Informe

Trabajo 1: Primera Parte, Algoritmos de Detección de Patrones de Calibración de Círculos y Anillos

Índice

Índice 1
I. Patrón de Círculos 2
1. Implementación con Funciones Propias de OpenCV 2
1.1. Proceso de Avance 2
1.2. Descripción del Algoritmo 2
1.3. Resultados 3
2. Implementación del Algoritmo de Detección de Patrón Desarrollado 5
2.1. Proceso de Avance 5
2.2. Descripción del Algoritmo 6
2.3. Resultados 9
3. Comparación de Implementaciones 10
II. Patrón de Anillos 11
1. Implementación del Algoritmo de Detección de Patrón Desarrollado 11
1.1. Evolución del Código 11
1.2. Descripción del Algoritmo 11
1.3. Resultados 14
III. Como Correr los Programas 16

Trabajo 1: Primera Parte, Algoritmos de Detección de Patrones de Calibración de Círculos y Anillos

I. Patrón de Círculos

1. Implementación con Funciones Propias de OpenCV

Como los requerimientos del trabajo lo solicitan, lo primero que se ha realizado es la etapa de detección de los elementos del patrón mediante las funciones que la biblioteca OpenCV ya trae implementadas.

Esta etapa servirá para poder comprar el desempeño del algoritmo que hemos desarrollado y así comprobar la calidad del mismo.

1.1. Proceso de Avance

Todos los códigos desarrollados han pasado por diferentes etapas para lograr su objetivo.

A continuación detallamos los distintos avances del desarrollo de las funcionalidades implementadas para esta etapa de trabajo:

09/12/2016: Inicio de las investigaciones sobre detección de padrones de círculos, en esta etapa todos los miembros del equipo investigaron, inicialmente de manera individual y después de forma conjunta, sobre el proceso de detección de padrones para su utilización en procesamiento de imágenes.

12/12/2016: Es a partir de ahora que comenzamos a realizar las implementaciones de código, el primer objetivo conseguido fue la detección de los centros de los círculos del padrón de calibración.

13/12/2016: Después de que obtuvimos los centros de los círculos, la siguiente etapa consistiría en dibujar las líneas en zigzag que conecta todos los cirulos. Una vez concluida esta parte, ampliamos un poco las funcionalidades del código para permitir que este fuera más versátil y poder realizar la calibración del patrón de tablero de ajedrez con 3 tipos de entradas (Archivo de Video, conjunto de imágenes y mediante la webcam).

1.2. Descripción del Algoritmo

Para la ejecución del sistema se requiere ingresar los parámetros propios del patrón a ser reconocido, tales como la cantidad de filas y columnas que posee, el tipo de patrón que se va a reconocer, entre otros.

Para mejorar la versatilidad del programa estos parámetros deben ser escritos en un archivo xml. El programa implementa una clase que se encarga de leer los parámetros escritos en el archivo xml y realiza una serie de evaluaciones para verificar que todo está correcto y no se

generaran errores en tiempo de ejecución, tales como el que no se encuentre el archivo especificado o que la cantidad de elementos en el patrón no sea válida.

Para el caso de prueba, con los videos patrones de círculos, el archivo xml que contiene, los parámetros específicos tiene el nombre `"in_VID5.xml"`.

Una vez obtenidos todos los parámetros necesarios se empieza con el procesamiento para la detección de los centros de los círculos del patrón según el tipo de patrón que se haya especificado en el archivo xml de configuración.

Por ultimo una vez más, según el tipo de patrón que se haya especificado en el archivo de configuración se aplica la función adecuada para el dibujado de las líneas en zigzag en los centros de los círculos del patrón.

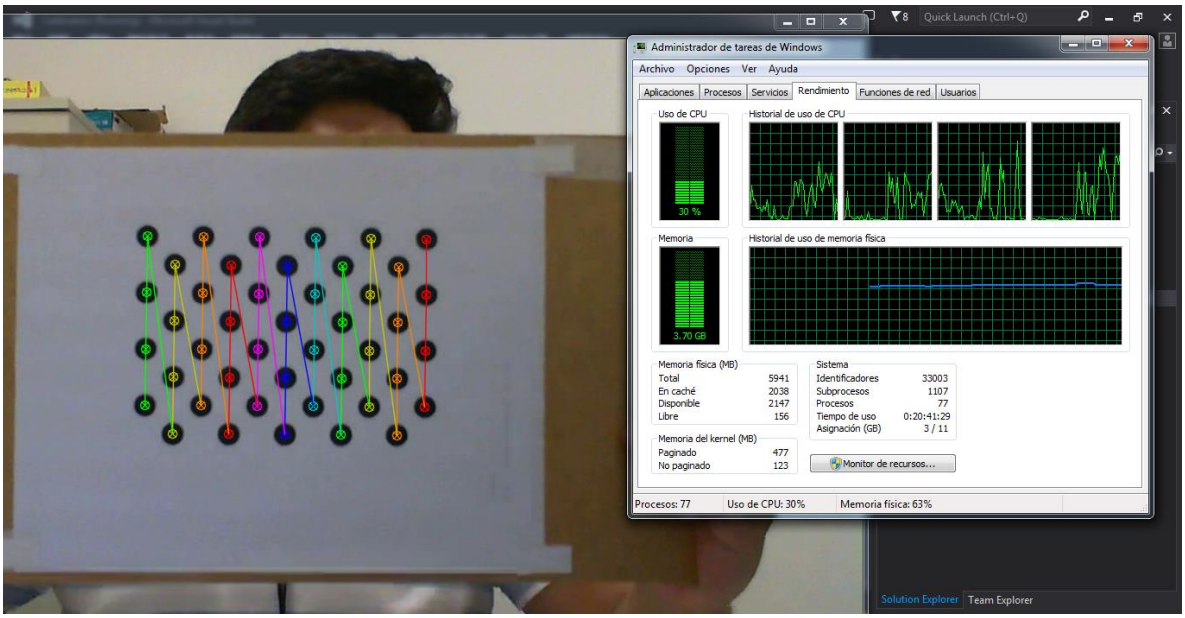
En el archivo IN_VID5.xml se encuentran todos los parámetros de configuración.

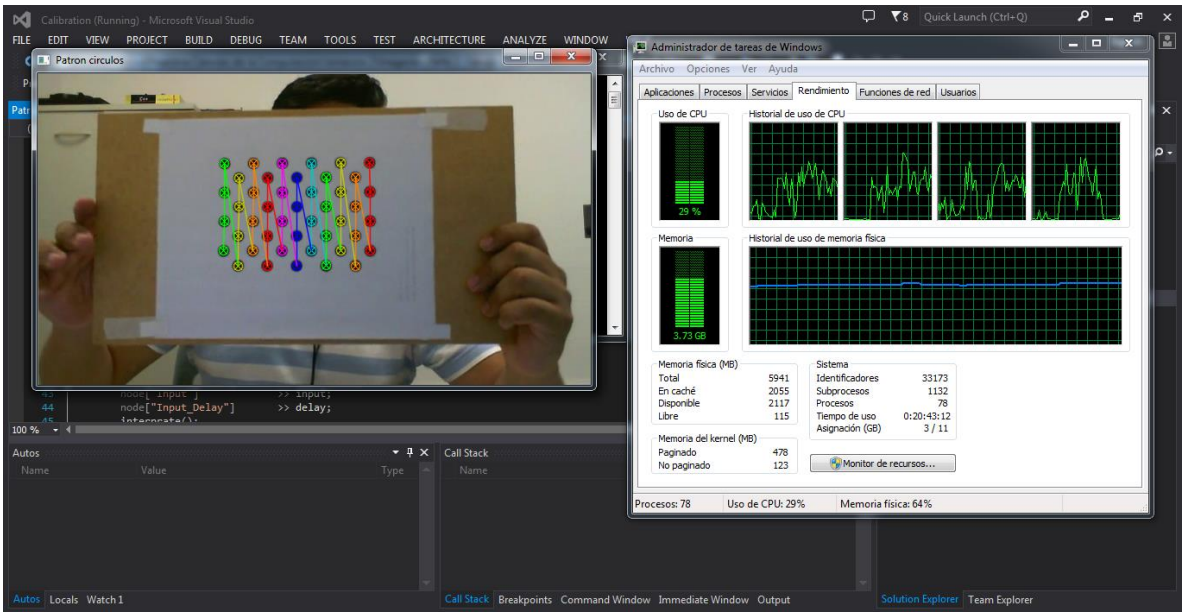
```
<?xml version="1.0"?>
<opencv_storage>
<Settings>
  <!-- numero de esquinas interiores para cada elemento de las filas y las columnas. (Cuadrado, circulo) -->
  <BoardSize_Width>4</BoardSize_Width>
  <BoardSize_Height>11</BoardSize_Height>

  <!-- El tamaño de un cuadrado en algun sistema metrico definido por el usario (pixel, milimetros)-->
  <Square_Size>15</Square_Size>

  <!-- El tipo de entrada usada para la calibracion de la camara. Uno de: CHESSBOARD CIRCLES_GRID ASYMMETRIC_CIRCLES_GRID -->
  <Calibrate_Pattern>"ASYMMETRIC_CIRCLES_GRID"</Calibrate_Pattern>

  <!-- Tipo de entrada
              Webcam -> escribe como entrada  "1"
              Archivo de video  -> escribe como entrada el nombre del archivo ejm "PadronCirculos_01.avi"
              Lista de imagenes   -> escribe como entrada el nombre del xml que contiene las imagenes ejm "VID5.xml"
              -->
  <Input>"PadronCirculos_03.avi"</Input>

  <!-- Tiempo de retraso enre cada frame en el caso de la camara. -->
  <Input_Delay>1</Input_Delay>

  <Calibrate_NrOfFrameToUse>25</Calibrate_NrOfFrameToUse>

</Settings>
</opencv_storage>
```

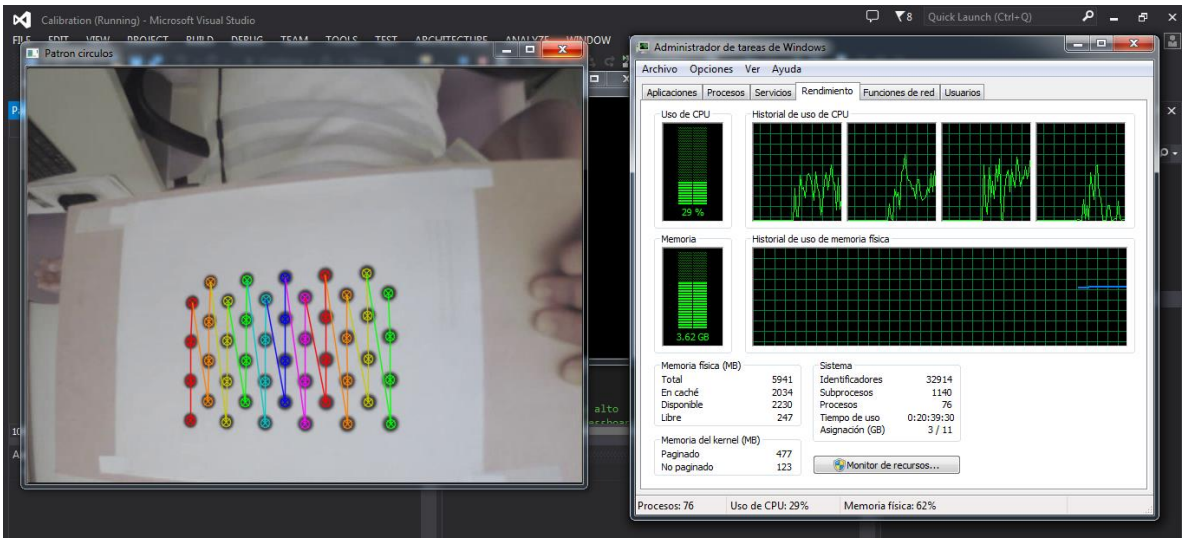
1.3. Resultados

Como se esperaba este código encuentra correctamente todos los puntos del patrón y dibuja adecuadamente las líneas del zigzag. El resultado es bastante preciso y nunca se llegan a perder los puntos.

Sin embargo, el procesamiento es demasiado lento considerando la cantidad de consumo de procesador que realiza, como se puede ver en las siguientes capturas de pantalla.


Administrador de tareas de Windows
Archivo Opciones Ver Ayuda
Aplicaciones Procesos Servicios Rendimiento Funciones de red Usuarios
Uso de CPU
30 %
Historial de uso de CPU
Memoria
3.70 GB
Historial de uso de memoria física
Memoria física (MB)
Total 5941
En caché 2038
Disponible 2147
Libre 156
Sistema
Identificadores 33003
Subprocesos 1107
Procesos 77
Tiempo de uso 0:20:41:29
Asignación (GB) 3 / 11
Memoria del kernel (MB)
Paginado 477
No paginado 123
Monitor de recursos...
Procesos: 77 Uso de CPU: 30% Memoria física: 63%
Solution Explorer Team Explorer


Calibration (Running) - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW
Patron circulos
Administrador de tareas de Windows
Archivo Opciones Ver Ayuda
Aplicaciones Procesos Servicios Rendimiento Funciones de red Usuarios
Uso de CPU
29 %
Historial de uso de CPU
Memoria
3.73 GB
Historial de uso de memoria física
Memoria física (MB)
Total 5941
En caché 2055
Disponible 2117
Libre 115
Sistema
Identificadores 33173
Subprocesos 1132
Procesos 78
Tiempo de uso 0:20:43:12
Asignación (GB) 3 / 11
Memoria del kernel (MB)
Paginado 478
No paginado 123
Monitor de recursos...
Procesos: 78 Uso de CPU: 29% Memoria física: 64%
Autos
Call Stack
Autos Locals Watch 1
Call Stack Breakpoints Command Window Immediate Window Output
Solution Explorer Team Explorer



2. Implementación Propia del Algoritmo de Detección de Padrón de Círculos

Habiendo terminado el desarrollo del programa anterior es momento de pasar a desarrollar un algoritmo propio que realice la detección de los elementos del patrón.

2.1. Proceso de Avance

Esta etapa del desarrollo del trabajo inicio inmediatamente después de haber terminado la anterior.

14/12/2016: El primer objetivo a alcanzar en este momento era el de poder encontrar todos los centros de los círculos y aislarlos del ruido contenido en el video para evitar encontrar falsos positivos que pasasen como centros cuando no lo son. Por lo que empezamos desarrollando la heurística necesaria para poder conseguirlo.

17/12/2016: Una vez conseguidos los centros de cada círculo elemento del patrón debíamos encontrar la manera de encontrar el orden de los puntos para poder realizar el dibujado del zigzag.

19/12/2016: Ya habiendo elaborado del algoritmo necesario para conseguir el orden de los centros de los círculos procedemos a desarrollar el código para el dibujado del zigzag.

20/12/2016: Después de haber dialogado y mostrado los avances respectivos al profesor Manuel Loaiza, nos puso en aviso de que habíamos cometido algunos errores, los que pasaríamos a corregir de inmediato.

20/12/2016: Después de descartar el código erróneo que indicado por el profesor pasamos al desarrollo algoritmo adecuado siguiendo las sugerencias del que se nos habían dado. Para poder llegar a la implementación de un programa adecuado tuvimos que hacer diferentes modificaciones a cada código que se realizaba basándonos en diferentes técnicas que creíamos adecuadas. Logrando finalmente desarrollar un código lo más refinado posible para,

tanto para la parte de la detección de los centros, como encontrar el orden de os mismo y finalmente poder dibujar de manera adecuada lo más adecuada posible el zigzag en el video.

Cabe destacar que este código nos resultó más difícil de elaborar que el del patrón de anillos, el cual pudimos hacer funcionar antes; el mismo que se desarrolla en la parte II de este documento.

28/12/2016: La mejora, consistió en obtener el zigzag de manera robusta. Si bien eliminábamos todo el ruido de la imagen y nos quedábamos con el padrón, el zigzag era inestable. Esto se realizó obteniendo las diagonales del padrón, el cual es explicado en mayor detalle en la descripción del algoritmo.

2.2. Descripción del Algoritmo

La heurística del algoritmo desarrollado se divide en 2 parte.

Parte 1. Detectar el padrón de círculos.

En esta primera parte debemos detectar los 44 círculos del padron y eliminar todo el ruido.

Para poder obtener los centros de los cirulos se necesita hacer una detección de borde, para ello, primeramente se aplica un filtro gaussiano (*Gaussian Blur*) para suavizar la imagen y después de estos se aplica un detector de Canny con lo que se obtienen todos los bordes presentes en la escena.

Una vez que se tienen todos los bordes, se aplica la función "*findCountours*" para poder detectar todos los contornos de la imagen.

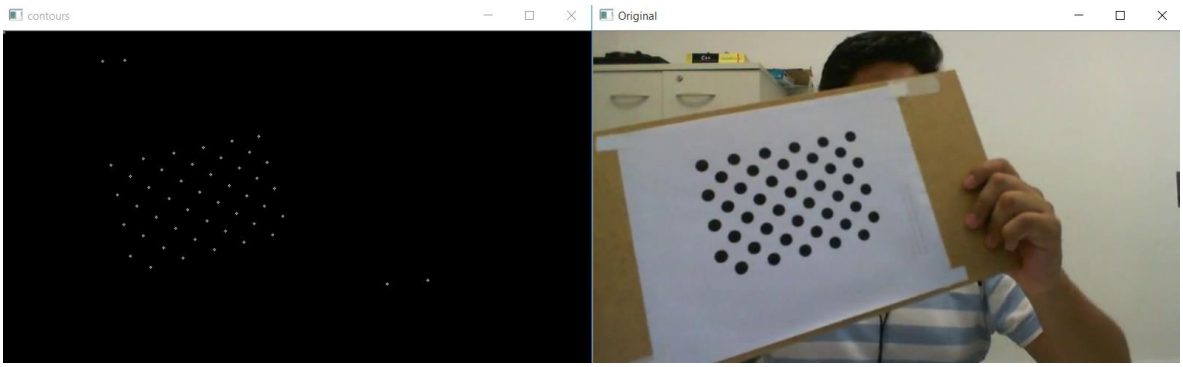
A continuación se deben aislar todos aquellos contornos que no formen parte de los elementos del padrón (demás bordes en la escena y ruido), para ello calculamos el área de los contornos. En base a varias pruebas llegamos a la conclusión de que todos aquellos contornos que tengan un área menor a 1 o mayor a 10000 no son las elipses que pertenecen al patrón de círculos y no deben ser consideradas.

Para cada contorno aplicaremos la función "fitEllipse" con esta función podemos obtener el **centro de la elipse** que contiene al contorno, asimismo se otiene el alto y el ancho del rectángulo que la circunscribe, y se verifica que la elipse no tenga un área mayor a 1500 ni que la diferencia del ancho y el alto sea mayor a 12. Esto con el fin de evitar considerar una elipse demasiado grande que se haya podido detectar erróneamente asi como alguna elipse muy distorsionada (con un largo mucho mayor al ancho).

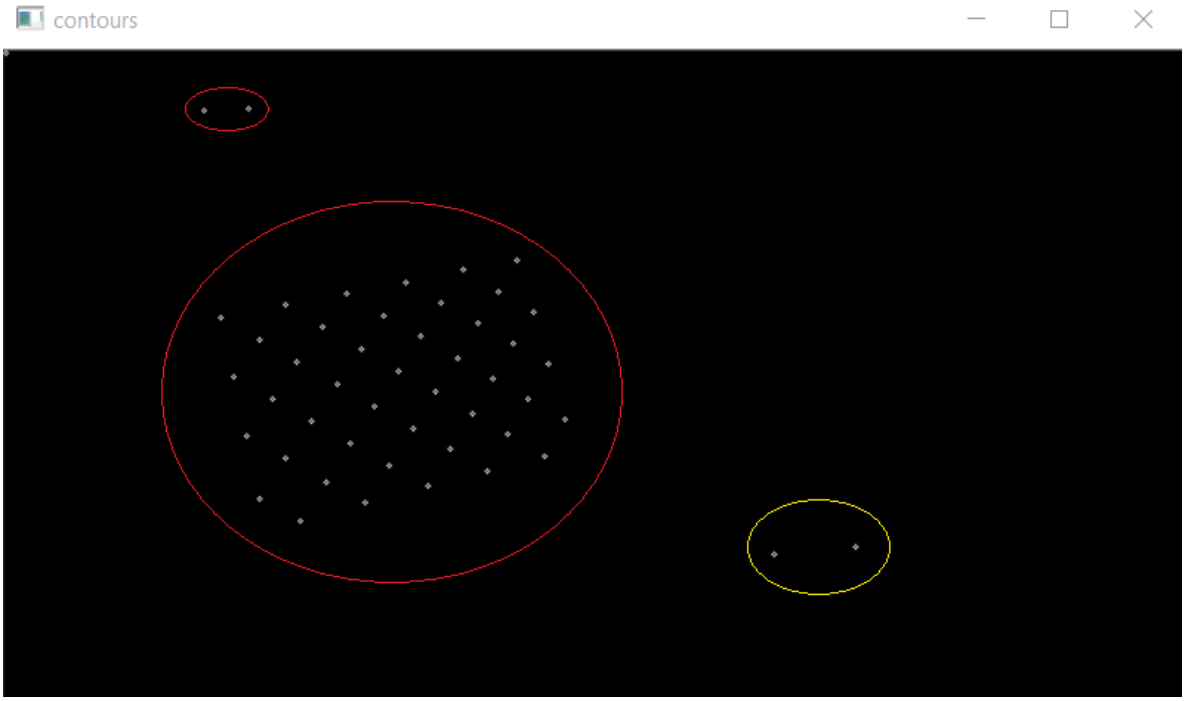
Para el padron en uso sabemos que debemos tener 44 contornos y después del paso anterior se tienen muchos de ellos. Para poder obtener los 44 que necesitamos buscamos la componente conexa más grande dada una determinada distancia permitiendo así aislar las 44 elipses propias del patrón.

A partir de este punto, solo utilizaremos los **centros de cada contorno**, el cual debemos aislar exactamente los 44 que pertenecen al padron.

Ejm. En caso de obtener lo siguiente:



Observamos que los 44 puntos del padrón están muy cercanos entre sí, el ruido está más alejado del padrón y este aparece en menor cantidad. Esto lo aprovecharemos para aplicar la siguiente estrategia.



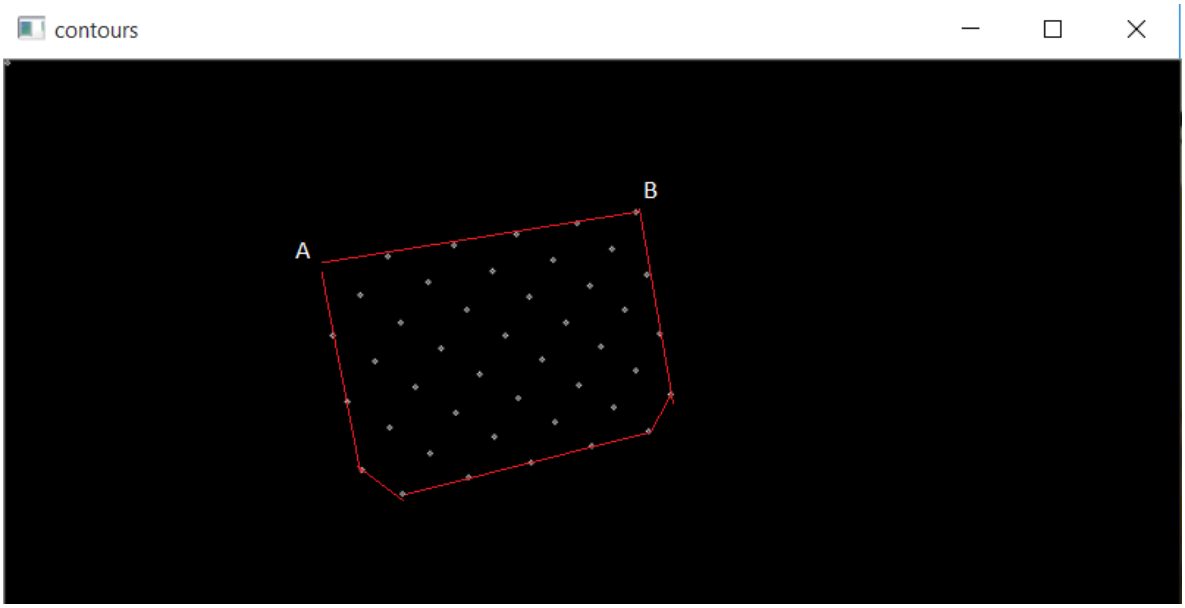
Hay 3 componentes conexas. Tamaños 2, 2 y 44.
Observamos que la componente más grande el cual es de tamaño 44 corresponde al padron
Para obtener la componente conexa más grande es decir la del padrón, vamos a conectar los centros con los vecinos que se encuentren a una distancia menor o igual a **D.** Esto lo realizamos mediante un algoritmo como DFS o BFS, busquedad en profundidad o anchura**.**

Para encontrar el valor óptimo de **D** observamos lo siguiente:
Si D es muy pequeño cada centro formara parte de una componente conexa de tamaño 1.
Si D es muy grande solo existirá una componente conexa para el ejemplo de tamaño 48=44+2+2.
Por lo tanto aquí observamos que para encontrar el valor de D utilizaremos una búsqueda binaria, puesto que mientras más grande sea la componente conexa más grande tendrá un mayor tamaño.

De lo anterior podemos obtener el padrón de 44 círculos de manera robusta.

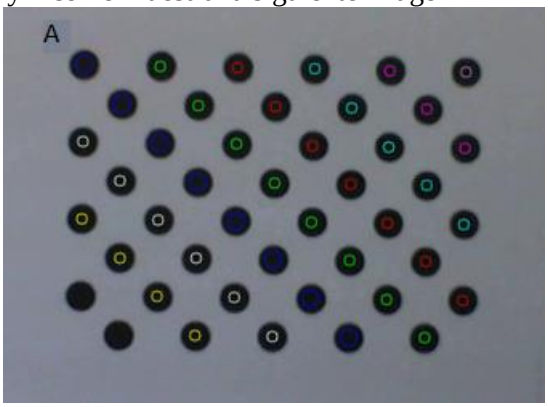
Parte 2. Realizar el zigzag del padrón de círculos.

En este punto obtenemos los 44 centros del padrón de manera desordenada.
Lo que realizaremos es obtener el Convex Hull de los 44 puntos.
Se obtendrá aproximadamente el polígono en rojo del grafico mostrado.



Notamos que el punto A y B tienen ángulos de 90 grados o cercanos a estos.
Encontraremos esto mediante el producto vectorial ya que (|PxQ|=|P||Q|Seno(angle))
Todo esto lo haremos para encontrar el punto A el cual es el que posee menor eje en X

Ahora nuestro enfoque se centrara en detectar las diagonales del padron, los cuales son de tamaño 8,8,7,6,5,4,3,2 y 1 como muestra la siguiente imagen



Paso 1: Lo que haremos en un comienzo es encontrar un segmento que contenga 8 centros (Existen 4 segmentos que cumplen esta condición, pero añadiremos algo adicional para encontrarlo de manera única, el segmento debe contener **el punto A** que encontramos anteriormente.
Paso 2: Luego de esto estos 8 puntos lo marcamos para que no puedan ser visitados en las siguientes iteraciones. (En este primer paso marcamos el segmento de color azul).

Paso 3: Ahora buscaremos otro segmento que contenga 8 centros. (Como ya hemos borrado el segmento azul) ahora solo existirá un segmento con esta condición este será el de color verde.
Paso 4: Análogamente haremos lo mismo con el segmento de tamaño 7,6,5,4,3.
Paso 5: Para el caso del segmento de tamaño 2, serán de los 3 puntos faltantes, los dos puntos que están más cercanos será el segmento de tamaño 2.
Paso 6: Finalmente quedara un solo punto el cual pertenecerá al segmento de tamaño 1.

Observación: En el paso 1. Encontrar un segmento que contenga 8 centros, el procedimiento es el siguiente. Fijar 2 puntos los cuales serán el extremo del segmento, los demás puntos no borrados guardaremos **la distancia del punto al segmento** y nos quedaremos con los 6 puntos que tuvieron menor distancia. (Cabe mencionar de que si un punto se encuentra en el segmento esta distancia debería ser 0 pero por errores de precisión o distorsion esto no será el caso por ello nos quedamos con los que tienen menor distancia.

Con lo anterior, tenemos 9 segmentos lo ordenaremos de abajo hacia arriba mediante producto vectorial. Cada segmento se ordenará de menor a mayor x.

Lo que debemos hacer es realizar el zigzag en vertical.
Para esto visitaremos de abajo hacia arriba cada segmento y visitaremos los menores x, luego de esto lo marcaremos como visitado.

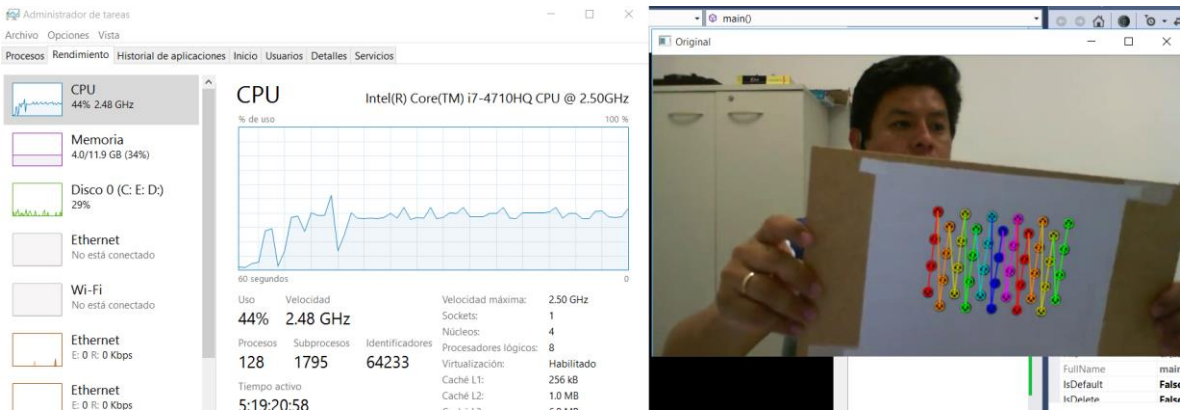
Una vez se hayan ordenados todos los elementos del patrón, solo resta dibujar el zigzag, que se logra usando la función "*drawChessboardCorners*".

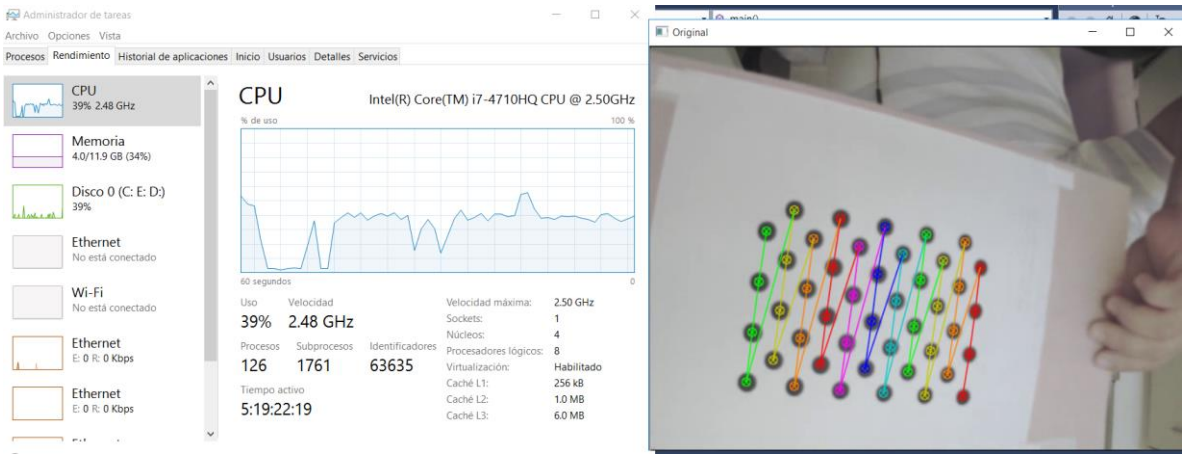
2.3. Resultados

La heurística desarrollada ha sido corrida con los tres videos de prueba, y en todos ellos ha dado podido encontrar todos los centros y dibujar el zigzag con bastante fluidez y con un consumo de procesador moderado.

Hemos utilizado OpenMP para optimizar los tiempos de respuesta

En las siguientes imágenes se muestra los resultados mencionados.





3. Comparación de Implementaciones

Como los requerimientos del trabajo lo requieren, una vez terminados de desarrollar los dos programas (uno con implementaciones ya funcionales de OpenCV y otro con heurística propia), debemos realizar la comparación entre ambos códigos.

Como se ha podido ver en los resultados obtenidos con ambos códigos podemos en las secciones anteriores, ambos códigos tienen diferencias considerables.

En la siguiente tabla resumimos las diferencias entre ambos:

Código con Funciones de OpenCV	**Código con heurística propia**
1. Realiza el dibujado del zigzag sin perder el centro de los círculos en ningún momento ni deformar el patrón de las líneas.	1. Realiza el dibujado del zigzag adecuadamente, aunque en algunos momentos se pierde el patrón de líneas y algunas otras se deforma un poco.
2. El procesamiento realizado s demasiado pesado por lo que resulta en una ejecución exageradamente lenta.	2. El procesamiento es muchísimo más ligero y se ejecuta, por ende, mucho más rápido.

En conclusión podemos decir que el algoritmo que hemos desarrollado en mejor que el que se puede implementar con las funciones propias de OpenCV.

En los videos que se encuentran en el drive compartido se puede apreciar con mayor claridad lo explicado en esta sección.

III Patrón de Anillos

3.1Implementación del Algoritmo de Detección de Patrón Desarrollado

3.2 Evolución del Código

Antes de comenzar a describir como se desarrolló esta etapa cabe mencionar que el programa para la detección de este patrón se concretó aproximadamente 1 día antes (22 en la madrugada) que el del patrón anterior, tanto antes como después de las correcciones realizadas por el profesor Manuel Loaiza que fueron de mucha utilidad.

14/12/2016: Como se mencionó en el desarrollo del patrón de círculos, el primer objetivo era el de poder encontrar todos los centros de los círculos y aislarlos del ruido contenido en el video.

17/12/2016: Una vez habiendo podido obtener los centros de cada anillo del patrón tuvimos que encontrar el orden de los puntos para poder realizar el dibujado del zigzag.

19/12/2016: Ya habiendo conseguido el orden de los centros de los círculos procedimos a desarrollar el código para el dibujado del zigzag.

20/12/2016: Fue en este día que conversamos con el profesor Manuel Loaiza, y como ya se mención en el patrón de circulo, se nos indicó que nuestros código tenían errores.

20/12/2016: En este punto debemos indicar que las fechas de desarrollo son las mismas que las del patrón anterior debido a que desarrollábamos los códigos casi en paralelo, es decir que íbamos planteando las ideas para el desarrollo del patrón de anillos y cada vez que teníamos algún avance significante pasábamos a pensar y probar como podíamos extender ese funcionamiento al patrón de círculos, teniendo siempre en cuenta que las características entre ambos son considerablemente diferentes.

Como ya se dijo en la sección 2.1 de la parte I del documento, para poder llegar a nuestra implementación final, nuestro código paso por diferentes etapas mediante la implementación de los diversos métodos que creíamos adecuados.

28/12/2016: La mejora consistió en paralelizar nuestro algoritmo con OpenMP para mejorar tiempos de respuesta, asimismo al igual que en el padrón de círculos afinamos la detección del padrón aplicando búsqueda binaria + BFS (El cual es explicado con mayor detalle en la descripción del algoritmo)

3.3 Descripción del Algoritmo

La heurística del algoritmo desarrollado se divide en 2 parte.

<u>Parte 1. Detectar el padrón de anillos.</u>

En esta primera parte debemos detectar los 30 anillos del padron y eliminar todo el ruido.

Para poder obtener los centros de los anillos se necesita hacer una detección de borde, para ello, primeramente se aplica un filtro gaussiano (*Gaussian Blur*) para suavizar la imagen y después de estos se aplica un detector de Canny con lo que se obtienen todos los bordes presentes en la escena.

Una vez que se tienen todos los bordes, se aplica la función "*findCountours*" para poder detectar todos los contornos de la imagen.

A continuación se deben aislar todos aquellos contornos que no formen parte de los elementos del padrón (demás bordes en la escena y ruido), para ello calculamos el área de los contornos. En base a varias pruebas llegamos a la conclusión de que todos aquellos contornos que tengan un área menor a 1 o mayor a 10000 no son las elipses que pertenecen al patrón de círculos y no deben ser consideradas.

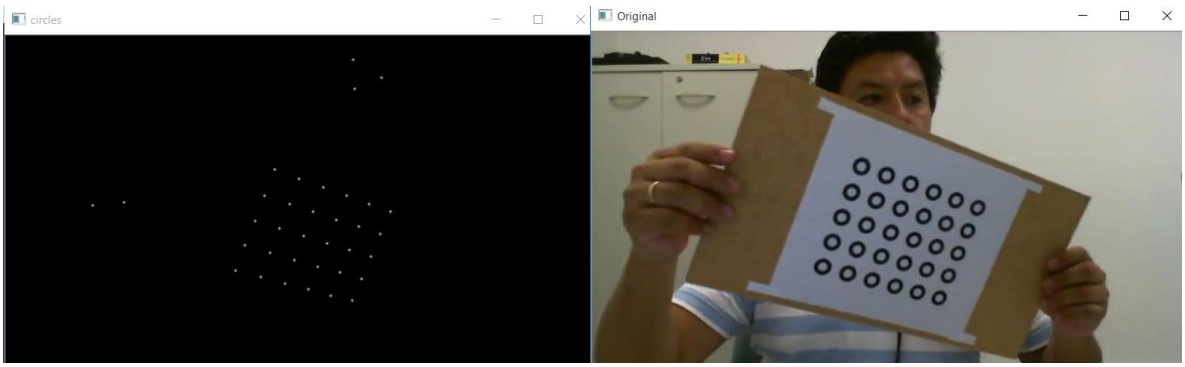
Para cada contorno aplicaremos la función "fitEllipse" con esta función podemos obtener el **centro de la elipse** que contiene al contorno, asimismo se otiene el alto y el ancho del rectángulo que la circunscribe, y se verifica que la elipse no tenga un área mayor a 1500 ni que la diferencia del ancho y el alto sea mayor a 12. Esto con el fin de evitar considerar una elipse demasiado grande que se haya podido detectar erróneamente asi como alguna elipse muy distorsionada (con un largo mucho mayor al ancho).

A diferencia del padron de circulos ahora tenemos los centros del anillo interior y superior. Por lo tanto tendremos 2 centros ubicados en la misma posición o a un pixel de distancia. Si encontramos 2 centros menor igual a 1.5 de distancia será considerado como candidato valido para el padron.

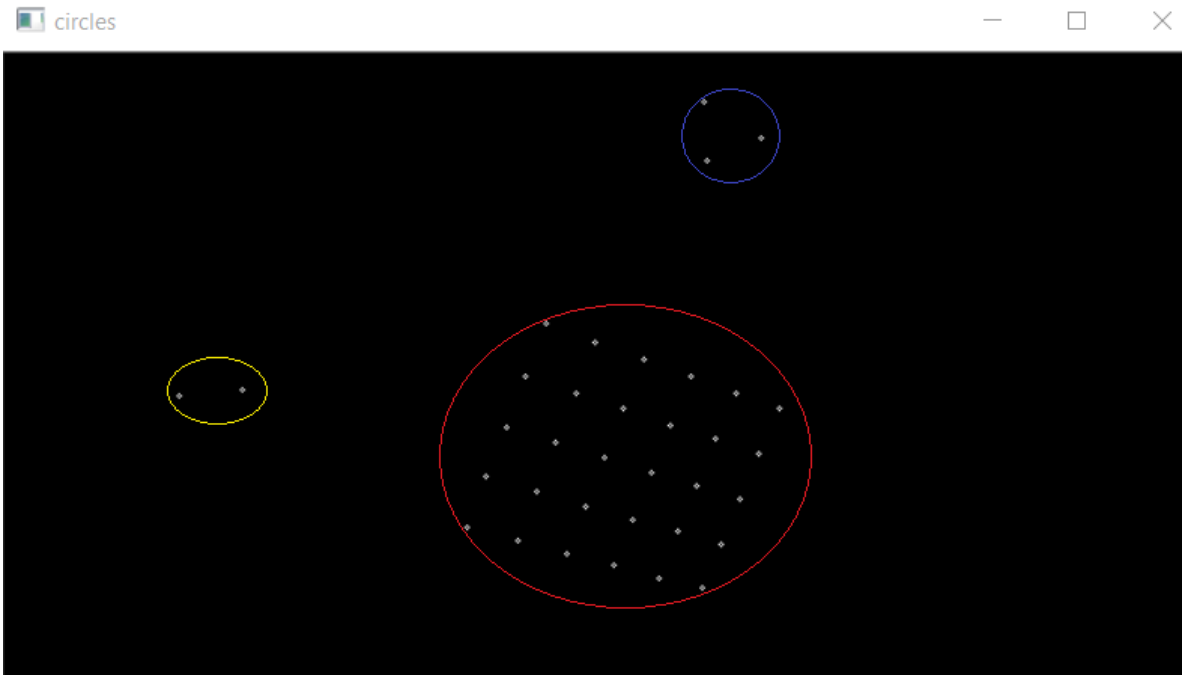
Para el padron en uso sabemos que debemos tener 30 contornos y después del paso anterior se tienen muchos de ellos. Para poder obtener los 30 que necesitamos buscamos la componente conexa más grande dada una determinada distancia permitiendo así aislar las 30 elipses propias del patrón.

A partir de este punto, solo utilizaremos los **centros de cada contorno**, el cual debemos aislar exactamente los 30 que pertenecen al padron.

Ejm. En caso de obtener lo siguiente:



Observamos que los 30 puntos del padrón están muy cercanos entre sí, el ruido está más alejado del padrón y este aparece en menor cantidad. Esto lo aprovecharemos para aplicar la siguiente estrategia. (Al igual que en el caso de circulos)



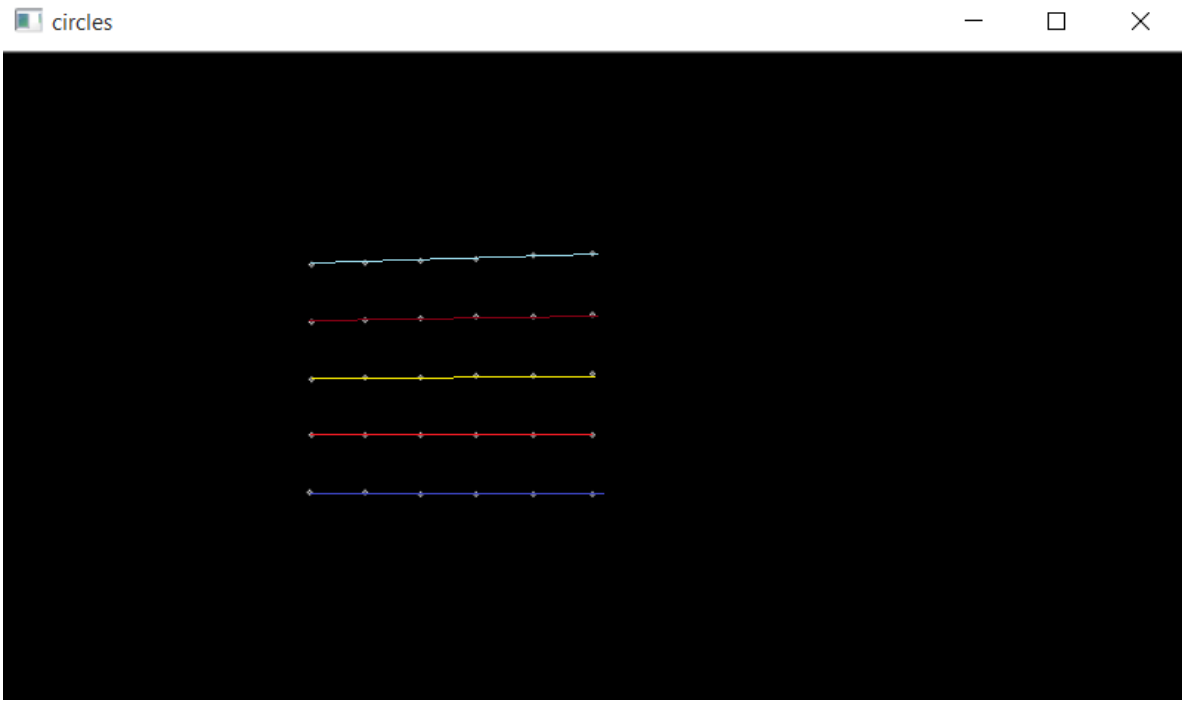
Hay 3 componentes conexas. Tamaños 2, 3 y 30.
Observamos que la componente más grande el cual es de tamaño 30 corresponde al padron
Para obtener la componente conexa más grande es decir la del padrón, vamos a conectar los centros con los vecinos que se encuentren a una distancia menor o igual a **D.** Esto lo realizamos mediante un algoritmo como DFS o BFS, busquedad en profundidad o anchura**.**

Para encontrar el valor óptimo de **D** observamos lo siguiente:
Si D es muy pequeño cada centro formara parte de una componente conexa de tamaño 1.
Si D es muy grande solo existirá una componente conexa para el ejemplo de tamaño 35=30+2+3.
Por lo tanto aquí observamos que para encontrar el valor de D utilizaremos una búsqueda binaria, puesto que mientras más grande sea la componente conexa más grande tendrá un mayor tamaño.
De lo anterior podemos obtener el padrón de 30 anillos de manera robusta.

<u>Parte 2. Realizar el zigzag del padrón de anillos</u>

Para el zigzag de anillos es más sencillo de realizar que el de círculos.
Lo que haremos es encontrar 5 segmentos que contengan 6 puntos.

Tal como se muestra en la gráfica.



A diferencia del padrón de círculos solo existen esas rectas de tamaño 6 (es decir contienen 6 puntos). Las diagonales son de tamaño 5 por ello no generan conflicto, por lo tanto no es necesario aquí realizar el convex hull.

Paso 1: Fijamos 2 puntos los cuales serán el extremo del segmento, para todos los demás puntos no visitados hallaremos **la distancia del punto al segmento** y nos quedaremos con los 4 puntos que tuvieron menor distancia. . (Cabe mencionar de que si un punto se encuentra en el segmento esta distancia debería ser 0 pero por errores de precisión o distorsion esto no será el caso por ello nos quedamos con los que tienen menor distancia)

Paso 2: Del paso anterior hemos obtenido un segmento que contiene 6 puntos. Estos puntos serán marcados como visitados y nuevamente repetimos el paso 1. (Esto se realizara hasta que obtengamos los 6 segmentos del grafico anterior.

Con lo anterior, tenemos 6 segmentos lo ordenaremos de abajo hacia arriba mediante producto vectorial. Cada segmento se ordenará de menor a mayor x.

Lo que debemos hacer es realizar el zigzag en vertical.
Para esto visitaremos de abajo hacia arriba cada segmento y visitaremos los menores x de cada segmento horizontal para poder posicionarlo en forma vertical tal como se muestran en los resultados (3.4)

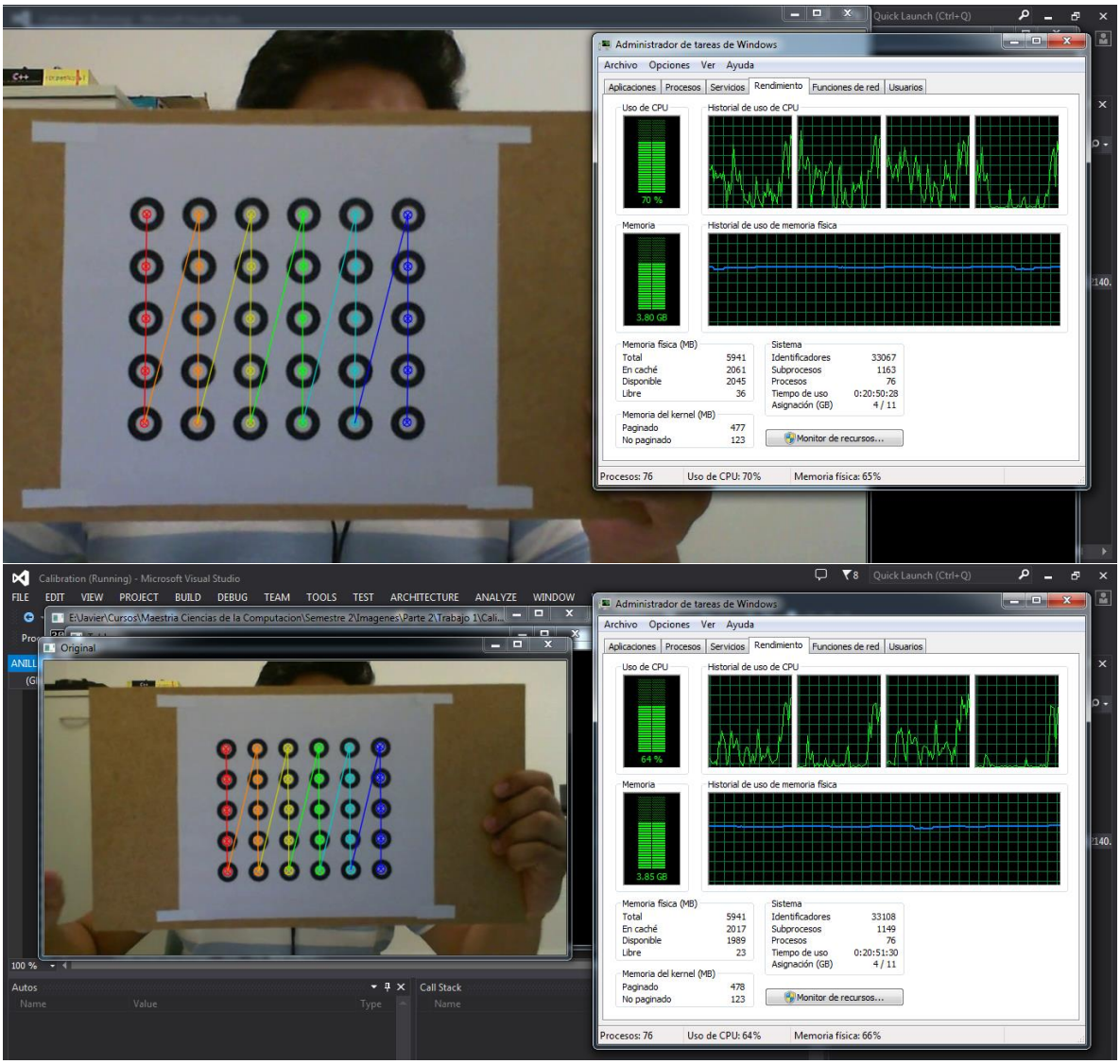
Una vez se hayan ordenados todos los elementos del patrón, solo resta dibujar el zigzag, que se logra usando la función "*drawChessboardCorners*".

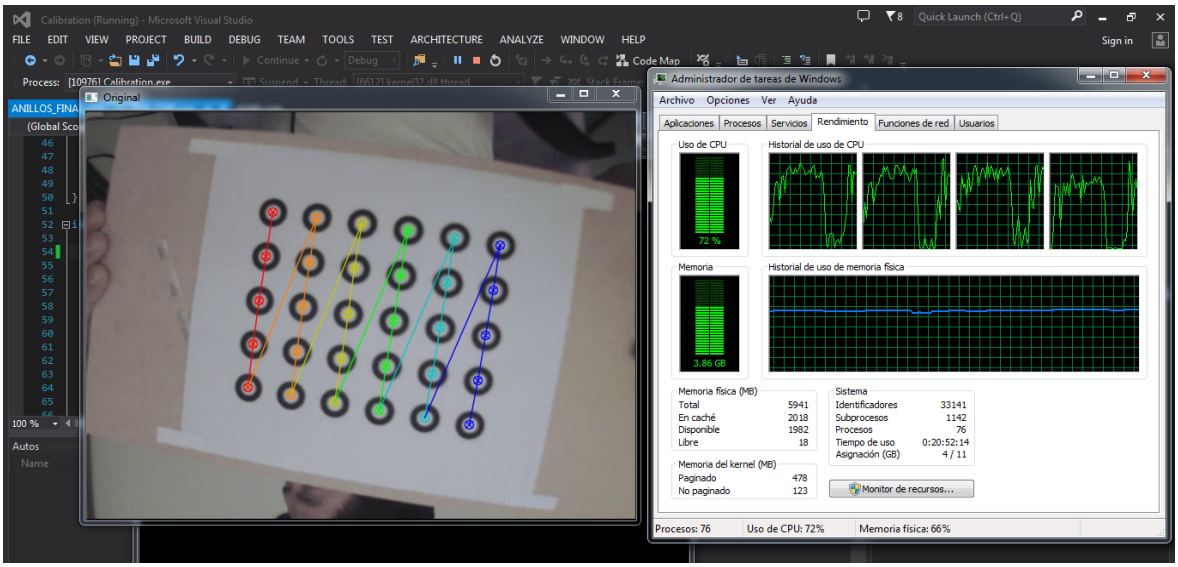
3.4 Resultados

El algoritmo que hemos desarrollado para la detección de los elementos de este patrón ha resultado ser bastante eficiente y robusto. Se consigue obtener todos los centros de los anillos y el dibujado del zigzag es constante casi todo el tiempo.

Este algoritmo corre con muy buena fluidez y hace un consumo de procesador relativamente alto.

Las imágenes mostradas a continuación mejoran la idea de lo descrito.





4 Como Correr los Programas

Los tres programas que se desarrollaron pueden ser obtenidos como archivos .cpp, para poder correrlos deberá realizar lo siguiente:

1. Crear un proyecto en Visual Studio 2013 Community o posterior.

2. Preparar (linkear) el proyecto para que trabaje con la biblioteca OpenCV 2.4.13 de preferencia y activar la compatibilidad con OpenMP.
   Configurar OpenMP en Visual Studio. (Properties-> C/C++ ->Language -> OpenMP Support -> Yes)

3. Acceder al enlace:

   https://github.com/jonathandrnd/Imagenes/tree/master/CalibracionTrabajo1

   Descargar los archivos "*in_VID5.xml*", "*Calibration.cpp*", "*Patterncircle.cpp*" y "*PatternGrid.cpp*", así como los videos para su prueba.

4. Copiar dentro de la carpeta que está en la misma ubicación y tiene el mismo nombre del archivo .sln que es el proyecto que se ha creado, los archivos .cpp, y los videos .avi

5. Agregar al proyecto el archivo del código que se desee hacer correr.

6. En el caso de que se vaya a correr el programa "*Calibration.cpp*", abrir el archivo "*in_VID5.xml*", y en el tag "*<Input>*" escribir el nombre del archivo que se desea probar en lugar del que ya está escrito.

Si se va a ejecutar cualquiera de los otros 2 programas, en el código, en la función "*main*", buscar la inicialización del objeto *"inputCapture"* y escribir el nombre del video que se desea probar en vez del que se encuentra escrito.

7. Compilar y ejecutar el código.

8. Repetir los pasos 5 y 6 con cada video que se quiera probar.

9. Para correr otro código solo se debe remover del proyecto el anterior y agregar el que se desee hacer funcionar, compila y ejecutar.