() is made. The process repeats until a 'Son' pixel is not found, MakEntity() then returns to the father of the current pixel and the Grid scan continues for the remaining pixels. The process terminates when all the connected pixels in the region are visited and the algorithm returns to the first 'Father' pixel. Negating the color code of the 'Father', prevents the algorithm from recursing indefinitely. Pixels are negated & not blackened(code:0) to enable restoration by re-negating. The entity generated is returned in *sigma*.



The first 'Father' pixel is marked with X, the solid black arrows show the forward path of the flood scan algorithm and the dashed arrows the reverse path, returning back to X. The Green path shows the image scan skipping over entities and the thick boxes represent border pixels with color code=0.



While visiting the pixels of an entity the perimeter pixels are visited. If a perimeter pixel, is at the top, bottom, left or right extreme locations, it is stored in an array in *sigma->edge[]* in the following order: P0,P1,P2,P3



An Entity is Smoothed by deleting loosely attached Father pixels on its surface. A Father pixel is deleted by making its color code 0(black). This speeds up Entity creation by MakEntity() and image scanning by GetEntity(). A Father pixel, is loose if only one of its faces has another pixel of the same color code.

## Parameters

*sigma*->*clr* Pixel color code  
*x,y* co-ordinates of start(father) pixel in the Color Classification o/p array  
*fst* first call(1)/recursive call(0) to MakEntity()  
*rwo* Output of Color Classification o/p: GetColor()  
*smth* smoothen(1)/!smoothen(0) the entity

## Result

*sigma*->*pcnt* Pixel count in entity  
*sigma*->*clr* Pixel color code(negated)  
*sigma*->*edge[]* Entity edge pixels  
*sigma*->*tot.x* total of x coords of pixels in the entity  
*sigma*->*tot.y* total of y coords of pixels in the entity

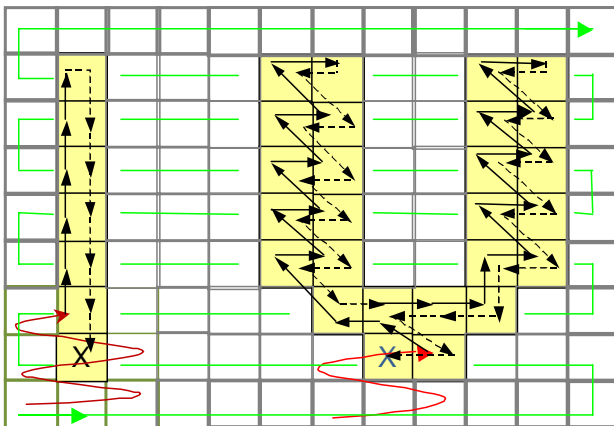
# GetEntity

```
DWORD GetEntity(PSET *pst,PSTG *ptg,IMGF *rwo,IMAGE *img);
```

## Parameters

*pst* Members used: *pcnt[],r[],g[],b[],fnm[1],fnm[3],fop[1],fop[3],smth[0]*  
*ptg* Members used: *ncc*  
*rwo* Output of Color Classification o/p: GetColor()

## Description



Classify the pixels of a Color Classified image into Entities. The function scans the image from the bottom left hand corner towards the right and upwards. When a pixel of a color code defined for an Entity is encountered, an Entity is generated by a call to MakEntity().

## Processing

A variable 'sigma' of type ENTITY is declared for Entity details. The color Classified image is scanned and every pixel encountered with a non zero, positive and within limits color code is converted to an Entity by MakEntity(). Properties of the Entity are filled into 'sigma' by MakEntity() & GetEntity(). The Entity is validated, its center of gravity calculated & the Entity(sigma) appended into the output array.

### 1. Validity

Noise effects make the generation of spurious Entities is possible. These Entities may be generated within valid entities, on their periphery or in open space. An Entity is valid if the below criterion are satisfied. Only valid entities are appended to the output array if VLDE(a preprocessor directive)=1. The results of validation are updated in the 'vld' member of 'sigma'.

#### 1.1 Area Comparison

The Area of an Entity is within the upper & lower limits specified for an Entity of that color code.

#### 1.2 Color Comparison

In a rectangular region generated from the Extreme points of an Entity, the pixels of each Entity color code are counted. The number of pixels with the color code of the Entity being checked must exceed the number of pixels with any other color code.

### 2. Center of Gravity

Calculated from the 'tot' and 'pcnt' members of 'sigma' and placed in the 'cg.x' and 'cg.y' members.

## Result

The following are returned in the *img* inputs

```
dpt[i] coli [i][j]
[0] 6 [0][0][1][2][3][4][5]
[1] 4 [1][0][1][2][3]
[2] 5 [2][0][1][2][3][4]
[3] 2 [3][0][1]
[4] 4 [4][0][1][2][3]
```

*img*->*udp*[i\*] Number of Entities generated of color code(i\*+1).

*img*->*coli*[i\*][j\*] Array of Entitys. Each element, of type ENTITY, corresponds to color code(i\*+1).  
*img*->*coli*[i] is address of the first j<sup>th</sup> element.

\*Note

i : 0<=i<(ptg->ncc).

j : 0<=j<(img->udp[i])

## delenty

**void delenty(PSTG \*ptg, IMAGE \*img)**

### Description

Delete dynamic arrays for GetEntity() outputs

### Parameters

*img* Output of Entity Classification: GetEntity()

*ptg* Members used: *ncc*

## makenty

**DWORD makenty(PSTG \*ptg, IMAGE \*img)**

### Description

Create dynamic arrays for GetEntity() outputs

### Parameters

*img* Output of Entity Classification: GetEntity()

*ptg* Members used: *ncc*

### Returns

0-Arrays could not be created

1-Arrays created successfully

# clrcmp

DWORD clrcmp(ENTITY \*sigma,PSTG \*ptg,IMGF \*roi,DWORD \*num)

## Description

Compare color of rectangular region circumscribing Entity *sigma* w/ color of the Entity.

## Parameters

*sigma*->*clrs* Color code of Entity  
*sigma*->*edge*[] Top y(0),RH x(1),Bot y(2) & LH x(3) extreme points of Entity  
*ptg* Members used: *ncc*  
*rwo* Output of Color Classification o/p: GetColor()

## Result

*num*[*i*] Pixel count of each color code eg *num*[0]=# pixels of color code 1.

\*note:  $0 \leq i < (ptg \rightarrow ncc)$

## Return

In the rectangular region circumscribing the Entity, If the number of pixels of the color code of the Entity are more than or equal to the number of pixels of any other color code, return(1) else return(0). Pixels w/ color code > (*ptg*->*ncc*) and smoothed pixels(color code=0) are ignored.(see GetEntity).

# entity\_wr

void entity\_wr(IMAGE \*img, char \*fn)

## Description

Write Entity Classification o/p to disk file

## Parameters

*fn* pointer to name and path string of output file  
*ptg* Members used: *ncc*  
*img* Output of Entity Classification: GetEntity()

## Object Classification Functions

# GetObject

DWORD \*GetObject(PSET \*pst,PSTG \*ptg,IMAGE \*img,FOBD \*fod);

## Parameters

*ptg* Members used: *ncc*, *ncl*  
*pst* Members used: *fnm*[2], *fop*[2], *pcd*[], *nob*[], *np*[]  
*img* Address of GetEntity() result holder.  
*fod* Address of GetObject() result holder

## Returns

Error code returned by emsg()

## Result

*fod* Output of Object Classification

## Description

Generates a Field Object list(*fod*->*rbd*[]) containing the number, location and orientation point(if peripheral entities are present) of each object. An **Object** is a Center Entity that may be surrounded by Peripheral Entities. Its properties are stored in Structure FOBJ. Peripheral Entities are related to the Center Entity by the distance of their centers. If this distance is within the specified limits, the peripheral Entity is a member of the Object. The *type* of Object is determined by the color of its Center Entity, thus an object with a Blue colored Center may be called a 'Friendly robot'.

**Processing**

Starts with the 1<sup>st</sup> Center Entity color(object type) in *pst->nob[]* & examines each valid Entity of that Color, in *img->coli[][]*. For the selected Center Entity, it selects the 1<sup>st</sup> Peripheral Entity color and examines each valid Entity of that color, in *img->coli[][]*. Peripheral Entities with a distance from the Center Entity within the limits set by *pst->pcd[0/1]* are included in *fod->rdb[]* as members of the center entity, the number of elements(*ard*) in *fod->rdb[]* is updated as is *fod->nob[]*. It then goes to the next Peripheral Entity color and repeats the process. After all the Peripheral Entity colors for the selected Center Entity are processed, the process repeats with the next Center Entity of the selected color. After all the Center Entities of the color are done, the program skips to the next Center Entity Color & repeats the process.

Once a Peripheral Entity becomes a member to an Object, it is not marked. In other words the *pcd[1]* circle must be within the Object to prevent Peripheral Entities becoming members to multiple Objects.

**Object Numbering**


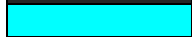




Object Classification starts with Object type(i=0) and proceeds upto  $i < ptg->ncl$ . Object numbers, given during Classification, are given as:-  
Object number = Segment value + offset.

Segment value is the highest Object # in the preceding segment.

If  $np[i]=1$  the offset is a sequential number from 1 onwards.

If  $np[i]>1$ , the offset is obtained by decoding the Peripheral Entities of the Object as shown below.

**Entity Details**

Entity Color	Color Name	Sticker Color (Printer Spec)			Color Code	Entity Type
		R	G	B		
	Black	-	-	-	0**	Peripheral
	Cyan	0	255	255	2**	Peripheral
	Magenta	255	0	255	3**	Peripheral
	Blue	0	0	255	1(nob[3])	Friendly Robot Center
	Yellow	255	255	0	4(nob[4])	Enemy Robot Center
	Orange	255	128	0	5(nob[5])	Ball Center

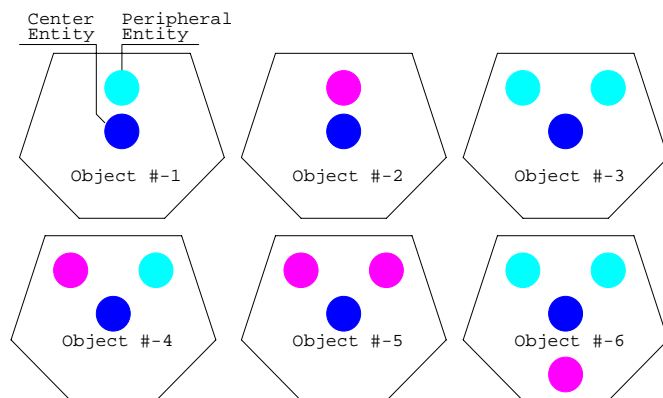
**Object Details**

Object Type (ptg->ncl=3)	i	Center Entity Color Code	#Peripheral Entities incl black	Peripheral Entity Color Code	# Objects
Friendly Robot	0	$nob[1*ptg->ncl+i]=1$	$np[i]=3$	0**, 2**, 3**	$nob[i]=8$
Enemy Robot	1	$nob[1*ptg->ncl+i]=4$	$np[i]=1$	-	$nob[i]=8$
Ball	$ptg->ncl-1$	$nob[1*ptg->ncl+i]=5$	$np[i]=1$	-	$nob[i]=1$

\*\*Peripheral Entity Color Codes must start from 2 and increment sequentially(no gaps) upto  $np[i]$

**Object Number**

Peripheral Entity Color Code Combination	Total	Object# (Total-1)
0,0,0(unused)	0	-1
0,0,2	2	1
0,0,3	3	2
0,2,2	4	3
0,3,2	5	4
2,2,2	6	5
0,3,3	6	5
3,2,2	7	6
3,3,2	8	7
3,3,3	9	8



As seen the number of positive & unique Object Numbers ie Number of Objects of type(i) is:  $nob[i] = np[i]^2 - 1$

## **Object Orientation**

The Center of Gravity of the Peripheral Entities can be used to determine the orientation of the Object(Angle wrt +ive x axis)

Angle= $\text{Tan}^{-1}(\text{dy}/\text{dx})$

Where: dy=Difference in Center of Gravity 'y' co-ordinate of Center and Peripheral Entities

dx=Difference in Center of Gravity 'x' co-ordinate of Center and Peripheral Entities

## **Error Detection**

Errors occur under the following conditions:-

1># of Peripheral Entities in object of type 'i' is  $> \text{np}[i]$

2># of Peripheral Entities in object of type 'i' is  $=0$  and  $\text{np}[i]>1$

3># of Objects of type 'i' found in returned 'nob[i]' are greater than expected in i/p 'nob[i]'

4>Duplicate Object numbers exist

5>The Peripheral Entities have been wrongly detected resulting in non sequential Object numbers

## **Limits**

$\text{np}[i]$  =Number of Peripheral Colors(incl black) and Entities for Object type(i)

$2 \leq \text{np}[i] \leq (\text{ptg} \rightarrow \text{ncc}) - (\text{ptg} \rightarrow \text{ncl}) + 1$

$\text{nob}[(1 * \text{ptg} \rightarrow \text{ncl}) + i]$  =Center color code for Object type (i)

$1 + \text{np}[i] \leq \text{nob}[(1 * \text{ptg} \rightarrow \text{ncl}) + i] \leq \text{ptg} \rightarrow \text{ncc}$  or  $\text{nob}[(1 * \text{ptg} \rightarrow \text{ncl}) + i] = 1$

Each  $\text{nob}[(1 * \text{ptg} \rightarrow \text{ncl}) + i]$  should be unique

$\text{nob}[(0 * \text{ptg} \rightarrow \text{ncl}) + i]$  =Number of Objects of type(i)

$\text{nob}[(0 * \text{ptg} \rightarrow \text{ncl}) + i] \leq \text{np}[i]^2 - 1$

where:  $0 \leq i < (\text{ptg} \rightarrow \text{ncl})$

## **aread**

**DWORD \*aread(FOBD \*fod,PSET \*pst,PSTG \*ptg)**

### **Description**

Classify Objects from AutoCad drawing. Generates a Field Object list( $\text{fod} \rightarrow \text{rbd}[\ ]$ ) containing the number, location and orientation point of each object. Objects other than friendly robots are numbered sequentially after the largest friendly robot number. Friendly robots numbers correspond to the object name e.g friendly robot 'F1' is given 1.

### **Parameters**

*ptg* Members used: *ncc, ncl*

*pst* Members used: *fnm[5], fnm[2], fop[2], pcd[], nob[], np[]*

*img* Address of GetEntity() result holder.

*fod* Address of GetObject() result holder

### **Returns**

Error code returned by *emsg()*

### **Result**

*fod* Output of Object Classification

## **robot\_db**

**void robot\_db(FOBD \*fod,char \*fn,PSTG \*ptg,PEOB \*pdb)**

### **Description**

Write Object Classification o/p to disk file

### **Parameters**

*fod* Output of Object Classification: GetObject(),aread()

*fn* Pointer to name and path string of output file

*ptg* Members used: *ncl*

*pdb* Peripheral entity list for all( $\text{fod} \rightarrow \text{ard}$ ) field objects

# duprob

**DWORD duprob(FOBD \*fod)**

## Description

Determine if object# of the object currently being appended into fod->rdb[] has been assigned earlier(duplicated).

## Parameters

*fod* Output of Object Classification: aread() or GetObject()

## Returns

1 - Duplicate Object number found

0 - Object number not duplicated

# emsg

**DWORD \*emsg(FOBD \*fod,PSET \*pst,PSTG \*ptg,DWORD pce,DWORD dup,FILE \*fptr)**

## Description

Object classification error codes

## Parameters

*fod* Output of Object Classification: aread() or GetObject()

*pst->nob[i]* # of objects of type(i) expected.

*ptg* Members used: *ncl*

*dup* Value returned by duprob()

## Returns

Address of unsigned integer array of 5 elements. Each element contains an error code pertaining to a specified condition and indicates an error if its value is non-zero.

ecode[0] - Number of objects of object type ecode[0] is incorrect.

ecode[1] - Peripheral entity detection incorrect

ecode[2] - Number of peripheral entities incorrect

ecode[3] - Duplicate object numbers exist

ecode[4] - Autocad i/p script file not found

# showerr

**void showerr(HWND hWnd,DWORD \*ecode)**

## Description

Display messages for error codes in ecode[], in window with handle *hWnd* .

## Parameters

*ecode* Array of error codes returned by emsg().

*hWnd* Handle of application window.

## General Functions

### Code2RGB

**BMPF \*Code2RGB(IMGF \*rwi,PSET \*pst){**

#### Description

Convert color coded image data in *rwi* to RGB data

#### Parameters

*rwi* Address of variable w/ details of color coded image  
*pst* Members used: *r[]*, *g[]*, *b[]*

#### Returns

A dynamic variable of type BMPF and a dynamic array within it with of 'COLOR' structures of the image. The variable and the array it points to must be deleted in the calling program.

### bmpwrite

**DWORD bmpwrite(char \*bmpfile,BMPF \*bmp,DWORD p)**

#### Description

Write bit map in *bmp* to a 24 bit, uncompressed, windows bitmap disk file

#### Parameters

*bmpfile* File name and path string  
*bmp* Address of variable with details of image to write.  
*p* If *p*=1, *bmp* is dynamic and is deleted.

### bmpread

**DWORD bmpread(char \*bmpfile,BMPF \*bmp)**

#### Description

Read a 24 bit, uncompressed, windows bitmap disk file

#### Parameters

*bmpfile* File name and path string  
*bmp* Address of variable with details and empty buffer for file to be read.

#### Result

*bmp->data[]* Array of 'COLOR' structures of the image

#### Returns

1, if successful, else 0.

### initcam

**DWORD initcam(HWND hWnd,CArtCamSdk \*cam,char \*pth)**

#### Description

Initialize camera with calls to class member functions: *cam->LoadLibrary()* and *cam->Initialize()*.

#### Parameters

*pth* Path and file name of camera dll: 'ArtCamSdk\_300MI.dll'  
*cam* Pointer to CArtCamSdk class object, of camera functions.  
*hWnd* Handle of application window.

# mapcord

**POINTF mapcord(double \*c,POINTF pi)**

## Description

Calculate a point in a transformed image from a point in the original and the coefficients calculated by mapcoef(). The image may be transformed by skewing, stretching, rotating, mirroring and shifting.

## Parameters

*c* Pointer to a 1D array of the eight image transformation coefficients calculated by mapcoef()

*pi* A point on the original image to be transformed. See CadFunc.doc for definition of POINTF.

## Returns

A point on the transformed image.

# mapcoef

**DWORD mapcoef(double \*fxy,double \*crn,double \*cof)**

## Description

Calculates coefficients used to transform an image by skew correction to a rectangle, stretching, rotating, mirroring and shifting to the origin. Used by mapcord() to calculate points in the transformed image from points in the original.

## Parameters

*fxy* Top Right hand point in the corrected image.

*crn* Tie points on the image to be corrected.

*cof* Pointer to an array of eight elements to store the results of calculation

## Returns

0 -A unique solution was not found or a divide overflow resulted.

1 -The function executed without error.

## Processing

Tie points are assigned as follows: -

$TL(x1,y1)=(crn[0],crn[1])$        $TR(x2,y2)=(crn[2],crn[3])$   
 $BR(x3,y3)=(crn[4],crn[5])$        $BL(x4,y4)=(crn[6],crn[7])$

Map points are assigned to ensure that the image is corrected to a rectangle at the origin: -

$TR'(x2',y2')=(fxy(0),fxy(1))$        $TL'(x1',y1')=(0,TR'(y2'))$   
 $BL'(x4',y4')=(0,0)$        $BR'(x3',y3')=(TR'(x2'),0)$

The Tie and Map points are converted to 2 sets of simultaneous equations one for their x co-ordinates and the other for their y co-ordinates. The equations are solved for the coefficients c1 to c8 and saved in the *<cof[]>* input array.

## Equation Set(1)

$(x1)c1 + (y1)c2 + (x1*y1)c3 + (1)c4 = x1'$   
 $(x2)c1 + (y2)c2 + (x2*y2)c3 + (1)c4 = x2'$   
 $(x3)c1 + (y3)c2 + (x3*y3)c3 + (1)c4 = x3'$   
 $(x4)c1 + (y4)c2 + (x4*y4)c3 + (1)c4 = x4'$

## Equation Set(2)

$(x1)c5 + (y1)c6 + (x1*y1)c7 + (1)c8 = y1'$   
 $(x2)c5 + (y2)c6 + (x2*y2)c7 + (1)c8 = y2'$   
 $(x3)c5 + (y3)c6 + (x3*y3)c7 + (1)c8 = y3'$   
 $(x4)c5 + (y4)c6 + (x4*y4)c7 + (1)c8 = y4'$

## When the real world is grabbed by camera

the bitmap(**Input Image**) of height(ht) and width(wd) is in **Spatial** co-ordinates and since GetImage() scan this image from the bottom left-hand corner, it appears in the **Image Array**, vertically flipped.

The Tie points in the **Input Image** seen in MS-Paint: tl, tr, br and bl need to be converted to Cartesian co-ordinates and passed through the *<crn[]>* input to TL, TR, BR and BL: -

$crn[0]=tl(X1)$        $crn[1]=ht-tl(Y1) =>$       TL  
 $crn[2]=tr(X2)$        $crn[3]=ht-tr(Y2) =>$       TR  
 $crn[4]=br(X3)$        $crn[5]=ht-br(Y3) =>$       BR  
 $crn[6]=bl(X4)$        $crn[7]=ht-bl(Y4) =>$       BL

## When the real world an AutoCad drawing

Co-ordinates from the real world drawing in cartesian xo-ordinates are passed to the *<crn[]>* input: -

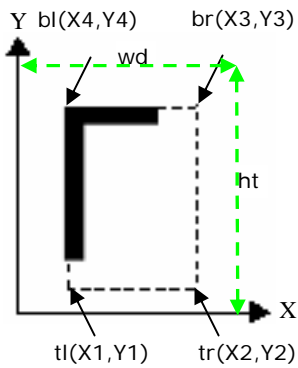
$crn[0]=tl(X1)$        $crn[1]=tl(Y1) =>$       TL  
 $crn[2]=tr(X2)$        $crn[3]=tr(Y2) =>$       TR  
 $crn[4]=br(X3)$        $crn[5]=br(Y3) =>$       BR

crn[6]=bl(x4) crn[7]=bl(y4) => BL

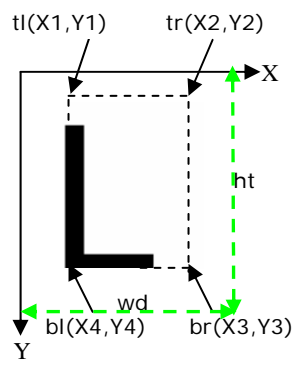
**Examples**

**1>Flip Vertical/No Rotation**

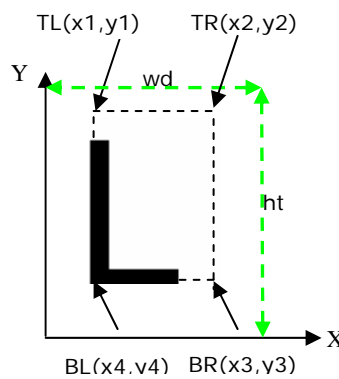
**Real World(Cartesian)**



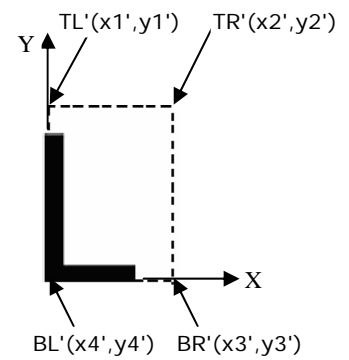
**Input Image(Spatial)**



**Image Array**

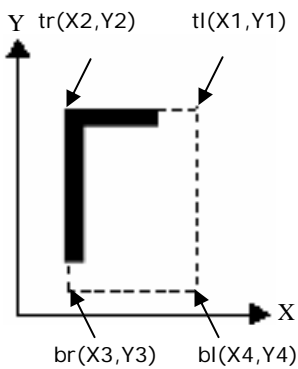


**Corrected Image**

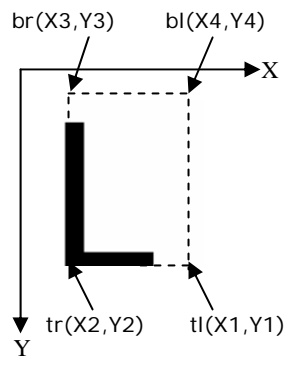


**2>Flip Vertical/Rotate 180° CCW**

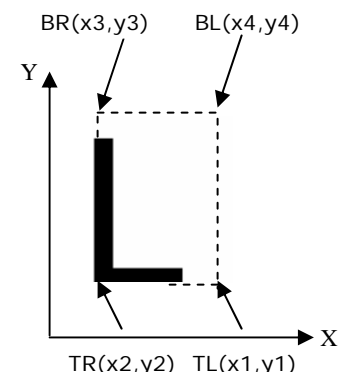
**Real World(Cartesian)**



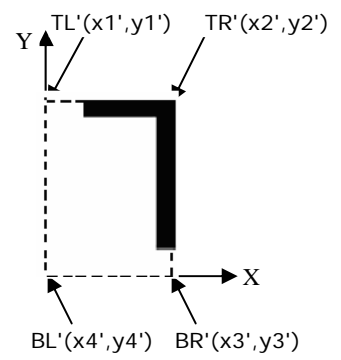
**Input Image(Spatial)**



**Image Array**

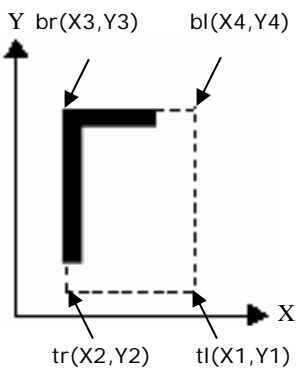


**Corrected Image**

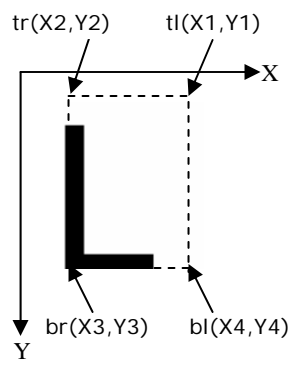


**3>No Flip/Rotate 180° CCW(used for Robocup setup at SAG)**

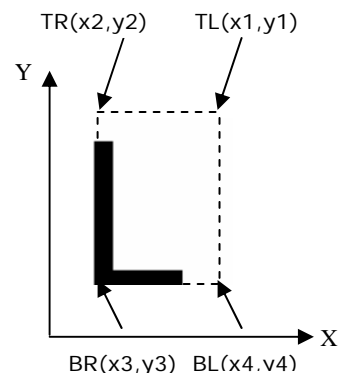
**Real World(Cartesian)**



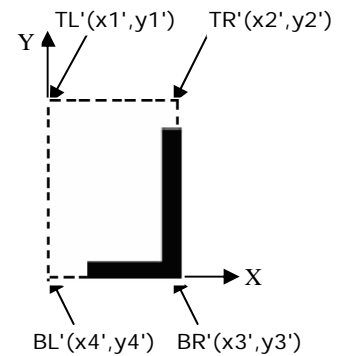
**Input Image(Spatial)**



**Image Array**

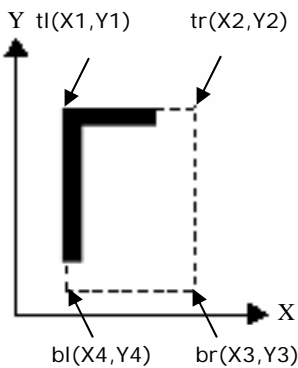


**Corrected Image**

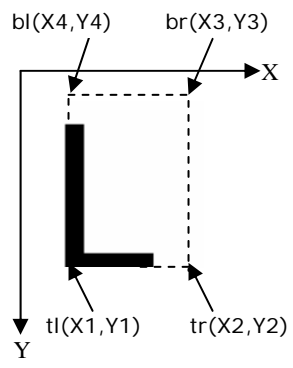


**4>No Flip/No Rotate**

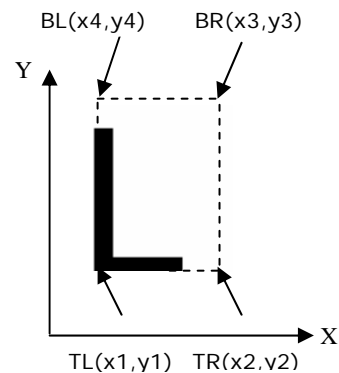
**Real World(Cartesian)**



**Input Image(Spatial)**



**Image Array**



**Corrected Image**

