2.8 OTRAS FUNCIONES

2.8.1 Conectar.cpp

Esta primera función es la que conecta con la cámara científica, para capturar las imágenes que después serán procesadas. Como característica sobresaliente, decir que esta función es la que desarrolla el modo Video de la cámara. Ahora podemos estudiar el código, convenientemente comentado, y después se añadirán matices y consideraciones para completar su explicación.

```
1   #include "stdafx.h"
2   #include "QCamAPI.h"   // para comunicarse con la camara
3   #include "CImg.h"   // para el tratamiento de imagenes
4
5   using namespace cimg_library;
6
7   // declaraciones de funciones
8   CImg<unsigned char> Capturar(QCam_Handle handle, QCam_Frame frame, unsigned char
    *ptr);
9   double DibujaLineaCalib(CImg<unsigned char> img);
10
11  double Conectar(void)
12  {
13     // valor que se devuelve
14     double dist;
15
16     // cargamos los drivers de la camara
17     QCam_LoadDriver();
18
19     // comprobamos si existe una camara conectada
20     QCam_CamListItem list;
21     unsigned long numberInList;
22     QCam_ListCameras(&list, &numberInList);
23
24     // si existe, comprobamos que no está abierta
25     if (list.isOpen == false)
26     {
27        // declaramos el manejador de la camara y abrimos una conexion con la camara
28        QCam_Handle handle;
29        QCam_OpenCamera(list.cameraId, &handle);
30
31        // leemos los parametros para definir nuestra Region de Interes
32        unsigned long maxWidth, maxHeight;
33        QCam_GetInfo(handle, qinfCcdWidth, &maxWidth);
34        QCam_GetInfo(handle, qinfCcdHeight, &maxHeight);
35
36        // declaramos la variable de lectura/escritura de propiedades de la camara
37        QCam_Settings settings;
38        settings.size = sizeof(settings);   // nunca olvidar esto!
39
40        // leemos las propiedades por defecto de la camara
41        QCam_ReadDefaultSettings(handle, &settings);
42
43        // escribimos los parametros para definir nuestra Region de Interes
44        unsigned long width, height, RoiX, RoiY;
45        width = 682;   // tamano de la ventana
46        height = 512;
47        RoiX = (maxWidth - width)/2;   // posicion de la esquina superior izquierda
48        RoiY = (maxHeight - height)/2;
49        QCam_SetParam(&settings, qprmRoiWidth, width);
50        QCam_SetParam(&settings, qprmRoiHeight, height);
51        QCam_SetParam(&settings, qprmRoiX, RoiX);
52        QCam_SetParam(&settings, qprmRoiY, RoiY);
53
54        // enviamos todos los parametros definidos anteriormente a la camara
55        QCam_SendSettingsToCam(handle, &settings);
56
57        // inicializamos los frames
58        unsigned long size;
```

```
59          QCam_GetInfo(handle, qinfImageSize, &size);
60
61          unsigned char* ptr;
62          ptr = new unsigned char[size];
63          QCam_Frame frame;
64          // inicializamos el buffer del frame donde se captura la imagen
65          frame.pBuffer = (void *)ptr;
66          frame.bufferSize = size;
67
68          // capturamos el primer frame
69          QCam_GrabFrame(handle, &frame);
70          // definimos la imagen que se va a usar
71          CImg<unsigned char> img(frame.width, frame.height);
72          // pasamos la informacion del buffer pixel a pixel
73          cimg_forXY(img, x, y){img(x, y) = ptr[x + frame.width*y];}
74          // cambiamos el formato de la imagen
75          img.BayertoRGB();
76          // definimos el display en el que se muestra la imagen
77          CImgDisplay disp(img.width(), img.height());
78          // y su posicion en la pantalla
79          const int posx(15), posy(40);
80          disp.move(posx, posy);
81          disp.display(img).set_title("Video - Pulse [X] para salir");
82
83          // bucle principal
84          while (!disp.is_closed())
85          {
86             // capturamos y mostramos imagenes mientras no se cierre el display
87             img = Capturar(handle, frame, ptr);
88             disp.display(img).set_title("Video - Pulse [X] para salir");
89          }
90          // y se muestra la ultima imagen capturada en video para medir sobre ella
91          dist = DibujaLineaCalib(img);
92
93          // limpiamos los buffers
94          delete [] frame.pBuffer;
95          frame.pBuffer = NULL;
96
97          // cerramos la camara
98          QCam_CloseCamera(handle);
99       }
100      // si hay algun fallo
101      else
102         dist = -1;
103      // liberamos los drivers de la camara
104      QCam_ReleaseDriver();
105
106      // devolvemos la longitud del segmento
107      return dist;
108   }
109
110   CImg<unsigned char> Capturar(QCam_Handle handle, QCam_Frame frame, unsigned char
      *ptr)
111   {
112      // seguimos capturando frames
113      QCam_GrabFrame(handle, &frame);
114      CImg<unsigned char> img(frame.width, frame.height);
115      cimg_forXY(img, x, y){img(x, y) = ptr[x + frame.width*y];}
116      img.BayertoRGB();
117
118      return img;
119   }
```

Al principio vemos cómo se van siguiendo todos los pasos que se enumeran en la documentación de las librerías de la cámara (../QImaging/SDK/QCam API.pdf), para poder conectarse a ella. Primero se cargan los drivers y se comprueba que haya una cámara disponible, luego se crean las variables necesarias para definir un manejador, configurar ciertas opciones, etc., para llegar a lo más importante, la reserva de espacio en memoria en forma de buffers para almacenar las imágenes que se capturarán después. Una vez capturada la imagen, termina la labor de la cámara, y toman el control las funciones de la librería CImg para mostrar un display, una ventana, en la que vemos cómo se escapan nuestras propias fuerzas. Porque pronto se cierra la cámara y se liberan los drivers.

Si se desea probar funciones nuevas de la cámara, se recomienda usar una variable de tipo QCam_Err como el que devuelven la mayoría de las funciones de la cámara, e implementarla en una condición en la que se exponga si actuó correctamente, a través de qerrSuccess (el primer tipo de error es el éxito), o si hay algún error, que quedaría contemplado con un valor numérico y entonces sólo hay que buscarlo en la lista de posibles errores contemplados.

Otro punto crítico de este trozo de código, es precisamente la inicialización del buffer del frame. Pongamos especial cuidado en la estructura QCam_Frame, donde el miembro pBuffer es un puntero a void, al cual le asignamos por ejemplo un puntero a unsigned char. Este paso no siempre es mostrado con claridad en la documentación disponible, y también es difícil encontrar ayuda en la red para la programación en lenguaje C++, siendo más común C# y Visual Basic.

Aun así, es importante para programar con la librería QCam API, el estudiar a fondo ejemplos que vienen en la documentación (../QImaging/SDK/Samples y dentro de esa carpeta, los ejemplo "ImgGrab" y "Simple Preview", aunque normalmente están escritos a otros niveles de C++, no para Formularios de Windows, pero sirven de ayuda).

También nos paramos a observar la rutina interna llamada "Capturar" y que asiste a la función principal con el modo Video. Podemos comprobar que este modo consiste en ir capturando frame por frame, y mostrándolo en cuanto se tiene en el display de la librería CImg. No se hace uso de la propiedad de Real Time Viewing de la cámara mediante la función asíncrona QCam_QueueFrame, y se sigue usando la síncrona QCam_GrabFrame, por lo que se consiguen ratios menores de frames por segundo mostrados por pantalla, pero aun así, el modo Video queda operativo, esperando a se desarrolle una verdadera aplicación asíncrona. Por ahora es suficiente con la funcionalidad que tiene.

2.8.2 CalibrarDesdeArchivo.cpp

Poco hay que decir de esta pequeña función, que sirve de puente cuando la cámara científica no está conectada, o aunque esté conectada se quiera abrir una imagen desde archivo, y no la que se captura directamente. Esta función es la que necesitaba la conversión de String^ a char*, debido a que CImg no trabaja con el primer formato.

```
1   #include "stdafx.h"
2   #include "CImg.h"   // para el tratamiento de imagenes
3
4   using namespace cimg_library;
5
6   // declaraciones de funciones
7   double DibujaLineaCalib(CImg<unsigned char> img);
8
9   // no conectamos con la camara y pasamos solo la ruta de la imagen
10  double CalibrarDesdeArchivo(char *fileAddress)
11  {
12     CImg<unsigned char> img(fileAddress);
13
14     return DibujaLineaCalib(img);
15  }
```

Y aquí hay que apuntar que los formatos de la imágenes que soporta abrir esta función, se reducen a los .bmp, y en cuanto se desea abrir una imagen .jpg o .tiff pueden saltar excepciones que bloquean la ejecución de la aplicación. De esto se hablará más en el apartado de Futuros Trabajos, teniéndose que buscar ayuda posiblemente en otras librerías como Magick++, u otras funciones propias de CImg que soporten estos formatos. Y sobretodo .tiff, que es un formato común de trabajo para los investigadores del oceanográfico para almacenar imágenes .

2.8.3 DibujaLineaCalib.cpp

Junto con la función Conectar esta es también muy importante, ya que una vez terminado el modo Video en Conectar, porque ya hayamos obtenido una imagen enfocada, ésta es la rutina en la que se dibuja un segmento sobre la imagen capturada para medir la distancia entre los dos puntos extremos del mismo y poder así calcular el factor de relación entre medidas en píxeles, que es lo que podemos contar con la librería CImg, y medidas reales en milímetros o en micras.

```
1   #include "stdafx.h"
2   #include <math.h>   // para abs, sqrt y pow al calcular la distancia
3   #include "CImg.h"   // para el tratamiento de imagenes
4
5   using namespace cimg_library;
6
7   // declaraciones de funciones
8   CImg<unsigned char> DibujaPunto(CImg<unsigned char> image, int x, int y, const
    unsigned char *color);
9
10  // dibuja un segmento (pto inicial y final) sobre la imagen img y devuelve su
    distancia
11  double DibujaLineaCalib(CImg<unsigned char> img)
12  {
13     // definimos una imagen auxiliar para poder dibujar sobre la otra
14     CImg<unsigned char> img_orig(img.width(), img.height());
15     img_orig = img;
16     // definimos el display en el que se muestra la imagen
17     CImgDisplay disp(img.width(), img.height());
18     // y su posicion en la pantalla
19     const int posx(15), posy(40);
20     disp.move(posx, posy);
21
22     // definicion de variables
23     int xi, yi, xf, yf;   // coordenadas de los puntos inicial y final
24     bool pto1(false), pto2(false), calib(false);   // condiciones de comprobacion
25     const unsigned char red[] = {255, 0, 0}, green[] = {0, 255, 0};   // colores
26     double dist(0.0);   // valor que se devuelve
27
28     // bucle principal
29     while (!calib)
30     {
31        // imagen sin nada dibujado
32        while (!disp.is_closed() && !pto1 && !pto2 && !calib)
33        {
34           img = img_orig;
35           disp.display(img).set_title("Calibrar microscopio: click izquierdo en
    primer punto");
36           disp.wait();
37           // se pide el punto inicial con el boton izquierdo del raton
38           if (disp.button()&1)
39           {
40              xi = disp.mouse_x();
41              yi = disp.mouse_y();
42              pto1 = true;
43              break;
44           }
45        }
46        // imagen con punto inicial fijado y linea flotante esperando el punto final
47        while (!disp.is_closed() && pto1 && !pto2 && !calib)
48        {
49           img = img_orig;
50           img = DibujaPunto(img, xi, yi, red);
51           img.draw_line(xi, yi, disp.mouse_x(), disp.mouse_y(), red);
52           disp.display(img).set_title("Calibrar microscopio: click izquierdo en
    segundo punto (click derecho para repetir)");
53           disp.wait();
54           // se pide el punto final con el boton izquierdo del raton
```

```
55          if (disp.button()&1)
56          {
57             img = img_orig;
58             xf = disp.mouse_x();
59             yf = disp.mouse_y();
60             pto2 = true;
61             break;
62          }
63          // o se cancela el punto inicial con el boton derecho del raton
64          else if (disp.button()&2)
65          {
66             pto1 = false;
67             break;
68          }
69       }
70       // imagen con el segmento que se ha introducido
71       while (!disp.is_closed() && pto1 && pto2 && !calib)
72       {
73          // se calcula la distancia entre el punto inicial y el final
74          double x, y;
75          x = abs(xi - xf);
76          y = abs(yi - yf);
77          dist = sqrt(pow(x, 2) + pow(y, 2));      // distancia en pixeles
78          disp.display(img).set_title("Calibrar microscopio: distancia de %4.0f
    pixeles (enter para salir, click derecho para repetir)", dist);
79          img = DibujaPunto(img, xi, yi, green);
80          img = DibujaPunto(img, xf, yf, green);
81          img.draw_line(xi, yi, xf, yf, green);
82          disp.wait();
83          // se acepta el segmento introducido
84          if (disp.is_keyENTER())
85          {
86             calib = true;
87             break;
88          }
89          // o se cancela el punto final con el boton derecho del raton
90          else if (disp.button()&2)
91          {
92             pto2 = false;
93             break;
94          }
95       }
96       // si cerramos el display antes de dibujar el segmento
97       while (disp.is_closed())
98       {
99          disp.show();
100         int res = cimg::dialog("Salir", "¿Desea dejar la calibración del
    microscopio?", "Si", "No", 0, 0, 0, 0, false);
101         if(res == 0)
102         {
103            disp.close();
104            // devolvemos 0
105            if (!calib)
106               dist = 0.0;
107            calib = true;
108            break;
109         }
110      }
111   }
112   // devolvemos la longitud del segmento
113   return dist;
114}
```

En el bucle principal vemos varios bucles while anidados, en los que cada uno soporta una parte del procedimiento. Para dibujar el segmento, primero se hace clic en el primer punto, correspondiendo esto al primer while de la línea 32. Después, ese primer punto queda fijado, y se va dibujando una línea flotante de color rojo que sigue los movimientos del cursor del ratón por la pantalla, hasta que se vuelve a hacer clic en el punto final. La línea flotante implica la parte más compleja de este código entrando en el bucle while de la línea 47. Y una vez que tenemos los dos puntos fijados, queda

dibujado el segmento en color verde sobre la imagen, sucediendo esto en el bucle while de la línea 71, y calculándose la distancia concretamente en la línea 77. Este método está bien implementado, de manera que se puede retroceder para repetir las posiciones de los puntos del segmento, y todo con el ratón, siendo el botón izquierdo sinónimo de avance, y el derecho de retroceso.

2.8.4 DibujaPunto.cpp

Esta función no es de gran utilidad, siendo su valor más que nada estético, pero es un buen ejercicio de familiarización con las herramientas que la librería CImg nos ofrece. Básicamente hace que cuando elegimos un punto de la imagen al calibrar para dibujar nuestro segmento, en vez de dibujar una simple recta, en los extremos donde se hace clic con el ratón, en los puntos inicial y final, se representa un punto más visible, con puntos alrededor que hacen más agradable el aspecto del programa.

```
1  #include "stdafx.h"
2  #include "CImg.h"
3  #include <iostream>
4
5  using namespace cimg_library;
6
7  // dibuja un punto grande en la imagen image, en la posicion (x,y), y de color color
8
9  //    |-|x|x|x|-|
10 //    |x|-|-|-|x|       12 puntos
11 //    |x|-|o|-|x|       alrededor
12 //    |x|-|-|-|x|       del punto
13 //    |-|x|x|x|-|
14
15 CImg<unsigned char> DibujaPunto(CImg<unsigned char> image, int x, int y, const
   unsigned char *color)
16 {
17    int pto[12][2] = {{x - 1, y - 2}, {x, y - 2}, {x + 1, y - 2},
18              {x - 2, y - 1}, {x + 2, y - 1}, {x - 2, y},
19              {x + 2, y}, {x - 2, y + 1}, {x + 2, y + 1},
20              {x - 1, y + 2}, {x, y + 2}, {x + 1, y + 2}};
21    for (int i = 0; i < 12; i++)
22       image.draw_point(pto[i][0], pto[i][1], color);
23
24    return image;
25 }
```

En la siguiente imagen observamos este efecto.

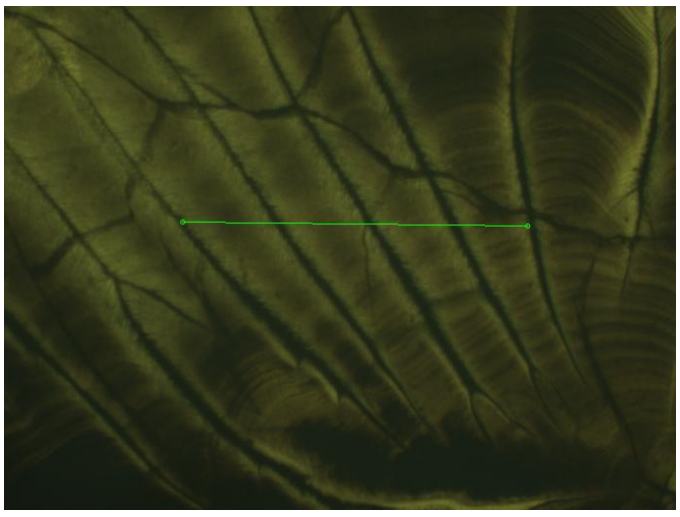

Figura 27.- Segmento medido sobre una imagen real de un otolito