Proyecto Fin de Carrera

DESARROLLO DE UN SISTEMA SEÑALADOR PARA PRESENTACIONES

Una aproximación a la integración del computador en el entorno de trabajo real

DISCA - 27B/03

D. David Millán Escrivá

Dirigido por: D. Manuel Agustí i Melchor

Índice

1 Introducción....3
1.1 Problema....3
1.2 Contexto....3
2 Planteamiento y Materiales....4
2.1 Planteamiento....4
2.2 Una primera aproximación....4
2.3 Periféricos....6
3 OpenCV....7
3.1 Instalación....7
3.2 Estructura de OpenCV....13
3.3 Ejemplos....14
3.3.1 Introducción a la interfaz....14
3.3.2 Mostrar imágenes....16
3.3.3 Dibujar con OpenCV....18
3.3.4 CallBacks del ratón en OpenCV....21
3.3.5 Webcam en OpenCV....24
4 Desarrollo y aplicación....27
4.1 Tarea Visión por Computador....28
4.1.1 Calibración....28
4.1.2 Seguimiento y detección....36
4.2 Eventos....41
4.3 Valoraciones....43
4.3.1 Errores: Resolución Vs Error Humano....44
4.3.2 Rendimiento....45
4.4 Experiencias....46
4.4.1 Windows....46
4.4.2 GNU/Linux....49
5 Conclusión....50
6 Bibliografía....51
7 Enlaces....52

1 Introducción

La metáfora del escritorio, desde que se inventó, ha sufrido una metamorfosis hasta llegar a las modernas interfaces gráficas. Sin embargo, las ideas básicas de manipulación directa de objetos, con el teclado y el ratón no han cambiado. El propósito general de este proyecto es poder ampliar el ámbito cotidiano de interacción con el escritorio, que imponen los tradicionales teclados y ratones, mediante el uso de un puntero láser y una cámara Web.

1.1 Problema

Se propone entonces, un mecanismo que extienda la labor del dispositivo apuntador tradicional (el ratón), fuera de la metáfora del escritorio.

Como propósito del proyecto, éste ha de ser multiplataforma (Windows y GNU/GNU/Linux), lo que nos plantea que librerías y lenguaje de programación utilizaremos.

También ha de ser una solución de bajo coste computacional y económico, al mismo tiempo que ha de ser no intrusivo y no competitivo.

1.2 Contexto

El contexto en el que vamos ha trabajar es el entorno próximo y por ello, emplearemos el aula como escenario de trabajo.

El proyecto trata de investigar la forma más óptima de interactuar, de forma que resulte intuitivo su uso. Para ello, se ha decido emular las funciones del ratón tradicional, mediante un puntero láser. Entonces se propondrá utilizar técnicas de Visión por Computador (en adelante VxC) como el de seguimiento de un patrón de luz para transformarlo en información de interacción con nuestro computador. Para ello, se va ha trabajar en un entorno próximo que es una pantalla de proyección, y se propondrá a partir de este trabajo el estudio de un entorno inmediato, como puede ser la mesa.

2 Planteamiento y Materiales

Este capítulo describe la evolución de las pruebas realizadas hasta definir la solución propuesta. En los siguientes apartados se describirán los planteamientos, primeras aproximaciones, periféricos, librerías y aplicaciones utilizadas hasta llegar a la solución deseada.

2.1 Planteamiento

Tras el problema formulado en el punto 1.1, donde se plantea la extensión de la labor del dispositivo apuntador (ratón) fuera del escritorio tradicional, se propone el desarrollo de una aplicación de seguimiento de una señal luminosa sobre algún tipo de soporte. Para ello, se implementarán algoritmos de segmentación de la imagen y calibración de la cámara, entre otros, mediante técnicas de VxC.

2.2 Una primera aproximación.

Una vez planteado el problema ha resolver se decidió realizar unas pruebas previas para determinar las necesidades y problemas que nos podríamos encontrar. Así pues, en primer lugar, se realizó una primera aproximación para dicho fin.

Esta aproximación se realizó en el sistema operativo Windows y con la plataforma de desarrollo de Microsoft Visual y el lenguaje Visual Basic para un rápido prototipo.

Para el desarrollo de esta primera versión se había planteado el reconocimiento del patrón de luz emitido por un diodo led rojo. Por ello, se creo un nuevo hardware (fig. 1.3) para sustentar y dar interacción al diodo. Dada la utilización de cables para la alimentación de dicho dispositivo, así como para una posibilidad de enviar señales al ordenador, nos encontramos con una desventaja principal que definimos como requisito, es un dispositivo intrusivo.

Aun Así, realizamos varias pruebas con dicho dispositivo y nos encontramos con un problema mayor, la poca luminancia que emite, y que sumado al entorno lo hace casi irreconocible.

Debido a la poca luminosidad del diodo led se pensó en utilizar otro tipo de emisor de luz, como puede ser un diodo led de alta luminosidad o un puntero láser. Debido a que el diodo de alta luminosidad también se ve afectado por el entorno, su alcance máximo es de unos 2 o 3 metros dependiendo del led y su pequeña circuiteria, y como la necesidad, también, del diseño de un nuevo hardware para sustentar dicho led, se pensó en un puntero láser (fig. 1.1). Éste, es más común de encontrar y todos los mandos de proyectores actualmente incorporan uno, además, se

ve poco afectado por el entorno de un aula (luz de las ventanas, luces del aula, etc...). Esta afirmación se explicará más adelante de forma física.

Fig. 1.1

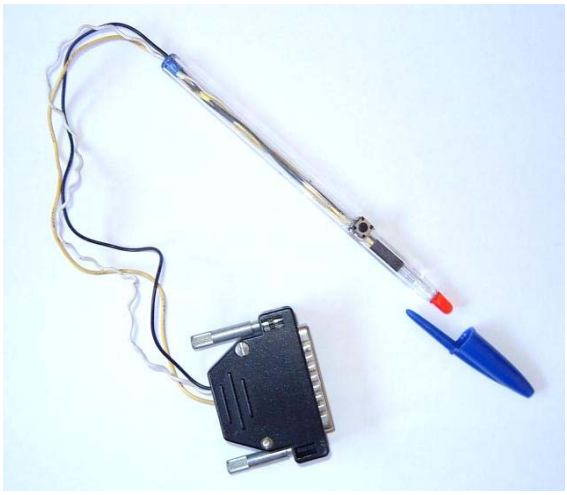


Fig. 1.2

Las primeras pruebas se realizaron sobre Visual Basic y un puntero láser de categoría II, sobre una superficie blanca y homogénea. El propósito de esta versión era el asegurarnos empíricamente que el patrón de luz era reconocido, así como también observar el tiempo de respuesta.

Así pues, se utilizó un componente Windows, ActiveX (VideoCapt), para realizar la captura de la imagen mediante la webcam y algoritmos de recorrido de imagen para detectar el punto en cuestión, que se explicarán con más detalle en el apartado 4.1 Tarea VxC.

Los problemas con que nos encontramos en este punto es que el proceso en Visual Basic era muy lento, y no realizaba las comprobaciones dentro de los tiempos estimados. Solamente se detectaba en partes cercanas al comienzo de la detección, y cuando más alejado el punto a detectar del comienzo del rastreo no era detectado porque se comenzaba el rastreo de una nueva imagen capturada. Había que tener mayor control de la webcam para realizar pausas de captura durante el proceso de detección, porque la captura no estaba sincronizada con el proceso de detección.

Una vez vistos los problemas encontrados en visual Basic optamos por utilizar librerías gráficas más propias del desarrollo de tareas de VxC. Nuestra elección ha sido OpenCV, una librería de código abierto y libre, que además nos ofrecía la posibilidad de trabajar en diversas plataformas como es el caso de Windows y GNU/Linux e incluso Mac, con lo que ampliamos el área de trabajo, haciendo hincapié en Windows y GNU/Linux.

También implementamos nuestra aplicación en C para que el código sea más portable de unas plataformas a otra, siendo las modificaciones mínimas entre una y otra, casi totalmente de interfaz gráfica.

2.3 Periféricos

Los perifericos utilizados en este proyecto son los mostrados en la fig. 1.3:

- Ordenador. En este caso se ha desarrollado el proyecto sobre un portatil Toshiba Sateillite 2450 con un procesador de 2,6 Ghz, 256 Mb de memoria, una tarjeta grafica Nvidia Geforce 420 go de 32 Mb y un disco duro de 30 Gb.
- Webcam. Se han probado en dos webcam distintas (webcam Toucam de Philips y Creative Labs Pro), sin embargo, por las características y propiedades de la webcam Toucam de Philips la he considerado más apropiada para llevar a cabo el desarrollo de este proyecto.
- Proyector. Se han utilizado los proyectores de las aulas de la escuela de informática, en distintas clases y diferentes condiciones del entorno.
- Puntero láser. Se ha utilizado un puntero láser LS-11N, estándar en todos los mandos de proyección.
- Pantalla de proyección. Aun no siendo un periférico lo nombro por ser una de las partes del sistema completo.

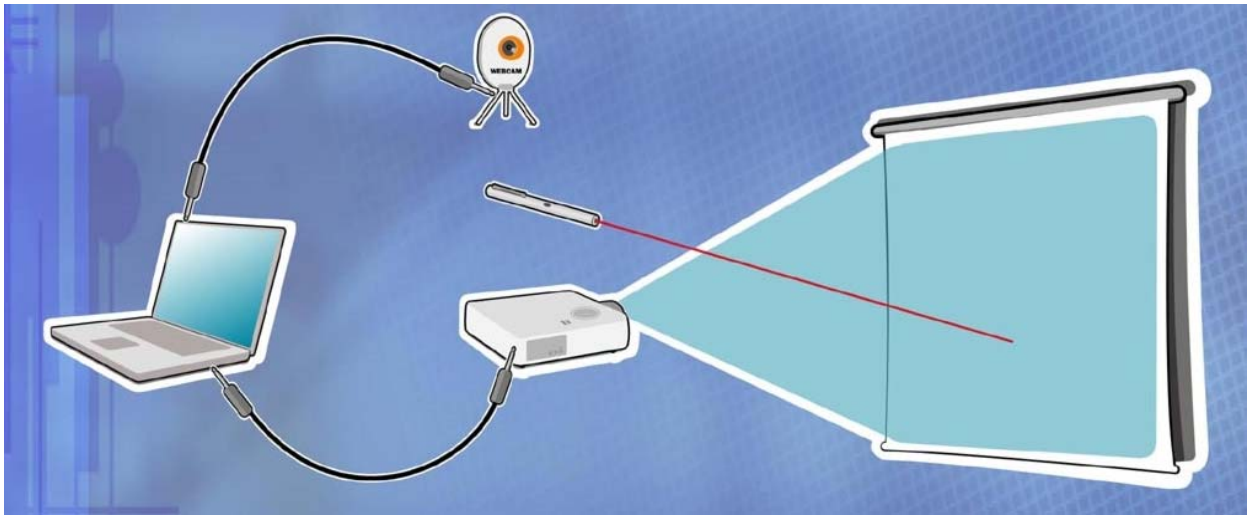


fig. 1.3

3 OpenCV

La principal librería utilizada en este proyecto es OpenCV (Open Source Computer Vision Library).

OpenCV es una librería GPL, orientada a la computación visual en tiempo real, utilizada principalmente en el campo de la IPO o HCI (Interacción persona ordenador, Human Computer Interaction).

Implementa una gran variedad de herramientas para la interpretación de las imágenes, es una libreria de alto nivel con algoritmos para técnicas de calibración, detección de objetos, seguimiento, análisis de contornos, reconstrucción 3d, con más de 300 funciones.

Una de las caracteristicas más importantes de OpenCV es que las funciones estan totalmente optimizadas para los procesadores de arquitectura Intel, y particularmente para la tecnoligia MMX™, Pentium® Pro, Pentium® III y Pentium® 4.

El software OpenCV trabaja en computadoras personales que están basadas en la arquitectura Intel y es totalmente integrable en aplicaciones escritas en C y C++. La versión más estable funciona sobre sistemas operativos Windows y están escritas en Ansi C. Aunque existen versiones de OpenCV para diferentes arquitecturas de procesadores y para diferentes sistemas operativos. En nuestro caso se ha hecho uso en procesadores intel Pentium® 4 y sobre plataformas Windows y GNU/Linux.

3.1 Instalación

La instalación bajo Windows se realiza de forma muy simple ya que OpenCV nos ofrece un instalador típico de Windows (fig. 3.1-1).

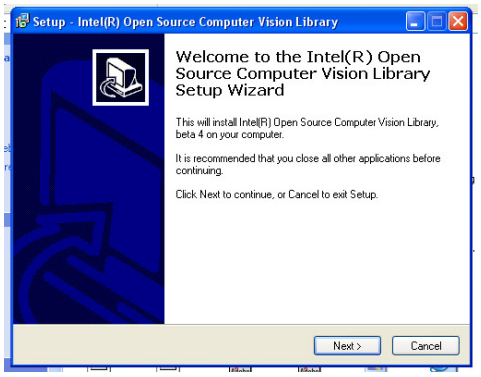


fig. 3.1-1

Lo único que tenemos que realizar es la configuración de nuestro entorno de trabajo para poder desarrollar bajo estas librerías. En este proyecto, el desarrollo en Windows se ha realizado bajo el entorno de desarrollo Visual Studio v6 y Visual .Net en el lenguaje C++, debido a la similitud entre ellos dos explicaremos solamente la configuración en .Net.

Así pues, lo único que tendremos que hacer, es decir a la plataforma de desarrollo (.Net o Visual Studio), donde se encuentran las librerías y las cabeceras de OpenCV, para ello vamos al menu: Herramientas > Opciones (fig: 3.1.1).

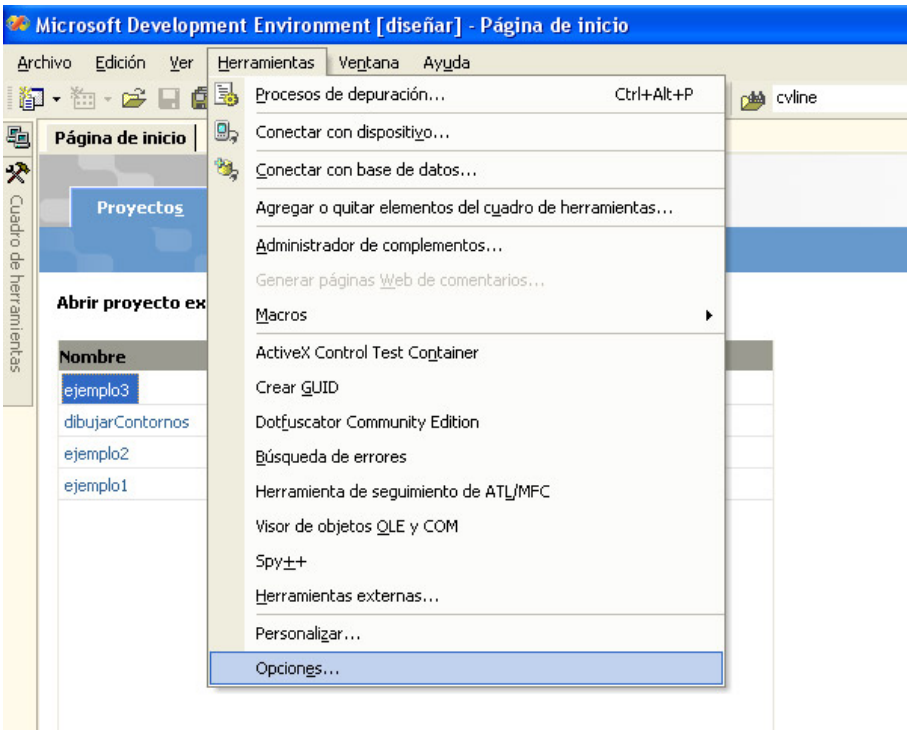


fig 3.1.1

Y seleccionando en el árbol del cuadro de diálogo de Opciones, Proyectos > Directorios de Visual C++, incluimos los siguientes directorios para los archivos de inclusión (fig 3.1.2):

- rutaOpenCV\cxcore\include
- rutaOpenCV\cvaux\include
- rutaOpenCV\cv\include
- rutaOpenCV\otherlibs\highgui
- rutaOpenCV\otherlibs\cvcam\include

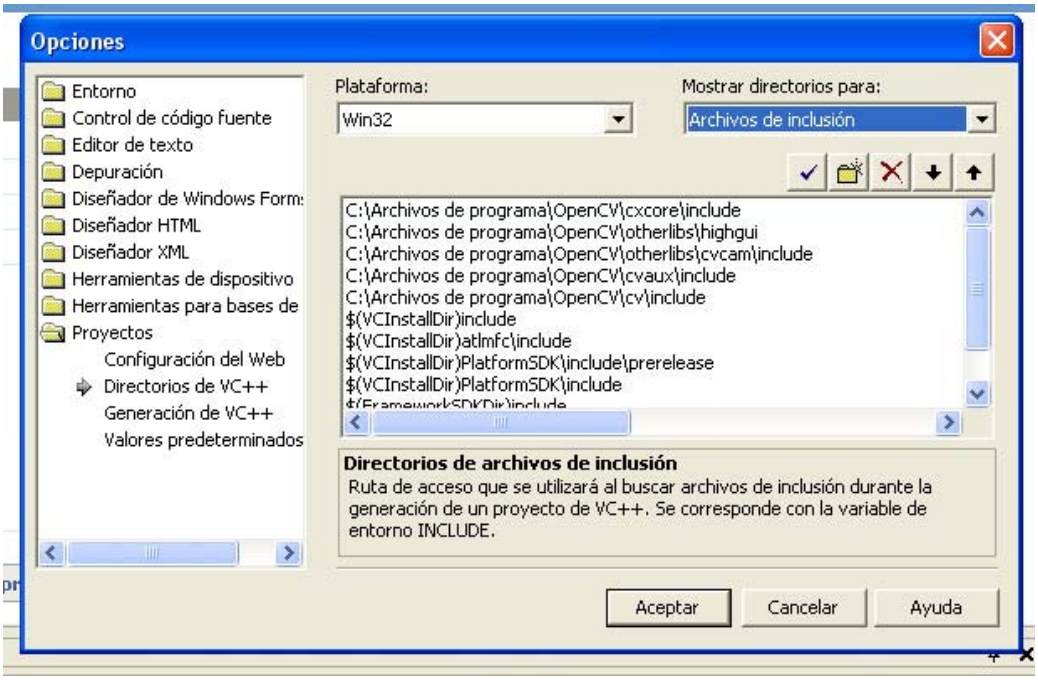


fig 3.1.2

Y en el mismo lugar del árbol, seleccionando “Archivos de biblioteca”, incluimos la ruta rutaOpenCV\lib\ fig 3.1.3

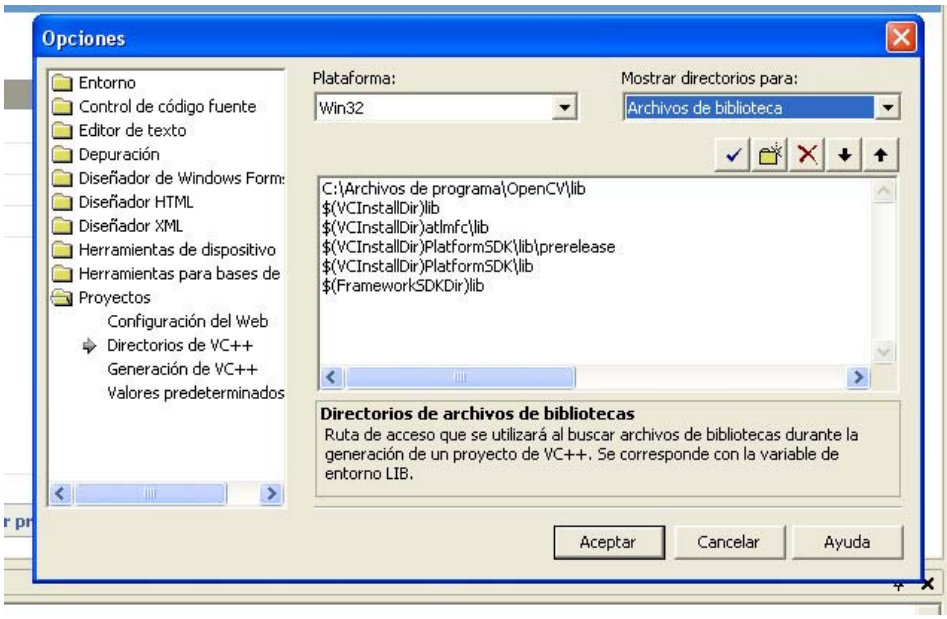


fig 3.1.3

De esta forma ya estariamos preparados para crear un nuevo proyecto en Visual Studio o en Visual .Net, en el lenguaje C++.

Para crear un nuevo proyecto en Visual.Net deberemos crear un proyecto de Visual C+ > Win32, seleccionando un proyecto de consola o un proyecto Win32 (fig 3.1.4), e incluso

podriamos crear un proyecto VisualC++ > MFC.

En nuestros ejemplos hemos utilizado proyectos de consola, ya que de esta forma el código es similar al de GNU/Linux.

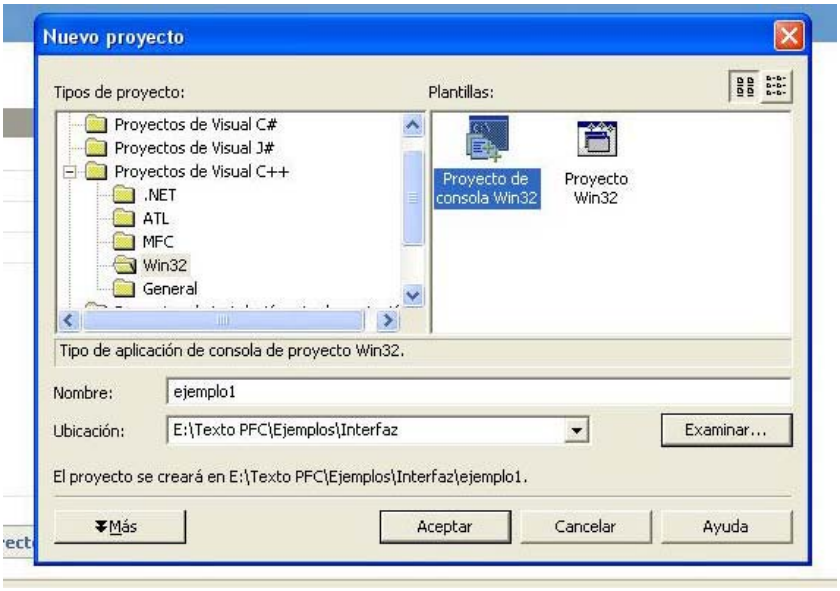


fig 3.1.4

Una vez creado el proyecto tenemos que configurarlo para que conozca que vamos ha utilizar las librerías de OpenCV, así pues, en Proyecto > propuedades, en el árbol seleccionamos Vinculador > Enlace e incluimos las siguientes dependencias adicionales (fig 3.1.5):

cv.lib
highgui.lib
cvcam.lib
cvcore.lib
cvaux.lib

En realidad no en todos los proyectos necesitaremos todas las dependencias, es decir, en el ejemplo 1 de más adelante sobraría con la highgui.lib, pero para nuestro proyecto final, las necesitaremos todas.

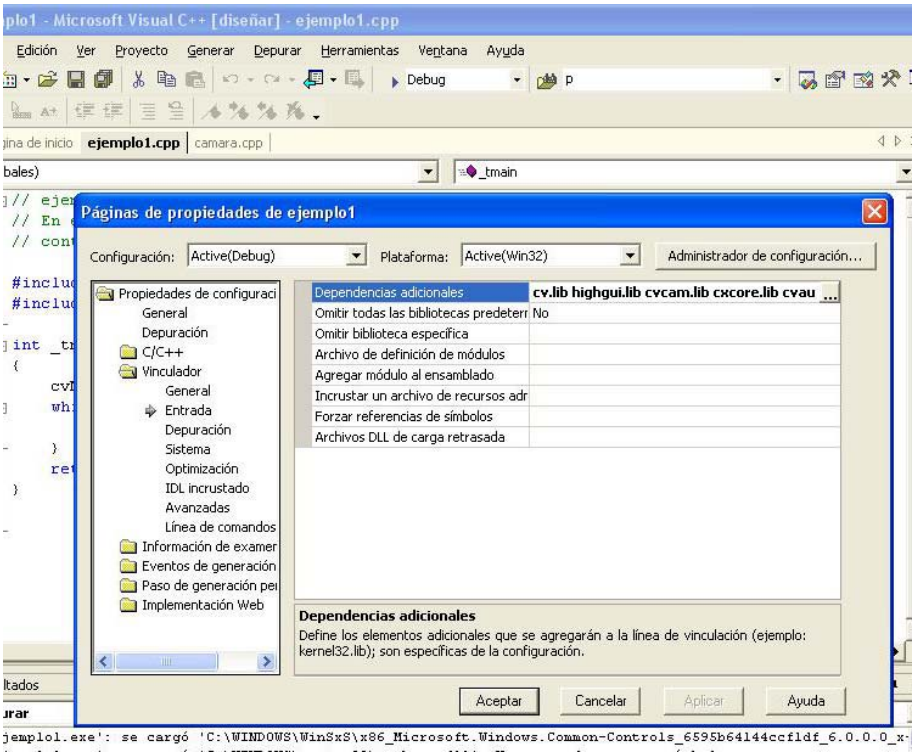


fig 3.1.5

La instalación bajo GNU/Linux es algo más compleja que la de Windows, y depende de la versión que estemos utilizando (Debian, Mandrake, Redhat, Fedora...). Durante el desarrollo del proyecto se ha probado en todas las mencionadas anteriormente. Debido al gran numero de paquetes que incluye Debian, nos resulta muy fácil su instalación, debido a que por defecto lo trae incluido, así pues con ayuda de alguna aplicación de instalación de paquetes de Debian, mediante aptitude (fig 3.1.6) en modo consola, o mediante Kpackage (fig 3.1.7) en sistema de ventanas KDE.

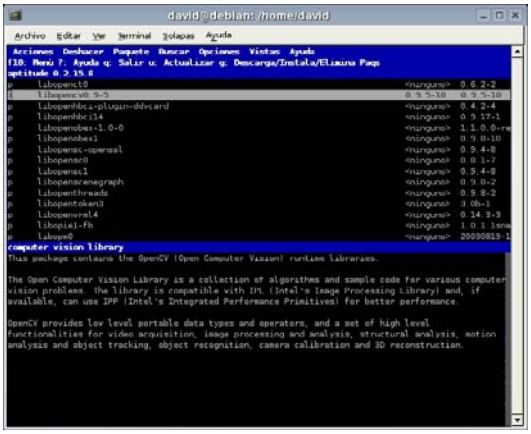


Fig 3.1.6

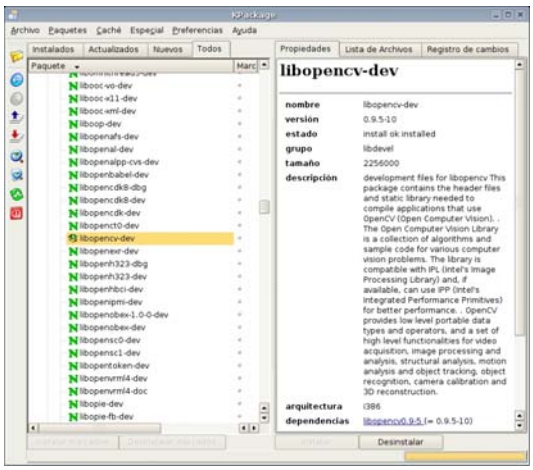


Fig 3.1.7

Aptitude o Kpackage, nos ayudarán el la instalación detectando las dependencias que OpenCV tiene, e instalandolas por nosostros si no las tenemos instaladas, de esta manera nos solucionará y ayudará en gran manera su instalación.

Si deseamos instalar una versión distinta a la que nos ofrece Debian, OpenCV 0.9.5, como por ejemplo al 0.9.6, o una anterior lo tendremos que hacer de la forma natural si no conseguimos el paquete Debian, esta forma nos servirá tambien para cualquier otra versión de GNU/Linux, como por ejemplo la Mandrake.

Luego, una vez satisfechas todas las dependencias de OpenCV: GTK+ 2.x, lesstif2, libjpeg, libpng, libtiff, v4l... Solamente tendremos que descomprimir el archivo de OpenCV, configurarlo, compilarlo e instalarlo:

```
david@debian:~/temp$ tar -zxvf opencv-0.9.6.tar.gz
...
david@debian:~/temp$ cd opencv-0.9.6
david@debian:~/temp/opencv-0.9.6$ ls -l
...
david@debian:~/temp/opencv-0.9.6$ ./configure --prefix=/home/opencv
...
david@debian:~/temp/opencv-0.9.6$ make
....
david@debian:~/temp/opencv-0.9.6$ su
...
root@debian:~/temp/opencv-0.9.6# makeinstall
```

En caso de instalar la versión 0.9.6 bastará con crear una variable de entorno para poder compilar nuestro codigo.

```
root@debian:~/temp/opencv-0.9.6#PKG_CONFIG_PATH=/home/opencv/libs/pkgconfig/
root@debian:~/temp/opencv-0.9.6#export PKG_CONFIG_PATH
```

Para versiones anteriores hay que añadir la linea “/home/opencv/lib” en el archivo “/etc/ld.so.conf” y ejecutar la instrucción “ldconfig -v”.

Una vez realizado esto, para compilar nuestro codigo solo tendremos que ejecutar la siguiente sentencia:

gcc -I/home/opencv/include -L/home/opencv/lib -lopencv -lhighgui -lcvcam -lstdc++ archivo.c -o archivo.exe

Para facilitar la compilación se ha creado un shell script llamado comp que para Debian es:

```
#!/bin/sh
#Compilacion de programas opencv
echo @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
echo @@@@ David Milllan Escriva @@@@
echo @@  Compilacion programas de @@
echo @@@@@@@@@@ OpenCV @@@@@@@@@@@@@@
echo @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

gcc -I/usr/include/opencv -lopencv -lhighgui -lcvcam $1 -o $2
```

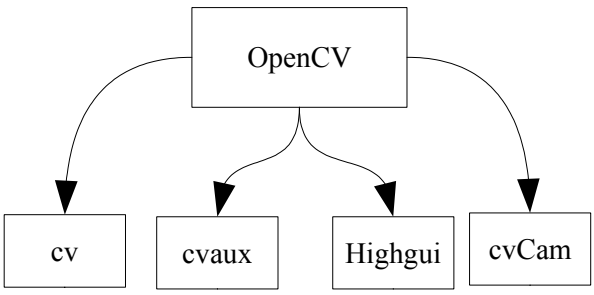
Que, para otro caso, simplemente tendríamos que cambiar la sentencia “gcc” por la mencionada anteriomente.

Luego para compilar un programa simplemente deberiamos ejecutar la sentencia:

./comp archivo.c archivo.exe

3.2 Estructura de OpenCV

OpenCV se puede dividir en varias partes bien diferenciadas para su uso, que estan a su vez divididas en cabeceras (*.h) y son las siguientes, cv, cvaux, highgui, cvcam.



De tal forma que cada parte esta especializada en una cosa en concreto. En las librerias cv están las funciones propias de VxC, como operaciones matriciales, histogramas, funciones especiales, calibración, etc. En cvaux están las funciones en fase de prueba o experimentales. En Highgui están las funciones de interfaz gráfica, y funciones de video y cámara. Y por último en las cvCam están las funciones más especificas de cámara que a partir de la versión 0.9.6 se eliminian para el sistema GNU/GNU/Linux, dejando solamente operativas para el trabajo con cámaras el modulo highgui.

3.3 Ejemplos

3.3.1 Introducción a la interfaz

En este ejemplo se describirá como crear una ventana en openCV así como controles asociados a ella.

Funciones utilizadas:

Creación de una ventana:

int cvNamedWindow(const char* name, unsigned long flags);

name: nombre de la ventana que será su identificador.

flags: Define las propiedades de la ventana. CV_WINDOW_AUTOSIZE o 1 para ajustar la ventana a la imagen mostrada o 0 en otro caso.

Cambiar el tamaño de una ventana:

void cvResizeWindow(const char* name, int width, int height);

name:Nombre de la ventana a cambiar el tamaño

width:Ancho de la ventana

height:Alto de la ventana

Cambiar la posición de la ventana:

void cvMoveWindow(const char* name, int x, int y);

Solamente versión 0.9.6

name:Nombre de la ventana que queremos mover

x: posición x

y: posición y

Debemos tener en cuenta que las cordenadas de la ventana tiene su punto de origen en la esquina superior izquerda de la pantalla.

Esperar el evento del teclado

int cvWaitKey(int delay CV_DEFAULT(0));

delay: valor entero que en caso de ser menor o igual a 0 será una espera infinita.

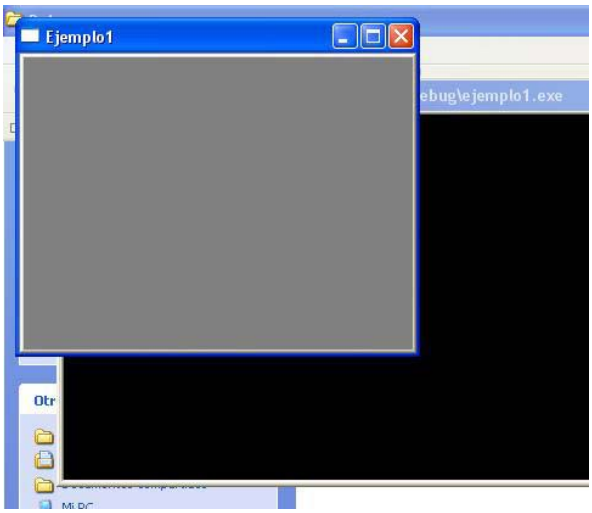
valor devuelto: el código de la tecla pulsado -1 si el tiempo se ha vencido.

Destruir la ventana.

void cvDestroyWindow(const char* name);

name:Nombre de la ventana que queremos destruir

```
// ejemplo1.cpp: define el punto de entrada de la aplicación de consola.
// En este ejemplo se describirá como crear una ventana en openCV asi como
// controles asociados a ella.
#include "highgui.h"
int main(int argc, char* argv[])
{
   //Creamos la ventana Ejemplo1, el segundo parametro
   // sirve para que la ventana se autoajuste a la imagen mostrada.
   cvNamedWindow("Ejemplo1",1);
   //Tamaño de la ventana (Nombre ventan,ancho,alto) Opcional
   cvResizeWindow("Ejemplo1",320,240);
   //Posicionamiento de la ventan (nombre,x,y) Opcional
   cvMoveWindow("Ejemplo1",10,10);
    for(;;)//Bucle infinito hasta que pulsen Esx o q o Q.
    {
                //Definimos una variable donde esperamos el evento de teclado
                int key;
                // esperamos un tiempo en milisegundos que se pasa como parametro
                //si lo ponemos a 0 es una espera indefinida
                key = cvWaitKey(0);
        if( key == 27 || key == 'q' || key == 'Q' ) // 'ESC', q o Q
            break;
   }
   cvDestroyWindow("Ejemplo1");//destruimos la ventana antes de salir.
   return 0;
}
```



3.3.2 Mostrar imágenes

En este ejemplo se describirá como crear una ventana en openCV así como mostrar una imagen

Funciones utilizadas:

Cargar un archivo de imagen:

IplImage* cvLoadImage(const char* filename, int iscolor CV_DEFAULT(1));

filename: nombre del archivo a cargar.

iscolor: si > 0 la imagen es cargada con 3 canales (RGB)
si = 0 la imagen será cargada con un canal, (escala de grises)
si < 0 la imagen es cargada según el archivo.

Los archivos soportados por OpenCV son: Windows bitmaps - BMP, DIB; JPEG files - JPEG, JPG, JPE; Portable Network Graphics - PNG; Portable image format - PBM, PGM, PPM; Sun rasters - SR, RAS; TIFF files - TIFF, TIF.

Valor devuelto: devuelve un puntero a una estructura IplImage, con la que OpenCV puede trabajar.

Mostrar una imagen:

void cvShowImage(const char* name, const CvArr* image);

name: nombre, identificador de la ventana donde mostrar la imagen

image: puntero a una estrurctura IplImage

Destruir todas las ventanas que se han creado:

void cvDestroyAllWindows();

Esta función destruye todas las ventanas que hemos creado.

```
// ejemplo2.cpp: define el punto de entrada de la aplicación de consola.
// En este ejemplo se describirá como crear una ventana en openCV asi como
// Mostrar una imagen
#include "highgui.h"
#define ventana "Ejemplo2-1"
#define ventana1 "Ejemplo2-2"

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

```
    printf("--------------------------------\n"
                "Ejemplo 2 - David Millan - Highgui\n"
                "--------------------------------\n"
                "Hot keys:\n"
                "   Esc,q,Q - Salir");
    //Comprobamos si han pasado una image como parametro, sino una por defecto
    char* filename = argc >= 2 ? argv[1] : (char*)"lena.jpg";
    //cargamos la imagen en RGB
    IplImage* imagen;
    //(archivo,int a) a>0:RGB; a=0:Escala Grises; a<0:la de la imagen
    imagen=cvLoadImage(filename,1);
    //Cargamos la imagen en Escala de grises
    IplImage* imagen1;
    //(archivo,int a) a>0:RGB; a=0:Escala Grises; a<0:la de la imagen
    imagen1=cvLoadImage(filename,0);
    cvNamedWindow(ventana,0);
    cvNamedWindow(ventana1,0);
    //Mostramos la imagen en la ventana
    cvShowImage(ventana,imagen);
    cvShowImage(ventana1,imagen1);
     for(;;)
    {
                int key;
                key = cvWaitKey(0);
        if( key == 27 || key == 'q' || key == 'Q' ) // 'ESC', q o Q
            break;
    }
    cvDestroyAllWindows();//Destruimos todas las ventanas
    return 0;
}
```

3.3.3 Dibujar con OpenCV

En este ejemplo se explicará las funciones más usuales a la hora de dibujar en OpenCV.

Funciones utilizadas:

Crear una imagen vacia:

IplImage* cvCreateImage(CvSize size, int depth, int channels);

size: Ancho y alto de la imagen

depth: Profundidad de color de la imagen. Puede ser:

IPL_DEPTH_8U - unsigned 8-bit integers

IPL_DEPTH_8S - signed 8-bit integers

IPL_DEPTH_16S - signed 16-bit integers

IPL_DEPTH_32S - signed 32-bit integers

IPL_DEPTH_32F - single precision floating-point numbers

IPL_DEPTH_64F - double precision floating-point numbers

channels: Numero de canales por pixel, puede ser 1,2,3 o 4.

Inicializar una nueva imagen:

void cvZero(CvArr* arr);

arr: Array a vaciar o inicializar a 0. Se utiliza también para inicializar una imagen toda a 0 (color negro).

Algunas funciones de dibujo de la versión 0.9.5:

void cvRectangle(CvArr* img, CvPoint pt1, CvPoint pt2, double color, int thickness=1);

img: Imagen donde dibujar el rectángulo.

pt1: Uno de los vértices del rectángulo.

pt2: El vértice contrario al vértice pt1.

color: Color de la línea.

thickness: Grosor de la linia, con un valor negativo rellenamos el rectangulo con el color que se epecifico para la línea.

void cvLine(CvArr* img, CvPoint pt1, CvPoint pt2, double color, int thickness=1, int connectivity=8);

img: Imagen donde dibujar un segmento.

pt1: Primer punto del segmento.

pt2: Segundo punto del segmento.

color: Color del segmento.

thickness: Grosor del segmento.

Connectivity: Algoritmo de conectividad de Bresenham utilizado para el dibujo de la línea, solo admite los valores 8 o 4, Si se pone el valor 0 se toma como el valor 8,

void cvCircle(CvArr* img, CvPoint center, int radius, double color, int thickness=1);

img: Imagen donde dibujar el círculo.

center: Punto del centro del círculo.

radius: Radio del círculo.

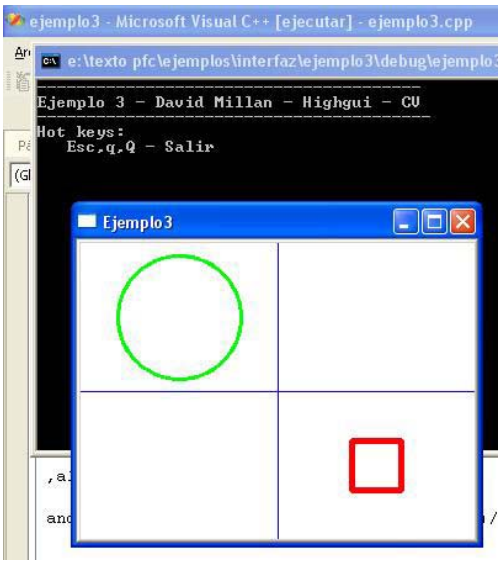
color: color del círculo.

thickness: Grosor de la línea, con un valor negativo rellenamos el circulo con el color que se epecifico para la linia.

```
// ejemplo3.cpp: define el punto de entrada de la aplicación de consola.
#include "cv.h"
#include "highgui.h"
#include "stdio.h"
#define ancho           320
#define      alto       240
#define ventana "Ejemplo3"
int main(int argc, char* argv[])
{
  printf("----------------------------------------\n"
              "Ejemplo 3 - David Millan - Highgui - CV\n"
              "----------------------------------------\n"
              "Hot keys:\n"
              "   Esc,q,Q - Salir\n\n");

  //Creamos una imagen vacia donde dibujar
  IplImage* imagen;
  imagen=cvCreateImage(cvSize(ancho,alto),8,3);
  cvZero(imagen);
  //Rectangulo
  cvRectangle( imagen, cvPoint(0,0), cvPoint(ancho,alto),CV_RGB(255,255,255),-1);
  //Linia
  cvLine(imagen,cvPoint(ancho/2,alto),cvPoint(ancho/2,0),CV_RGB(0,0,255),1,8);
  cvLine(imagen,cvPoint(0,alto/2),cvPoint(ancho,alto/2),CV_RGB(0,0,255),1,8);
  //Circulo
  cvCircle( imagen,cvPoint(ancho/4,alto/4), 50, CV_RGB(0,255,0),2 );
  //Rectangulo
  cvRectangle( imagen, cvPoint((3*ancho)/4-20,(3*alto)/4-20), cvPoint((3*ancho)/4+20,(3*alto)/4+20),CV_RGB(255,0,0),4);
```

```
    cvNamedWindow(ventana,0);
    cvResizeWindow(ventana,ancho,alto);
    //Mostramos la imagen en la ventana
    cvShowImage(ventana,imagen);
    for(;;)
    {   int key;
        key = cvWaitKey(0);
        if( key == 27 || key == 'q' || key == 'Q' ) // 'ESC', q o Q
            break;
    }
    cvDestroyAllWindows();//Destruimos todas las ventanas
    return 0;
}
```



3.3.4 CallBacks del ratón en OpenCV

En este ejemplo se explica la utilización de los eventos de ratón para la interacción con OpenCV.

Funciones utilizadas:

Capturar eventos del raton:

void cvSetMouseCallback(const char* window_name, CvMouseCallback on_mouse);

window_name: Nombre de la ventana de la que queremos capturar los eventos de ratón.

on_mouse: Nombre de la funcion que queremos que llame al ocurrir un evento de ratón.

La función on_mouse que definamos tiene que tener la siguiente estructura:

void on_mouse(int event, int x, int y, int flags)

donde:

- x e y son la posición en coordenadas de la imagen donde ha ocurrido el evento.
- event puede tomar cualquiera de los siguientes valores:

```
#define CV_EVENT_MOUSEMOVE      0
#define CV_EVENT_LBUTTONDOWN    1
#define CV_EVENT_RBUTTONDOWN    2
#define CV_EVENT_MBUTTONDOWN    3
#define CV_EVENT_LBUTTONUP      4
#define CV_EVENT_RBUTTONUP      5
#define CV_EVENT_MBUTTONUP      6
#define CV_EVENT_LBUTTONDBLCLK  7
#define CV_EVENT_RBUTTONDBLCLK  8
#define CV_EVENT_MBUTTONDBLCLK  9
```

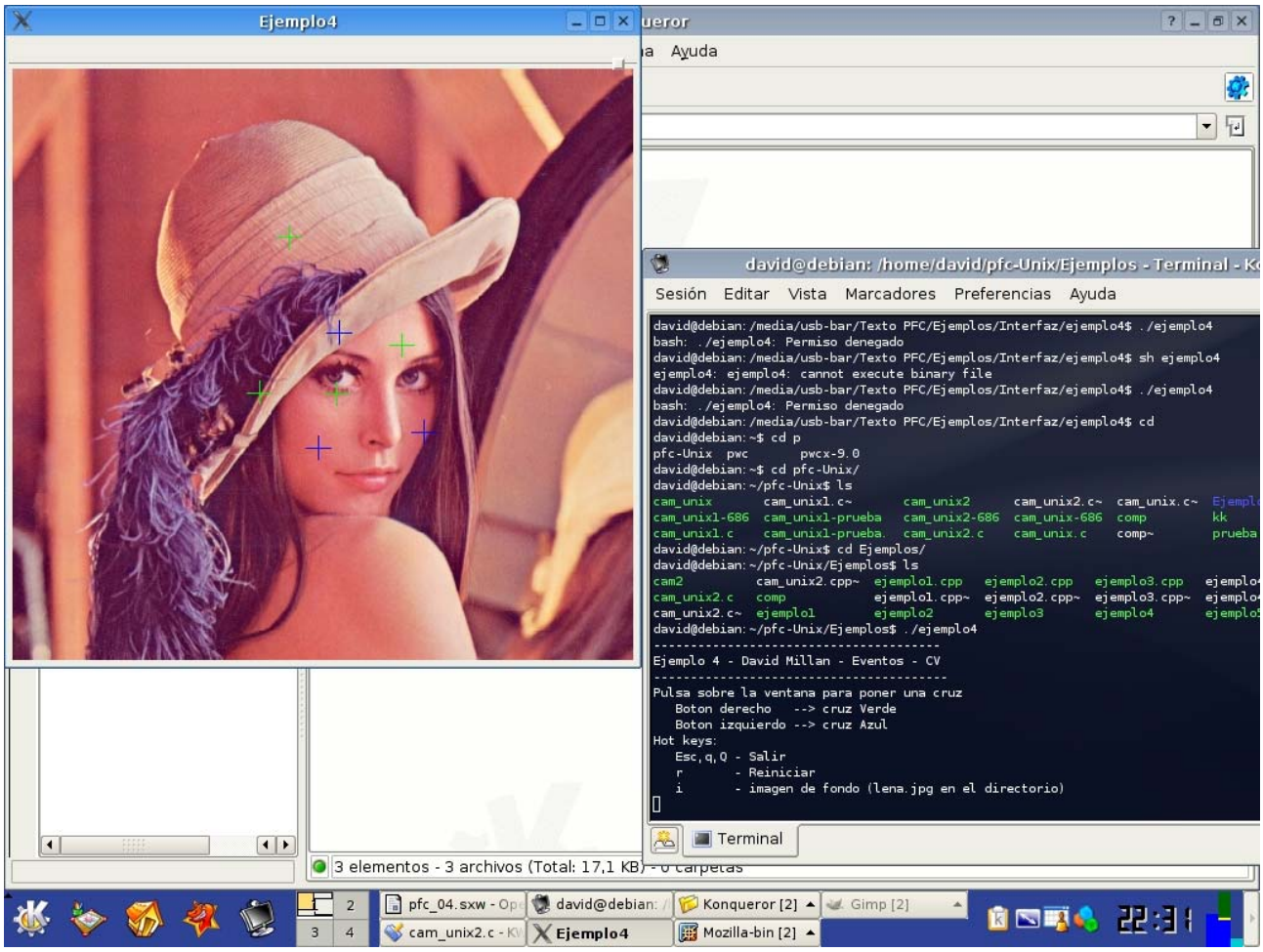
- y flags puede tomar los siguientes valores:

```
#define CV_EVENT_FLAG_LBUTTON   1
#define CV_EVENT_FLAG_RBUTTON   2
#define CV_EVENT_FLAG_MBUTTON   4
#define CV_EVENT_FLAG_CTRLKEY   8
#define CV_EVENT_FLAG_SHIFTKEY  16
#define CV_EVENT_FLAG_ALTKEY    32
```

```
#include "cv.h"
```

```
#include "highgui.h"
#include "stdio.h"
#define	ancho	512
#define	alto	512
#define ventana "Ejemplo4"
//Varialbes globales
IplImage* imagen;
//funcion de callback del ratón
void on_mouse(int event, int x, int y, int flags){
  switch(event){
  case CV_EVENT_LBUTTONUP://evento del boton izquerdo
  cvLine(imagen,cvPoint(x-10,y),cvPoint(x+10,y),CV_RGB(0,0,255),1,8);
  cvLine(imagen,cvPoint(x,y-10),cvPoint(x,y+10),CV_RGB(0,0,255),1,8);
  break;
  case CV_EVENT_RBUTTONUP://evento del boton derecho
                  //DIBUJAMOS OTRA CRUZ EN X Y EN VERDE
  break;
  }
}
//Funcion principal
int main(int argc, char* argv[])
{
        //Creamos una imagen vacia donde dibujar
  imagen=cvCreateImage(cvSize(ancho,alto),8,3);
  cvZero(imagen);//inicicalizamos
  cvRectangle( imagen, cvPoint(0,0), cvPoint(ancho,alto),CV_RGB(255,255,255),-1);//rellenanos de blanco
  cvNamedWindow(ventana,0);//creamos una ventana
  //DEFINIMOS EL EVENTO DEL RATON
  cvSetMouseCallback( ventana, on_mouse );
  //
  for(;;)
        {
        //mostramos la imagen en cada bucle de forma que refrescamos constantemente la visualizacion
        //Sino incluimos esta llamada en el bucle siempre veriamos la imagen fija.
        cvShowImage(ventana,imagen);
        int key;
        key = cvWaitKey(10);//esperamos el evento del teclado durante 10 ms
        if( key == 27 || key == 'q' || key == 'Q' ) // 'ESC', q o Q
                break;
        if(key=='r'){//reiniciamos la imagen
                cvZero(imagen);
                cvRectangle( imagen, cvPoint(0,0), cvPoint(ancho,alto),CV_RGB(255,255,255),-1);
        }
        if(key=='i'){//mostramos una imagen
                imagen=cvLoadImage("lena.jpg",1);
        }
  }
```

```
    cvDestroyAllWindows();//Destruimos todas las ventanas
    return 0;
}
```



3.3.5 Webcam en OpenCV

Funciones utilizadas:

En este ejemplo se utiliza la librería cvcam, que funciona correctamente en la versión 0.9.5, que es con la que se ha realizado el proyecto, pero al ser todavia una versión Beta en la versión 0.9.6 han decidido suprimirla bajo GNU/Linux, ya que intentan unificar todas las funciones de cámaras, vídeo e interfaz bajo la librería Highgui.

cvCam hace uso de las X11 en donde renderiza las imagenes. Las funciones de Xlib no las vamos a explicar ya que se escapan del ámbito de este proyecto. En la parte de bibliografia y enlaces podréis encontrar enlaces a tutoriales y artículos donde profundizan en las Xlib.

int cvcamGetCamerasCount( );

Devuelve el numero de camaras conectadas.

cvcamGetProperty(int cameraindex, const char* property, void* value);

cvcamSetProperty(int cameraindex, const char* property, void* value);

Nos devuelve o asignamos propiedades de la cámara.

Propiedad	Accion	Argumentos	Leer(G)/ Escribir(S)	Windows/ GNU/Linux
CVCAM_PROP_ENABLE	Selecciona o deseslcciona una cámara	Para asignar: CVCAMTRUE/ CVCAMFALSE Para preguntar: puntero a un entero o a CvCapture	SG	WL
CVCAM_PROP_RENDER	Renderiza la cámara	CVCAMTRUE/ CVCAMFALSE For Set, pointer to int for Get	SG	WL
CVCAM_PROP_WINDOW	Selecciona una ventana para renderizar en ella la cámara	Puntero a HWND(Win) o puntero a XWindow(Lin)	SG	WL
CVCAM_PROP_CALLBACK	Asigna la función de callback que renderizará cada frame	void (*callback)(IplImage* image)	S	WL
CVCAM_DESCRIPTION	Recoge el nombre y alguna informacion más sobre la cámara.	Devuelve un puntero a una estructura CameraDescription Sobre windows solo el campo DeviceDescription es activo.	G	WL
CVCAM_VIDEOFORMAT	Muestra la caja de dialogo del formato de la cámara	NULL	G	WL
CVCAM_CAMERAPROPS	Muestra la caja de dialogo de propiedades de la cámara	NULL	G	WL
CVCAM_RNDWIDTH	Asigna el ancho de la salida del video	Puntero a un entero que contiene el ancho	SG	WL

CVCAM_RNDHEIGHT	Asigna el alto de la salida del video	Puntero a un entero que contiene el alto	SG	WL
CVCAM_STEREO_CALLBACK	Asigna la función de callbakc que será llamada cada dos frames sincronos para dos camaras	Void (*callback)(IplImage* Image1, IplImage* image2)	S	W
CVCAM_PROP_RAW	Recoge el último frame	IplImage**	G	L
CVCAM_PROP_SETFORMAT	Asigna el formato de video sin ventanas	Puntero a una estructura videoformat.Para asignar se debe inicializar con los valores deseados.	SG	L

int cvcamInit( );

Inicializa la cámara una vez asignadas todas las propiedades. Devuelve 0 en caso positivo y un valor negativo en caso de error

int cvcamStart( );

Comienza la captura de vídeo para todas las camaras habilitadas. Devuelve 0 en caso positivo y un valor negativo en caso de error

int cvcamStop( );

Para la captura de vídeo. Esta funcion siempre devuelve 0

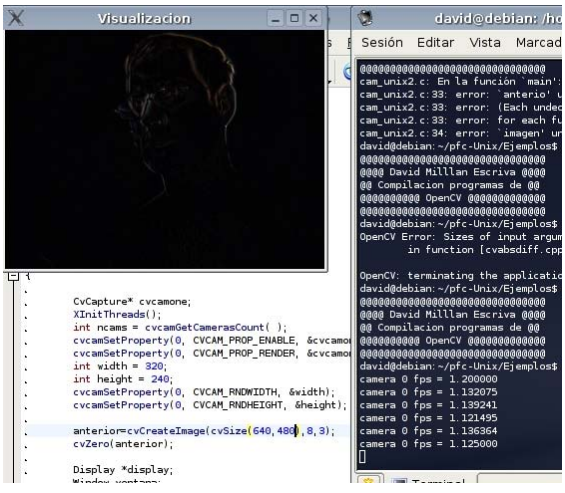
int cvcamExit( );

Libera todos los recursos utilizados por cvcam. Devuelve siempre 0

```
#include "cv.h"
#include "cvcam.h"
#include "highgui.h"
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>
#define ancho 640
#define alto 480
IplImage* anterior;
//Funcion de callback de la cámara donde realizamos los procesos de las imagenes
void callback(IplImage* image)
{
  ...
}

int main()
{
  XInitThreads();
  CvCapture* cvcamone;
  int ncams = cvcamGetCamerasCount( );
  if(ncams<1){
        printf("Error: No se ha detectado ninguna cámara");
```

```
            exit(-1);
    }
    cvcamSetProperty(0, CVCAM_PROP_ENABLE, &cvcamone);
    cvcamSetProperty(0, CVCAM_PROP_RENDER, &cvcamone);
    ...
    //FUNCIONES DE X11
    Display *display;
    Window ventana;
     /* abrimos el display estándar */
    display = XOpenDisplay( NULL );
     /* creamos una nueva ventana */
    ventana = XCreateSimpleWindow( display, RootWindow( display, 0), 10, 50, ancho, alto, 2, 0, 1 );
     /* Definimos las propiedades la ventana */
    XSetStandardProperties(display, ventana, "Visualizacion","PFC",None,NULL,0,NULL);
     /* Mapeamos la ventana para hacerla visible */
    XMapWindow( display, ventana );
    XFlush( display );
    //Aplicamos la camara ala ventana donde se renderizará
    cvcamSetProperty(0, CVCAM_PROP_WINDOW, &ventana);
    //Asignamos la cámara a una función de callback
    cvcamSetProperty(0, CVCAM_PROP_CALLBACK, callback);
    //inicializamos la cámara
    cvcamInit( );
    //Activamos la cámara
    cvcamStart( );
    //Bloquelamos la aplicación para capturar eventos etc...
    for(;;){
    //eventos de las X
    }
    //Antes de salir cerramos todo, paramos la cámara, y la destruimos.
    cvcamStop( );
    cvcamExit( );
    return 0;
}
```



4 Desarrollo y aplicación.

Llegados a este punto nos tenemos que plantear cómo vamos ha abordar el problema. Para ello, hacemos uso de un diagrama de estados (fig. 4), el cual nos mostrará de forma muy clara el proceso que vamos ha seguir para el desarrollo de la aplicación.

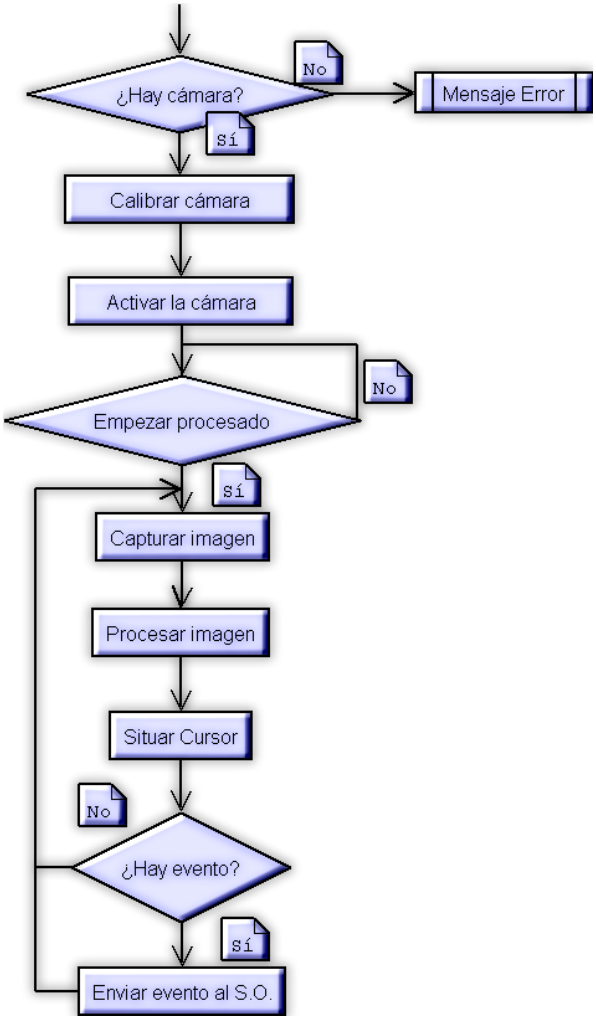


Fig. 4

En un principio, deberemos detectar la existencia de una cámara, en caso contrario, salimos de la aplicación avisando al usuario. Una vez detectada la cámara, se realizará una calibración de esta. Si no se realiza este paso, no existiría una correspondencia entre la imagen capturada y el escritorio debido a la perspectiva y la distorsión de la lente. La información de la calibración es guardada en un archivo para poder omitir este paso en futuros usos, ya que si la cámara no se ha movido de posición, los datos de calibración son los mismos. Este paso se explicará de forma más extensa en el punto 3.1.1.

Una vez calibrada la cámara se activará y se empezará el proceso de VxC, una vez el usuario lo indique. Una vez empezado el procesado se realizarán secuencialmente los pasos de captura de imagen, procesado de la imagen (principal tarea de VxC de la aplicación ,punto 3.1.2), e interacción con el sistena operativo (situar el cursor y envío de eventos, punto 3.2).

4.1 Tarea Visión por Computador

La tarea de VxC consiste en dos puntos principalmente: calibración de la cámara, y detección del patrón de luz.

4.1.1 Calibración

Una de las primeras tareas que se ha de realizar al comienzo de la aplicación es la calibración, ya que la imagen capturada está distorsionada por la perspectiva (parámetros extrínsecos) y por la propia lente de la cámara (parámetros intrínsecos) (Fig. 5). Así pues, tendremos que corregir dicha distorsión y escalar la imagen para obtener una equivalencia entre la captura y nuestra pantalla. Este paso previo nos dará la información necesaria para posteriormente poder interactuar correctamente con nuestro ordenador.

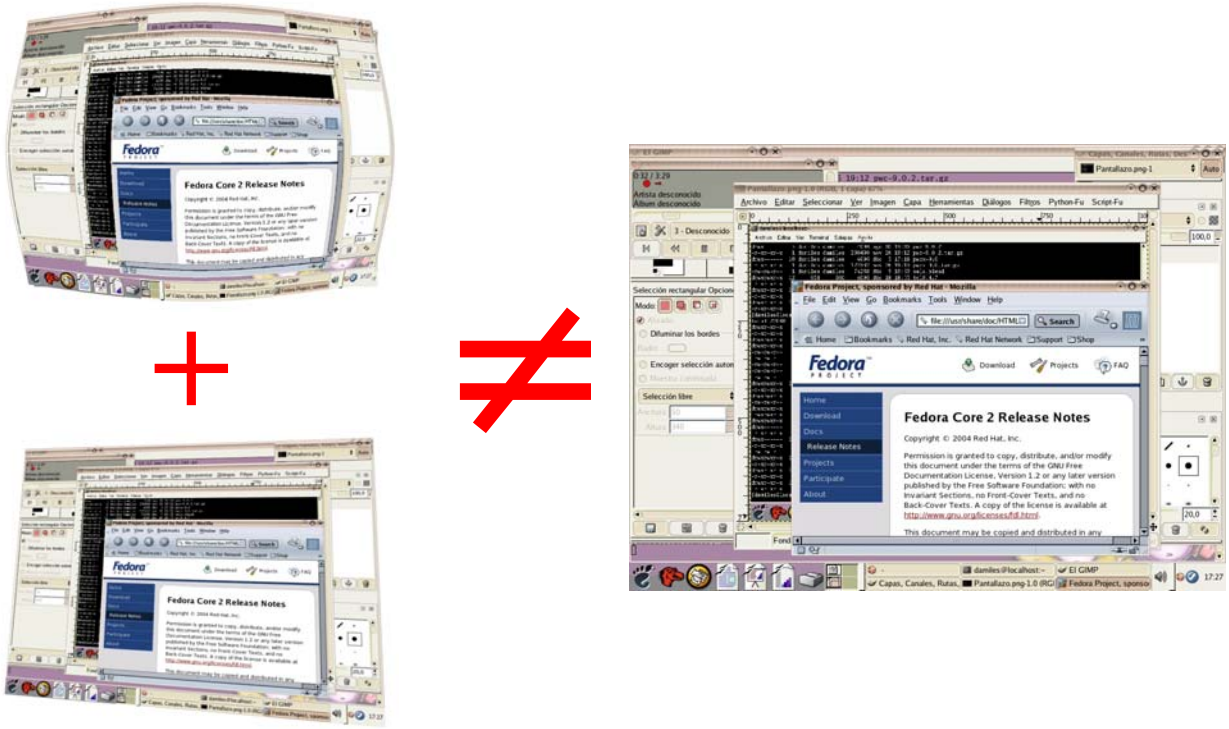


Imagen capturada Imagen proyectada

Fig. 5

Para poder solucionar esta distorsión y obtener una equivalencia se hace uso de una operación

matemática de transformación, llamada homografía. Esta nos resuelve dicha distorsión y se expresa generalmente en un producto de matrices (Fig. 6).

$$\lambda \begin{bmatrix} x1' \\ x2' \\ 1 \end{bmatrix} = \begin{bmatrix} H_{11} H_{12} H_{13} \\ H_{21} H_{22} H_{23} \\ H_{31} H_{32} H_{33} \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ 1 \end{bmatrix}$$

Fig. 6

Así pues, con esta formula de homografía y una vez calculada la matriz H, dado un punto (x1,x2) de la imagen capturada , se obtiene por producto, el punto (x1', x2'), que equivaldría con el punto de la imagen proyectada sobre el cual estaría incidiendo el puntero láser.

Pero,¿como obtenemos la matriz H?. Mediante el proceso previo de calibración comentado anteriormente, que consiste en definir de forma manual o automática cuatro puntos que a priori sabemos su equivalencia. Para ello, se toma como referencia las cuatro esquinas de la pantalla, que para una resolución de pantalla de 1024x768 serían: p1(0,0) p2(1024,0) p3(1024,768) y p4(0,768).

En nuestra aplicación, en un principio, se desarrolló una forma de calibración automática por medio de diferencias de imágenes, proyectando dos imágenes, una sin puntos y otra con puntos en cada esquina, de forma que con la diferencia de imágenes se obtenían 4 puntos: p1'(x11,x12), p2'(x21,x22), p3'(x31,x32) y p4'(x41,x42), que sabíamos de antemano que p1 equivale a p1' y sucesivamente, de forma que es trivial la resolución de la matriz H.

Posteriormente, debido a problemas de iluminación y otros contratiempos, se decidió realizar la calibración de forma manual. Esto, no supone ningún esfuerzo al usuario, porque dicha calibración solo seria necesaria una vez, si la cámara no se mueve de posición, ya que los datos necesarios para la calibración son almacenados en un fichero de registro.

Esta calibración manual se realiza mediante una ventana de calibración (fig. 7) de forma muy intuitiva, ya que se muestra una imagen capturada. Simplemente pulsando sobre cada esquina de la pantalla de proyección que muestra la imagen, se calibra la cámara.

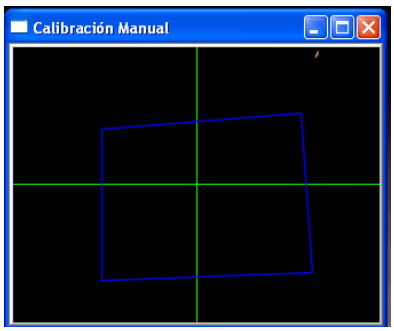


Fig. 7

La linea azul son los limites la pantalla de proyección y las verdes son una guia para poder asegurarnos de que la cámara está centrada, de forma que la intersección de líneas verdes es el centro donde debería estar la pantalla de proyección. Cuanta más zona azul abarque la ventana de calibración mejor porque se perderá menos resolución. Para que la zona azul abarque más ventana, simplemente deberemos acercar la cámara a la pantalla de proyección asegurándonos de que la abarque toda.

Para desarrollar esta interfaz se ha hecho uso de las librerías “highgui” de OpenCV, que nos ofrece la posibilidad de crear de forma simple ventanas de interacción, independientemente de la plataforma en la que se desarrolle. También nos facilita el manejo de imágenes que se muestra en ellas.

Debemos tener en cuenta que a la hora de calibrar la camara cuanta más porción de captura ocupe la pantalla de proyección más precisión obtendremos.

Así pues nuestro algoritmo de calibración quedaria de la siguiente forma:

```
//Funcion que a partir de un punto dado, calcula en que sector donde se encuentra y redibuja el poligono de 4 puntos que
//Sirve como las esquinas de calibración
void situarPuntoCalibracion(int x,int y){
	CvPoint* puntos=0;
	puntos=(CvPoint*)malloc( 4 * sizeof(puntos[0]));
	int numpts=4;
	img_Calib=cvCloneImage(fondo);
	//Ponemos los puntos segun han pinchado
	if ((x!=-1)||(y!=-1))
	{
		if(x<(ancho/2)){//sectores 1 y 4
			if(y>(alto/2)){
				//sector 1
				p1.x=x;
				p1.y=y;
```

```
                        }else{
                                //sector 4
                                p4.x=x;
                                p4.y=y;
                        }
                }else{//sectores 2 y 3
                        if(y>(alto/2)){
                                //sector 2
                                p2.x=x;
                                p2.y=y;
                        }else{
                                //sector 3
                                p3.x=x;
                                p3.y=y;
                        }
                }
        }
        puntos[0]=p1; puntos[1]=p2; puntos[2]=p3; puntos[3]=p4;
        cvPolyLine( img_Calib, &puntos, &numpts,1, 1, CV_RGB(0,0,255), 1, CV_AA, 0);
        cvShowImage("Calibración Manual",img_Calib);
}
//Funcion de captura de eventos del ratón
void on_mouse_calib( int event, int x, int y, int flags )
{
   switch( event )
   {
        case CV_EVENT_LBUTTONUP:{
                        puedoEscuchar=0;
                        situarPuntoCalibracion(x,alto-y);
                }break;
        case CV_EVENT_LBUTTONDOWN:{
                        puedoEscuchar=1;
                }break;
        case CV_EVENT_MOUSEMOVE:{
                        if(puedoEscuchar==1)
                                situarPuntoCalibracion(x,alto-y);
                }break;
   }
}
//Funcion de inicializacion de la calibración manual. Crea la ventana de interactuación y  elementos necesarios.
void calib_manual(){
        //Creamos la ventana
        char* name="Calibración Manual";
        cvNamedWindow(name,0);
        cvResizeWindow( name, ancho, alto );
        //Creamos una cruz de quia para separar los sectores
        CvPoint pq1=cvPoint(ancho/2,0);
```

```
        CvPoint pq2=cvPoint(ancho/2,alto);
        CvPoint pq3=cvPoint(0,alto/2);
        CvPoint pq4=cvPoint(ancho,alto/2);
        cvLine(fondo,pq1,pq2,CV_RGB(0,255,0),1,8);
        cvLine(fondo,pq3,pq4,CV_RGB(0,255,0),1,8);
        //Clonamos la imagen de fondo para trabajar sobre ella
        img_Calib=cvCloneImage(fondo);
        //Creamos los eventos de ratón
        cvSetMouseCallback( name, on_mouse_calib );
        //comenzamos la calibración
        situarPuntoCalibracion(-1,-1);
}
```

Así pues, con este algoritmo que realiza una interactuación con el usuario, que obitiene cuatro puntos p1, p2, p3 y p4 que necesitaremos para calcular la matriz de homografía fig6., para ello haremos uso de la función void CalcMH(). Esta función se encarga de generar todas las operaciones matemáticas necesarias para calcular la matriz H que nos servirá p ara despues calcular la relación entre el punto detectado en la imagen y nuestro escritorio.

Luego para calcular la matriz H, como habiamos comentado tenemos el siguiente sistema de ecuaciones matriciales:

$$\lambda 1\begin{bmatrix} x1' \\ y1' \\ 1 \end{bmatrix}=\begin{bmatrix} H_{11}H_{12}H_{13} \\ H_{21}H_{22}H_{23} \\ H_{31}H_{32}H_{33} \end{bmatrix}\begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix}$$

$$\lambda 2\begin{bmatrix} x2' \\ y2' \\ 1 \end{bmatrix}=\begin{bmatrix} H_{11}H_{12}H_{13} \\ H_{21}H_{22}H_{23} \\ H_{31}H_{32}H_{33} \end{bmatrix}\begin{bmatrix} x2 \\ y2 \\ 1 \end{bmatrix}$$

$$\lambda 3\begin{bmatrix} x3' \\ y3' \\ 1 \end{bmatrix}=\begin{bmatrix} H_{11}H_{12}H_{13} \\ H_{21}H_{22}H_{23} \\ H_{31}H_{32}H_{33} \end{bmatrix}\begin{bmatrix} x3 \\ y3 \\ 1 \end{bmatrix}$$

$$\lambda 4\begin{bmatrix} x4' \\ y4' \\ 1 \end{bmatrix}=\begin{bmatrix} H_{11}H_{12}H_{13} \\ H_{21}H_{22}H_{23} \\ H_{31}H_{32}H_{33} \end{bmatrix}\begin{bmatrix} x4 \\ y4 \\ 1 \end{bmatrix}$$

Que simplificandolo obtenemos la siguiente ecuación matricial:

$$U_{8x1}=A_{8x8}\cdot h_{8x1}$$

Siendo U una matriz de 8 fila y una columna definida por los 4 puntos de las esquinas del escritorio:

$$U_{8\,x1} = \begin{bmatrix} x1' \\ y1' \\ x2' \\ y2' \\ x3' \\ y3' \\ x4' \\ y4' \end{bmatrix}$$

Siendo A una matriz de 8 filas por 8 columnas como sigue:

$$A_{8\,x8} = \begin{bmatrix} p1.x & p1.y & 1 & 0 & 0 & 0 & -u1.x*p1.x & -u1.x*p1.y \\ 0 & 0 & 0 & p1.x & p1.y & 1 & -p1.x*u1.y & -p1.y*u1.y \\ p2.x & p2.y & 1 & 0 & 0 & 0 & -u2.x*p2.x & -u2.x*p2.y \\ 0 & 0 & 0 & p2.x & p2.y & 1 & -p2.x*u2.y & -p2.y*u2.y \\ p3.x & p3.y & 1 & 0 & 0 & 0 & -u3.x*p3.x & -u3.x*p3.y \\ 0 & 0 & 0 & p3.x & p3.y & 1 & -p3.x*u3.y & -p3.y*u3.y \\ p4.x & p4.y & 1 & 0 & 0 & 0 & -u4.x*p4.x & -u4.x*p4.y \\ 0 & 0 & 0 & p4.x & p4.y & 1 & -p4.x*u4.y & -p4.y*u4.y \end{bmatrix}$$

Y por último la matriz h definida como una matriz de 8 filas y una columna que equivale a la matriz H como siguie:

$$h_{8\,x1} = \begin{bmatrix} h1 \\ h2 \\ h3 \\ h4 \\ h5 \\ h6 \\ h7 \\ h8 \end{bmatrix} \Leftrightarrow \begin{bmatrix} h1 & h2 & h3 \\ h4 & h5 & h6 \\ h7 & h8 & 1 \end{bmatrix} = H_{3\,x3}$$

Así pues, como conocemos de antemano los puntos p1,p2,p3 y p4, definidos por el usuarios de forma manual o mediante una autocalibración, y sus respectivos equivalentes al escritorio, las esquinas del escritorio, por sustitución tenemos definidas la matriz A, y la matriz U, que despejando de la ecuación $U_{8x1} = A_{8x8} \cdot h_{8x1}$ obtendremos la matriz h $A_{8x8}^{-1} \cdot U_{8x1} = h_{8x1}$ que como hemos indicado anteriormente equivale a la matriz H y que es necesaria para calcular la equivalencia entre un punto de la imagen y el escritorio.

Entonces la función encargada de ello quedaría como sigue:

```
void CalcMH(){
  //puntos de las esquinas del escritorio que para este caso es una resolución de 1024x768, estas son en el programa final detectadas por el sistema operativo
  CvPoint u1,u2,u3,u4;
  u1.x=0;u1.y=768;//0-1
  u2.x=1024;u2.y=768;//1-1
  u3.x=1024;u3.y=0;//1-0
  u4.x=0;u4.y=0;//0-0


//Definicion de la matriz A, a partir de los puntos del escritorio (U) y los puntos de la calibración manual o automatica (p)
  double A[]={
        p1.x,   p1.y,   1,   0,     0,     0,   -u1.x*p1.x,   -u1.x*p1.y,
        0,      0,      0,   p1.x,   p1.y,   1,   -p1.x*u1.y,   -p1.y*u1.y,
        p2.x,   p2.y,   1,   0,     0,     0,   -u2.x*p2.x,   -u2.x*p2.y,
        0,      0,      0,   p2.x,   p2.y,   1,   -p2.x*u2.y,   -p2.y*u2.y,
        p3.x,   p3.y,   1,   0,     0,     0,   -u3.x*p3.x,   -u3.x*p3.y,
        0,      0,      0,   p3.x,   p3.y,   1,   -p3.x*u3.y,   -p3.y*u3.y,
        p4.x,   p4.y,   1,   0,     0,     0,   -u4.x*p4.x,   -u4.x*p4.y,
        0,      0,      0,   p4.x,   p4.y,   1,   -p4.x*u4.y,   -p4.y*u4.y
        };
//Definicion de la matriz U
  double U[8]={u1.x,u1.y,u2.x,u2.y,u3.x,u3.y,u4.x,u4.y};
  //Inicializamos las matrizes de OpenCV donde trabajaremos
  CvMat MA,MU;
  cvInitMatHeader( &MA, 8, 8, CV_64FC1, A );//Funcion que inicializa una matriz y asigna un Array a dicha matriz
  cvInitMatHeader( &MU, 8, 1, CV_64FC1, U );
  //creamos matriz inversa de A;
  CvMat MAinv;
  double Ainv[64];
  cvInitMatHeader( &MAinv, 8, 8, CV_64FC1, Ainv );
  cvmInvert(&MA,&MAinv);
  //obtenemos la matriz h
  CvMat Mh;
  double h[8];
```

```
    cvInitMatHeader( &Mh, 8, 1, CV_64FC1, h );
    cvmMul(&MAinv,&MU,&Mh);
    //Creamos la matriz H;
    for(int i=0;i<8;i++)
                H[i]=h[i];
    H[8]=1.0;
    //Almacenamos la matriz H
    cvInitMatHeader(&MH,3,3,CV_64FC1,H);
}
```

Una vez definida la matriz H con el algoritmo anterior, ya podemos definir la función que dado un punto de la imagen calibrada calcula su equivalente en el escritorio, esta función se ha llamado como TransPunto, que se le pasa como parametros dos enteros x, e y y devuelve un punto.

```
CvPoint2D32f TransPunto(int x,int y){
    CvPoint2D32f aux;
    CvMat Maux,Maux1;
//Creamos los arrays necesarios para almacenar los puntos
    double mp[]={x,y,1};//Punto dado
    double mp1[3];//punto resultado de la equación p'=H·p

//Inicializamos las estructuras de matrices de openCV
    cvInitMatHeader( &Maux, 3, 1, CV_64FC1, mp);
    cvInitMatHeader( &Maux1, 3, 1, CV_64FC1, mp1);
    //Realizamos la operación MH·Mp=Mp'
    cvmMul(&MH,&Maux,&Maux1);
//El resultado devuelto será una matriz p1 de 3 filas y una columna, de donde el punto equvalente x'=p1[0]/p1[2] y'=p1[1]/p1[2]
//p1[2] es el factor escala o como se definia en la funcion lambda
    aux.x=mp1[0]/mp1[2];
    aux.y=768-(mp1[1]/mp1[2]);
    //Devolvemos el punto obtenido
    return (aux);

}
```

4.1.2 Seguimiento y detección

La primera opción de desarrollo se ha planteado mediante un seguimiento de la imagen, es decir, mediante la búsqueda de movimiento producido por el láser al pasar sobre un fondo estático, este planteamiento inicial se desarrolla mediante diferencias de imagenes, imagen base menos imagen capturada o imagen anterior menos imagen capturada. Así pues, matemáticamente podríamos definir el algoritmo de “diferencias“ de la siguiente manera:

$$F(x,y)= | f(x,y)_{t} - f(x,y)_{t-1} |$$

donde f devuelve la componente de Rojo entre los rangos [0..255] del pixel definido por las coordenadas x, y. Así pues definiriamos de la siguiente forma el algoritmo de búsqueda del punto ha buscar.

$$(x , y) : F(x,y) \neq 0 \ \& \ F(x,y) \geq F(x1,y1)$$

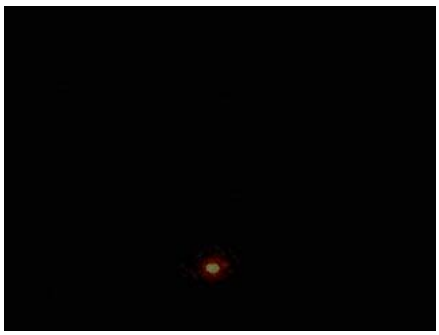


Fig.4

Éste se resuelve correctamente, pero no ante todas las situaciones que queremos abordar, como por ejemplo con fondos en movimiento como puede ser diapositivas cambiantes o incluso sobre vídeos, cambios de luz, o cualquier situación en que se pueda encontrar nuestro escritorio proyectado.

Así pues el problema se ha resuelto mediante el estudio físico de la iluminancia en donde buscamos el punto de la imagen, donde la componente roja más staurada supera un umbral de reconocimiento dado. Luego afirmamos que siempre que se detecte este punto que supera el umbral se ha detectado la presencia del láser.

Esto afirmación es facilmente explicable mediante la iluminancia emitida por los dos dispositivos utilizados, el puntero y el proyector de vídeo:

Un proyector genérico está alrededor de unos 2000 lumens, y un puntero láser con una potencia de 1mW emite como máximo 0,683 lumens, pero la superficie que abarca cada dispositivo es muy distinta, pues un proyector ilumina una superficie de unos 3x2 metros y un puntero, aproximadamente, 1 centimetro cuadrado.

Proyector: Iluminancia=E=Lumen/m^2

E=2000 Lumen / (3x2) m^2 =333,3 lux

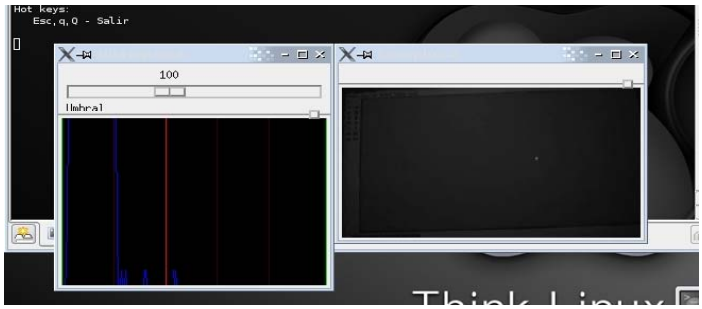
Puntero Láser Cat II con 1mW max=0,683 Lumen

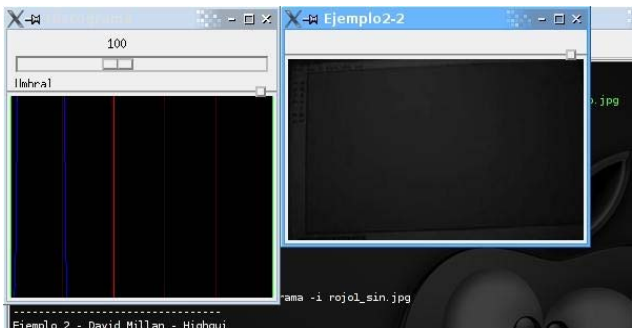
E=0,683 Lumen / (0,01x0,01) m^2 =6830 lux

De esta forma observamos que el puntero láser tiene 6830 lux mientras que el proyector unos 333 lux, y sabiendo que la iluminancia está en relación a la luz recibida por la cámara, podemos afirmar que siempre, absolutamente siempre, el puntero láser estará más saturado que el proyector.

Así pues nuestro problema se divide en dos fases: diferenciar entre el fondo y el láser, y entrontrar la posición del punto del laser dentro de la imágen.

Para diferenciar entre el fondo y el láser nos ayudaremos con un histograma. Para realizar estas pruebas de forma simple hemos creado un pequeño programa que nos facilita el histograma de una imágen que convierte previamente en escala de grises, este programa facilita un control a modo de barra de desplazamiento para poder encontrar fácilmente un umbral.





Como se puede apreciar en el histograma se puede introducir un umbral de 100 en una escala de 0-255, que hará que podamos diferenciar el fondo del láser, este umbral es impuesto por el usuario en el inicio del programa o almacenado para futuras sesiones del programa, y se realiza de forma empírica dependiendo de las condiciones del entorno.

Ahora solamente nos falta poder localizar en la imagen el láser, para ello vamos ha utililzar una gráfica tridimensional que para cada punto de la imagen mostrará su nivel de saturación o luminosidad. Como hemos comentado en el parrafo anterior, aquel punto que pase del umbral impuesto será tomado como el punto del puntero.

En la gráfica se observa la totalidad de la imagen capturada que deja entrever la luz que viene de una ventana lateral, esto podría ser un inconveniente, ya que la luz del exterior sí puede ser tomado como el puntero láser, como se observa en la gráfica, fig 4.2.1.1, pero como previamente se ha realizado una calibración de la cámara, sabemos pues los limites de la pantalla de proyección y entonces tenemos definida una area de interés, la cual nos va ha ayudar ignorar todo aquello que no sea proyectado, y así no interferir en nuestro porceso.

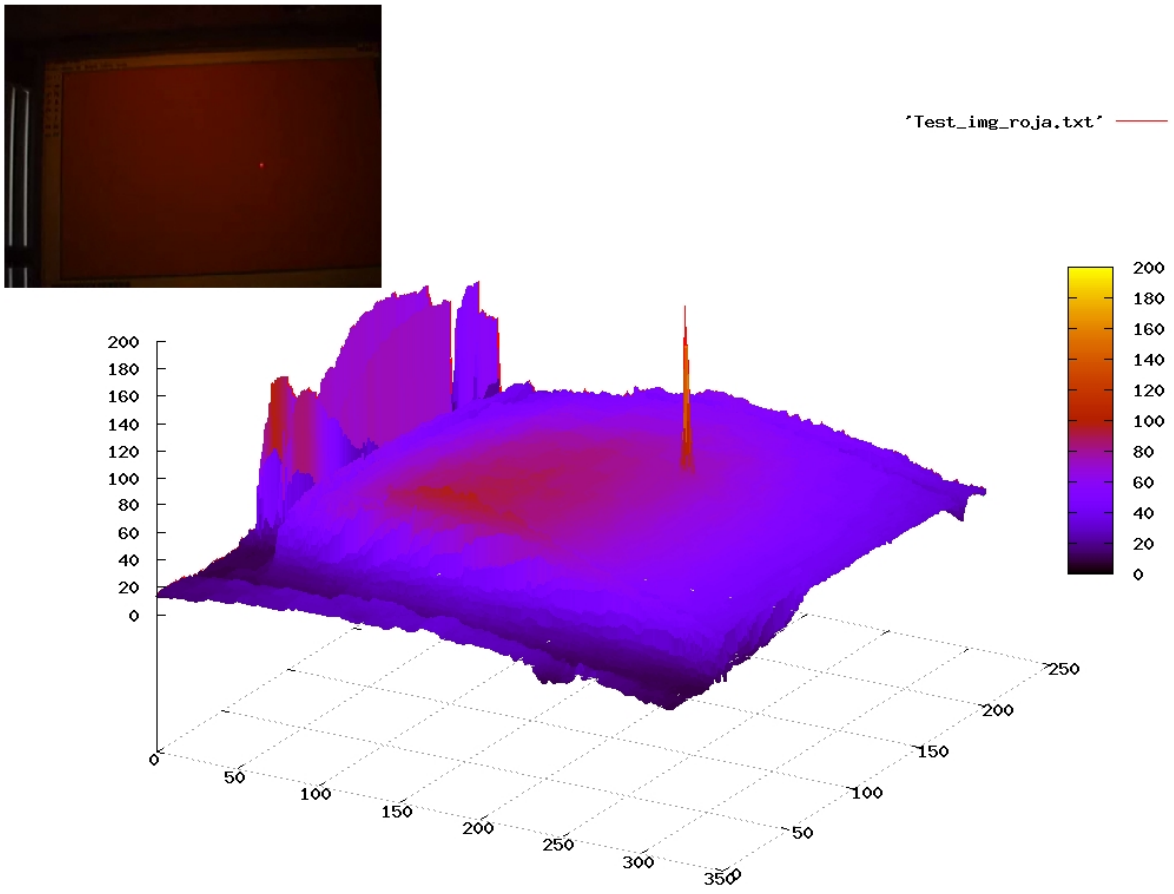


Fig 4.2.1.1

De tal forma que podriamos reformular de la siquiente forma la solución al problema de detección y seguimiento.

Tenemos una imagen, que al interesanos únicamente los niveles de luz, podemos trabajar con una imagen en escala de grises, por lo tanto la imagen la podemos expresar como una matriz de un numero definido de columnas por un número definido de filas:

$$M_{ancho\,x\,alto}=\begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} & \cdots & p_{1,ancho} \\ p_{2,1} & p_{2,2} & p_{2,3} & \cdots & p_{2,ancho} \\ p_{3,1} & p_{3,2} & p_{3,3} & \cdots & p_{3,ancho} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ p_{alto,1} & p_{alto,2} & p_{alto3} & \cdots & p_{alto,ancho} \end{bmatrix}$$

Donde $P_{i,j}\in[0..255]$, siendo $P_{i,j}$ el valor que toma en la escala de grises el pixel x,y (x=i, y=j) de la imágen.

De esta forma un Histograma se define como la funcion f(T) que es igual a la suma de todos los puntos que tienen un el valor $T\in[0..255]$ dentro de la imagen.

$$f(T)=\sum(\frac{P_{i,j}}{T}): P_{i,j}=T\wedge P\,i,j\in M_{ancho\,x\,alto}$$

Con lo que el umbral es un valor $U\in[0..255]$ tal que:

$$Laser\Leftrightarrow P_{i,j}>U$$
$$Fondo\Leftrightarrow P_{i,j}<U$$

De esta forma el proceso de detección y seguimento quedaría como sigue:

$$Punto(x,y)\,del\,Láser=Max(P_{i,j}): P_{i,j}>U\,, i\in[0..ancho]\,, j\in[0..alto]\,, U\in[0..255]$$

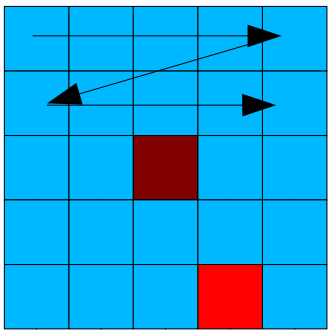
Por lo tanto, nuestro algoritmo quedaría como sigue en forma de pseudocódigo:

```
funcion deteccionSeguimiento (imagen)
    valorMax = 0
    xMax, yMax
    desde x =0 hasta imagen.ancho
```

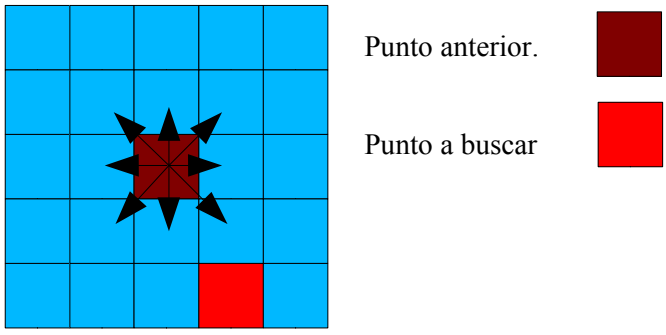
```
                desde y = 0 hasta imagen.alto
                        si (imagen(x,y)>valorMax) entonces
                                valorMax=imagen(x,y)
                                xMax=x
                                yMax=y
                        finsi
                findesde
        findesde
        si (valorMax > Umbral) entonces
                retorna el punto definido por xMax, yMax, (valorMax es el puntero Láser)
        sino
                retorna que no se ha detectado puntero, todo es fondo
        finsi
finfuncion
```

A partir de este punto solo nos faltaría la interacción con el sistema operativo, los eventos que se explicará en el punto 4.2.

Una posible mejora del algoritmo podria realizarse mediante la busqueda del punto a partir de su última posición conocida con un algoritmo 8-conectada de forma recursiva.



Algoritmo de recorrido o fuerza bruta



Algoritmo 8-conectada

Ya que como hemos comentado el Umbral impuesto nos servirá para distinguir el fondo del láser, de esta forma todo aquel pixel que pase del umbral es reconocido como láser, y si suponemos que el láser no abarca mas que unos 10 pixels de área, podriamos parar el algoritmo nada más detectar un pixel que sobrepase dicho umbral.

Pero el pixel detectado no tiene porque ser el adecuado, ya que el punto del láser es de mayor dimension que un pixel, luego deberiamos, a partir de dicho punto realizar una búsqueda para hallar el área afectada por el puntero y calcular el centro de dicha area. De esta forma obtendríamos el punto exacto. Este cálculo aumentaría en complejidad el código y por lo tanto su coste computacional.

Además existen errores diversos como por ejemplo por resolucion de captura, homografía, escalado, que hacen que este proceso de cálculo del área sea absurdo puesto que la precisión va ha verse afectada por muchos otros factores quedando relegado a segundo plano el área del puntero, este proceso debería tomarse en cuenta si tubieramos la intención de realizar capturas mayores que el escritorio, o estuviesemos interesados en obtener una precisión mayor.

4.2 Eventos

Llegados a este punto, hemos conseguido detectar el punto dentro de la imágen, y por lo tanto sabemos que hay o no acción del puntero láser dentro de la imágen.

Nuestro siguente punto dentro del algoritmo, definido al comenzar el proyecto, es saber si se produce o no un evento realizado por el usuario.

Para ello lo primero que debemos preguntarnos es cómo vamos a comunicarnos con nuestro ordenador, mediante hardware o softwar, en un principio se barajó la opción de realizar el envio de eventos mediante hardware como si fuese el lápiz de una tableta gráfica, que mediante un pulsador o dos como en los lápices de las tabletas, podemos enviar el evento de clic o doble clic, o como el usuario desease programar los botones, así pues mediante una interfaz hardware nos comunicariamos con nuestro ordenador.

Se realizó un prototípo mediante un pulsador y una comunicación LPT (puerto paralelo) como se puede apreciar en la figura 4.2.1. El prototípo tenía plena funcionalidad, y una gran respuesta, pero la imposición de cables es un gran impedimento para la movilidad del usuario, además no cumple nuestro requisito de no intrusivo.

Así pues este dispositivo se descartó, a partir de este concepto se buscó una forma más optima para este dispositivo, y se penso en un dispositivo inalambrico imitando la ergonomia de los lápices de las tabletas gráficas como se muestra en la figura 4.2.2, pero al tener que crear un dispositivo nuevo caeriamos otra vez en contradicción con uno de los requisitos impuestos, económico, ya que la creación de este nuevo dispositivo generaría un coste económico adicional encareciendo el producto final.

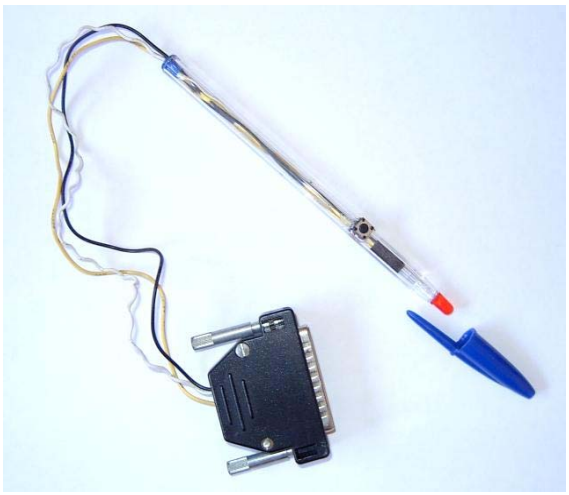


fig 4.2.1

fig 4.2.2

De tal forma, al observar que el proceso de detección y seguimiento tenía un coste computacional bastante económico, se opta finalmente por una solución software, en la que mediante un código de estados del puntero láser preestablecidos se entenderian como un evento u otro.

Así pues, el evento de un clic de ratón lo definimos como la transición de un estado del puntero detectado a un estado, de un tiempo determinado, en la que no se detecta el puntero, volviendo finalmente a un estado de detección del puntero de un tiempo mayor determinado pasando a un estado indeterminado culesquiera. Este evento se puede visualizar de forma más clara en el diagrama de estados 4.2.3.

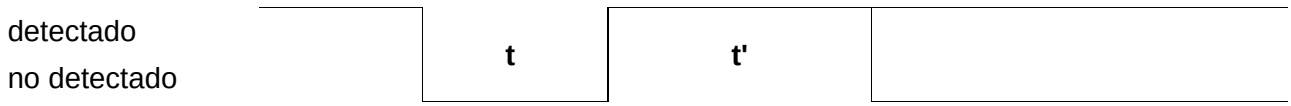


fig 4.2.3

Siendo t un tiempo determinado el cual puede variar entre tmin y tmax, tmin<t<tmax, y t' un tiempo mayor que un tiempo tmin, tmin<t'.

El doble clic del ratón se podría definir de la misma forma que un clic como se muestra en el diagrama de estados 4.2.4.

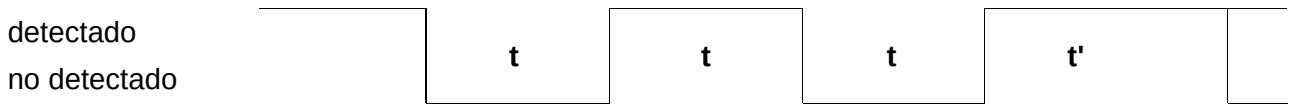


fig 4.2.4

Para nuestra aplicación, se ha desarrollado únicamente el evento de doble clic para poder lanzar aplicaciones, cerrar, ejecutar un icono del escritorio pasar de transparencia de una presentación, etc. Para ello hemos utilizado el diagrama de estados de la figura 4.2.3 por comodidad. Así pues aplicariamos el diagrama 4.2.3 como el siguiente diagrama de estados de la figura 4.2.5.

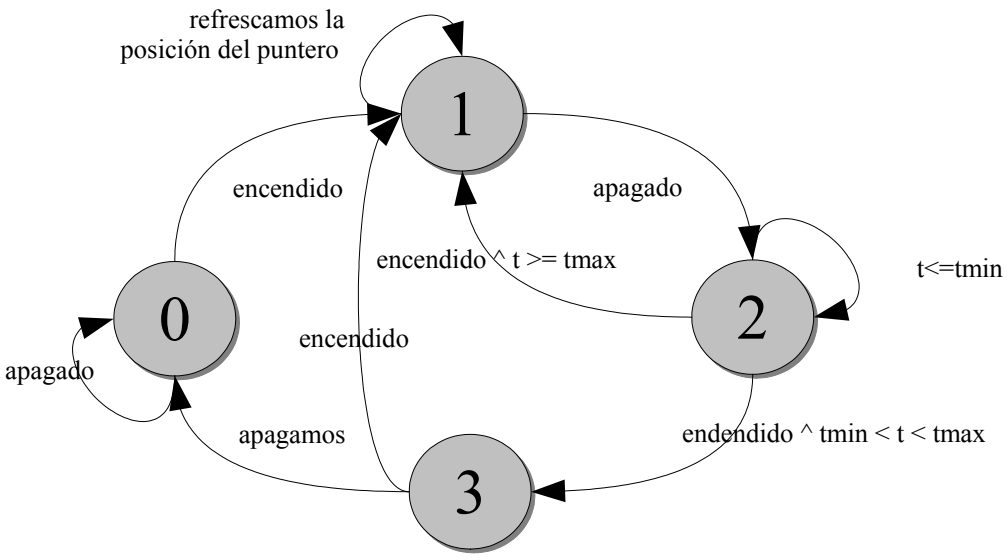


fig. 4.2.5

Los tiempos los tomaremos en referencia a los fotogramas capturados por no saturar más el sistema con un contador externo. De tal forma utilizaremos una variable que iremos incrementadon una vez apagado el làser, en caso de que la variable, una vez encendido otra vez el láser, tenga un valor entre tmin y tmax, lo entenderemos como un evento, y volveremos a inicializar la variable, cada vez que se vuelva ha detectar el puntero se inicializará la variable.

4.3 Valoraciones

Llegados a este punto, podríamos decir que nuestro proyecto ha llegado a su fin en cuanto a código, aunque como veremos los siguientes puntos deberemos introducir alguna línea que otra más, pero ante todo, vamos ha evaluar nuestro proyecto, en cuanto a los errores y precisión que tiene así como su rendimiento.

4.3.1 Errores: Resolución Vs Error Humano

Una vez llegados a este punto, nos falta saber si nuestro proyecto es preciso y que errores introduce, asi como si estos son lo suficientemente bajos como para despreciarlos o no tomarlos en cuenta.

Así pues, si estudiamos el proyecto veremos que se produce una pérdida de resolución considerable al trabajar con una cámara web con una resolución típica de 320x240, y en casos excepcionales con una resolución de 640x480. En el caso típico, estaríamos perdiendo una resolución considerable al tener que escalar la imagen capturada a una resolucion mucho mayor como puede ser la de la pantalla de un ordenador de entre 800x600 o más usual hoy en dia como de 1024x768, así pues podriamos decir que en una captura de 320x240 en la que se ha detectado un punto en la posicion (x,y) equivale a una zona de actuación sobre una pantalla de 1024x768 de 3,2 $pixel^2$, luego perderiamos la posibilidad de alcanzar por ejemplo de forma exacta el pixel (1,1), ya que un movimiento del puntero de un pixel de la imagen capturada se ve reflejado como una diferencia de 3 pixel al escalarla para posicionar el cursor en nuestro escritorio.

Si unimos la pérdida anterior y que no toda la imagen capturada pueda se nuestro interés debido a la calibración, este error se aumenta, y no de forma proporcional en todo el escritorio como se podía observar en el capitulo de calibración en la figura 5, en la que la perspectiva hace que la mayoría de veces una zona del escritorio tenga más resolución que otra. Así pues si centramos la cámara web e intentamos que la pantalla de proyección ocupe lo máximo én nuestra imagen capturada, menor pérdida acumularemos, esta pérdida por la calibración se ha estimado que puede aumentar en un pixel como mucho el error que teníamos por la resolución de la webcam, dejando de esta forma en 4 pixels de error.

Además, durante las primeras pruebas que realizamos con nuestro proyecto, nos percatamos de que no era fácil apuntar a un icono y dejar más o menos estático el puntero del ratón, ya que nosotros, como seres humanos, no somos perfectos, y al intentar apuntar fijamente a una zona en concreto, producimos un temblor, que somos incapaces de evitar, y que con la distancia aumenta, de forma que este temblor es captado por nuestro programa, y traducido en constantes movimientos del ratón, lo cual es una molestia para el usuario.

Para solucionar esta molestia se opta por no mover el cursor del ratón si el usuario no se desplaza por la imagen capturada más de número de pixels definidos por el propio usuario, es decir, que entre la última posición y la actual haya un incremento mayor que el número de pixels definidos por el usuario, ya que no todas las personas producimos el mismo temblor, se ha comprobado que con un desplazamiento de como mínimo mayor de 1 pixel de la imagen

capturada basta como para poder subsanar esta molestia.

Luego aquí introducimos el error humano, puesto que el usuario no podrá moverse directamente un pixel más a la derecha o a la izquierda, que una vez transformado por escala y calibración se convierte en 3 pixels.

Este error sumado al anterior, se puede considerar grande (7pixels) pero debido a que el error humano no se puede evitar, y va ha estar presente siempre, está justificado este error. Además, si consideramos que nuestra aplicación no va ha estar enfocada a obtener una precisión absoluta, sino a tener una interacción fluida e intuitiva con nuestro ordenador desde la distancia y con unos recursos mínimos, una perdida de 7 pixels, en una presentación de transparencias en la que no es necesario apuntar a ningun punto en especial para interactuar con el, sino que con un simple apaga-enciende del laser podemos pasar de transparencia, o en otro caso, interactuar con nuestro escritorio, para ejecutar una aplicación, en la que los iconos, suelen tener un mínimo de 16 pixels^2 o 32 en el caso de los del escritorio, este error es despreciable, al poder alcanzar nuestro objetivo, interactuar con nuestro ordenador (lanzar una aplicación, pasar transparencias, jugar a un juego de “dar al blanco”)

4.3.2 Rendimiento

Una vez terminado nuestro proyecto entramos en el proceso de evaluación de nuestro proyecto en cuanto al rendimiento que este nos ofrece, de tal forma que evaluamos de forma crítica si se han cumplido nuestros requisitos en cuanto a rendimiento.

Se ha observado que el proceso se comporta de forma estable y constante durante todos los test realizados, de forma que tanto el consumo de procesador así como de memoria permanece constante durante todo el uso del programa, este consumo se ve influenciado directamente dependiendo de el tamaño de captura de la webcam (320x240 o 640x480) así como por el número de fotogramas por segundos capturados (10, 15, 25, 30 fps).

El proceso puede ser mejorado mediante los algoritmos descritos anteriormente, además de el uso de macros.

En el portatil descrito en el punto 2.3 perifericos, el consumo tanto en Windows como en GNU/Linux ha sido de un 15% del procesador, así como de un incremento de memoria utilizada para el proceso de unos pocos MB. Así pues se puede considerar un proceso que pueda llevarse conjuntamente con el de una interacción con el escritorio, o como su principal tarea, la de

señalador interactivo para presentaciones, acompañado de un programa de presentaciones como podria ser el openOffice Impress o PowerPoint.

4.4 Experiencias

En este punto vamos ha explicar los inconvenientes o problemas con que nos hemos encontrado durante el desarrollo de nuestro poyecto en los dos sistemas operativos empleados.

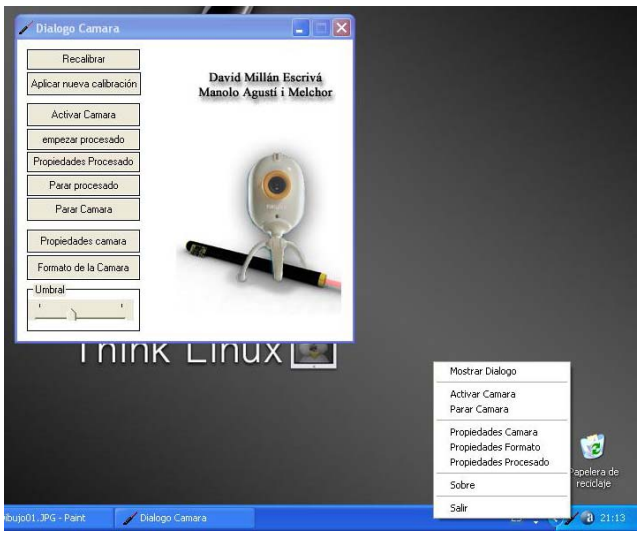
4.4.1 Windows

El desarrollo en la plataforma Windows ha sido bastante cómoda así como bastante versátil en cuanto al desarrollo de interfáces gráficas para el desarrollo del proyecto.

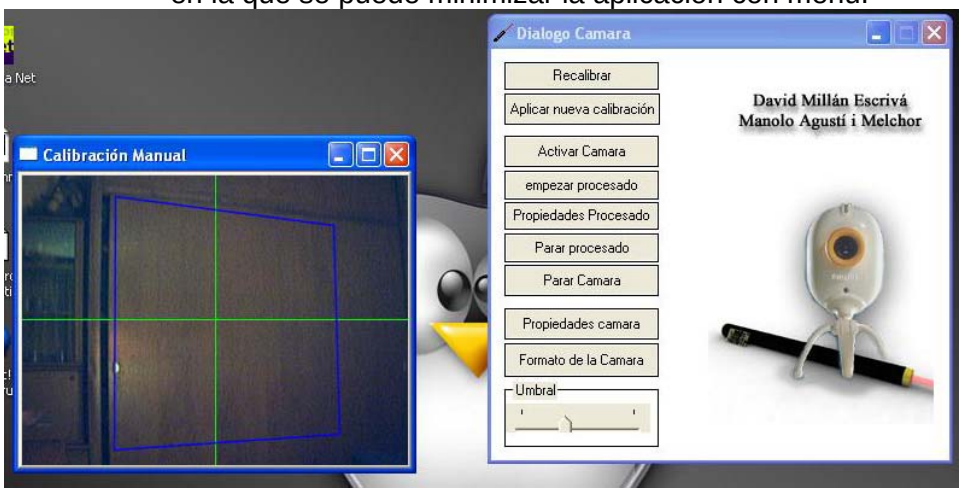
Tenemos que recalcar que el paso de configuración de nuestra plataforma de desarrollo de Windos, ya sea Visual Studio 6 o Visual .net, debemos realizar correctamente los pasos indicados en el punto 3.1 instalación, para poder compilar sin errores nuestros proyectos.

Así pues, la construcción de interfaces gráficas en Windows es realmente fácil, y cómodo en el desarrollo, de tal forma que nuestro proyecto ha sido desarrollado en Windows mediante distintas interfáces gráficas como se observa en las siguientes capturas en las que observaremos las distintas funciones que realiza nuestro proyecto.

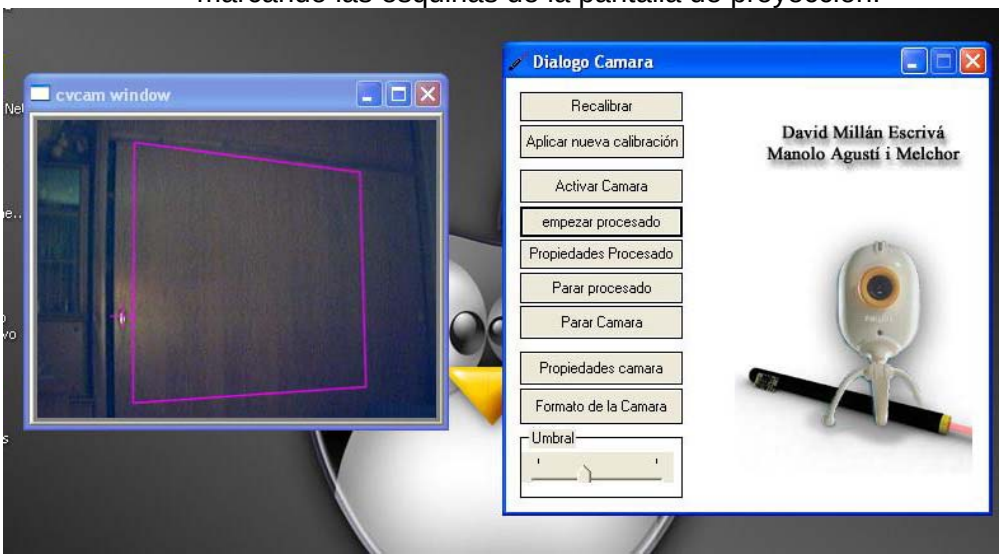
Pantalla principal



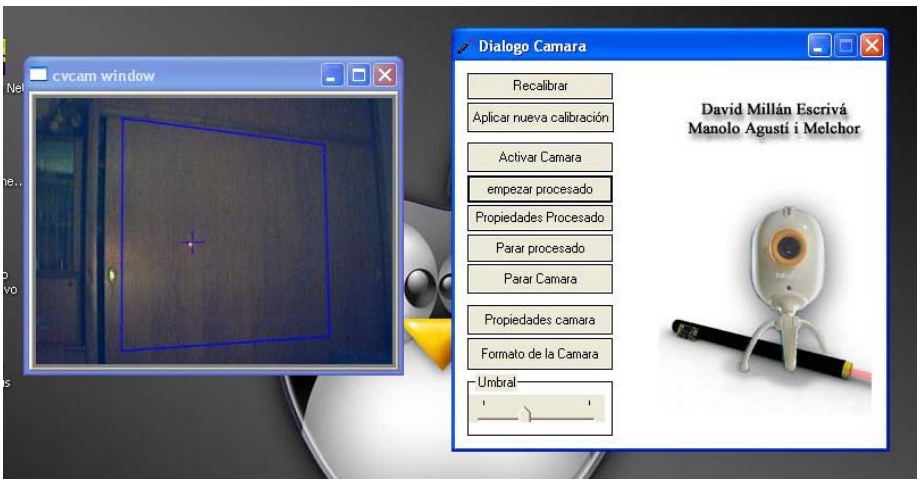
Pantalla principal con acceso en la barra de tareas
en la que se puede minimizar la aplicación con menu.



Pantalla de calibración, en la que se puede calibrar la cámara de forma intuitiva
marcando las esquinas de la pantalla de proyección.

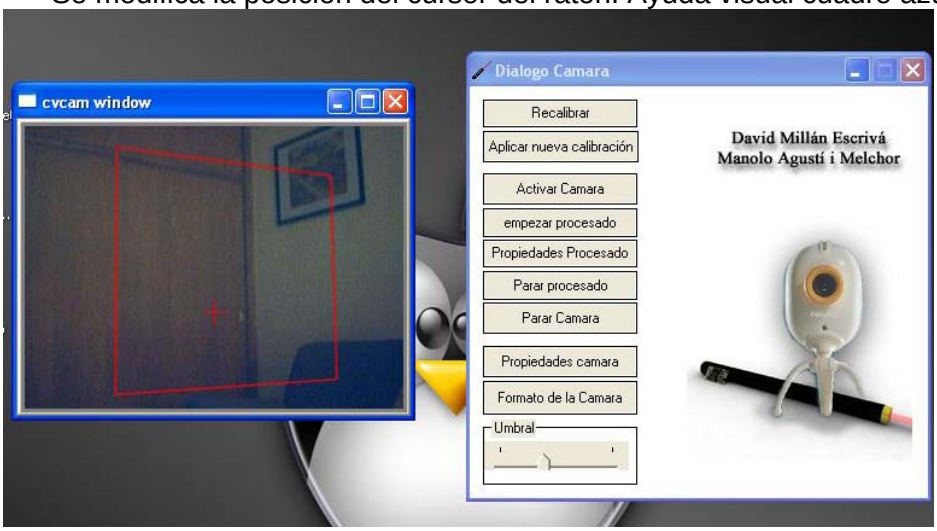


Programa en ejecución: punto detectado fuera del área de interés.
No se modifica la posición del cursor del ratón. Ayuda visual: cuadro rosa.



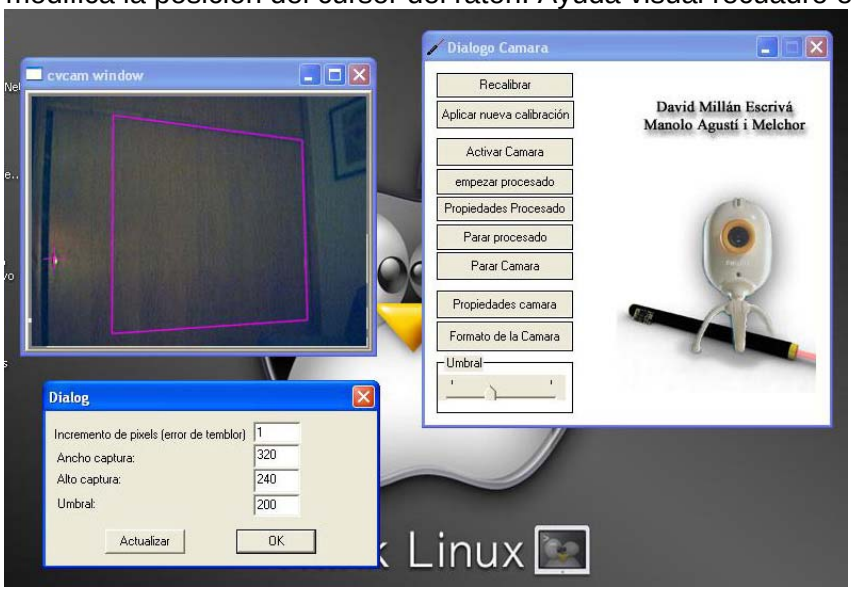
Programa en ejecución: punto detectado dentro del área de interes.

Se modifica la posición del cursor del ratón. Ayuda visual cuadro azul.



Programa en ejecución: Punto no detectado en la imagen.

No se modifica la posición del cursor del ratón. Ayuda visual recuadro en color rojo.



Cuadro de dialogo de propiedades de procesado de la imagen: Umbral...

4.4.2 GNU/Linux

El desarrollo bajo GNU/Linux ha sido más complicado, pero aún así se ha obtenido muy buenos resultados, cabe destacar que las librerías, de codigo abierto, aún están en versión beta, por lo que todavía carecen de estabilidad, como por ejemplo con las librerias de soporte de ventanas de GNU/Linux las X. Así pues se a intentado evitar en todo momento la utilización de ventanas gráficas, de forma que todo el trabajo se realiza en modo consola, de forma documentada.

Cabe destacar que para el buen funcionamiento de el proyecto, en caso de utilizar la misma cámara que la empleada en este proyecto, la webcam Philips TouCam, es necesario tener instalado el modulo PWC así como el descompresor de imagen PWCX, que en su versión más nueva ya viene las PWCX incluidas en las PWC, versión 10. La instalación de estos módulos, para dar soporte a la webcam necesita la mayoria de veces la recompilación del kernel, lo cual entraña una cierta complejidad, que con el tiempo y la experiencia este proceso se automatiza sin problemas.

5 Conclusión

En este proyecto se ha propuesto la realización de un sistema que permita extender el área de trabajo del escritorio de un sistema de ventanas actual al entorno de trabajo cotidiano. El sistema propuesto hace uso de los elementos ya existentes en este tipo de métafora de escritorio, sin introducir nuevos conceptos sino ampliando el marco de actuación de las posibilidades de interacción usuarles.

Siempre se ha hecho uso de herramientas que han permitido la portabilidad del sistema propuesto a las plataformas de trabajo habituales en el entorno de trabajo considerado. Los requerientos de materiales y de capacidad computacional para llevarlo a cabo han sido contemplados y minimizados para que su impacto fuese mínimo, así como las necesidades de aprendizaje de utilización del mismo.

Para ello se ha expuesto ha situaciones típicas del trabajo diario. El sistema se ha demostrado factible, económico y flexible.

Este proyecto ha sido evaluado por el V congreso INERACCIÓN2004 (Interacción persona ordenador) celebrado en Lérida, y aceptado en la categoria demostración-poster, e incluido el artículo en su líbro de actas, artículo “Una aproximación a la integración del computador en el entorno de trabajo real.” página 374 del libro de actas.

6 Bibliografía

- **Open Source Computer Vision Library.** Reference Manual. Edic. Intel Corporation.
- **Programming with Intel IPP. A Beginner's Tutorial.** Jérôme Landré. Institut Universitaire de Technologie. Laboratoire Électronique,Informatique et Images. Université de Bourgogne.
- **Programación en Visual6 C++.** Edic. Paraninfo. Stephen Gilbert. Bill McCarty.
- **Aprendiendo a trabajar con GNU/Linux.** Edic. Inforbook's. Bill Ball.
- **Computer display control and interaction using Eye-Gaze.** M Farid, F.Murtagh, J.L.Starck. Queen's University Belfast.
- **Robust Methods of computing fundamenta Matrices.** Phil David.
- **Detecting planar homographies in an image pair.** Etienne Vincent, Robert Laganière. School of information Technology and engineering. University of Ottawa.
- **Visual Navigation usign Planar Homographies.** Bojian Liang and Nick Pears. Departament of Computer Science. University of York.
- **Laboration 1: Geometry and Visualization.** Stefan Carlsson, Carsten Rother. Numerical Analysis and Computing Science. Stockholm.
- **Parameter estimation and calibration.** Hagen Spies. Computer Vision Laboratory. Departament of electrical engineering. Linköping University. Sweden.
- **On projection Matrices** $p^k \to p^2, k = 3,\ldots,6$ **, and their applications in Computer Vision.** Lior Wolf and Ammon Shashua. School of computer Science and Engineering. The Hebrew University. Jerusalem. Israel.
- **Notes on the Applications of Homographies in Computer Vision.** A. Fusiello. Dept. of computing & electrical engineering. Heriot-Watt University.
- **On Plane-Based Camera Calibration: A General Algorithm, Singularities, Applications.** Peter F. Sturm and Stephen J. Maybank. Computational Vision Group, Departament of computer Science, The University of Reading WhiteKnights.
- **Manual de visión en robótica.** Jose M. Cañas. Universidad Rey Juan Carlos.
- **Temario de Sistemas de Visión.** Antonio Sánchez (DISA) , Alberto J. Perez (DISCA). Universidad politécnica de Valencia.

7 Enlaces

- Group Open computer Vision Library in Yahoo. http://www.yahoo.group.com
- Intel. http://developer.intel.com
- SourceForge. http://soureforge.net
- OpenCV SourceForge.
- PWC SourceForge
- PWCXSourceForge
- Debian. http://www.debian.org
- Fedora.
- Mandrake.