Informe

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

Índice

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 calibración de cámaras, en esta etapa todos los miembros del equipo investigaron, inicialmente de manera individual y después de forma conjunta, sobre el proceso de calibración de una cámara 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 patrón de calibración.

13/12/2016: Después de que hubimos obtenido 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 o que se pudiera tener como entrada la cámara.

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"`.

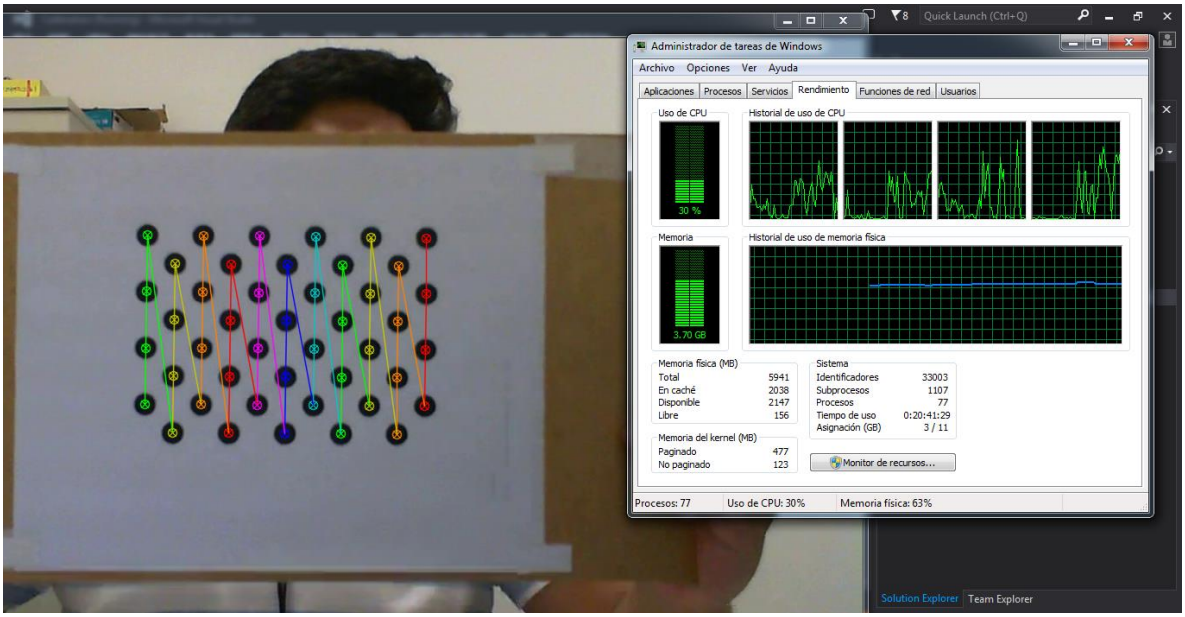
Una vez obtenidos todos los parámetros necesarios se empieza con el procesamiento para la detección de los centros de los círculos del patrón según el tipo de patrón que se haya especificado en el archivo xml de configuración.

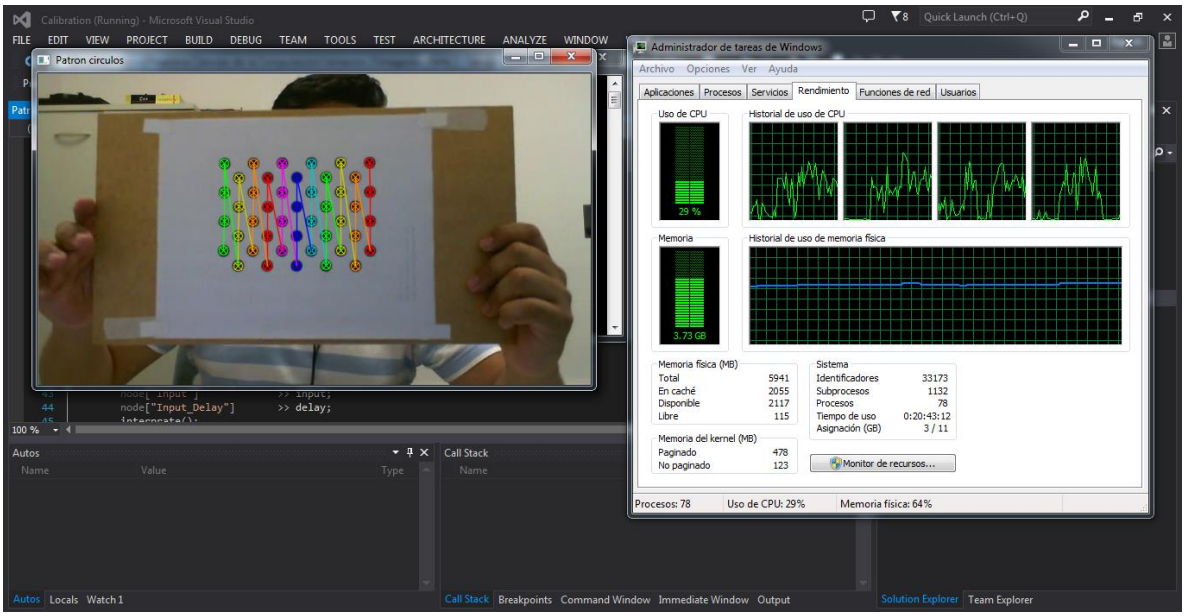
Por ultimo una vez más, según el tipo de patrón que se haya especificado en el archivo de configuración se aplica la función adecuada para el dibujado de las líneas en zigzag en los centros de los círculos del patrón.

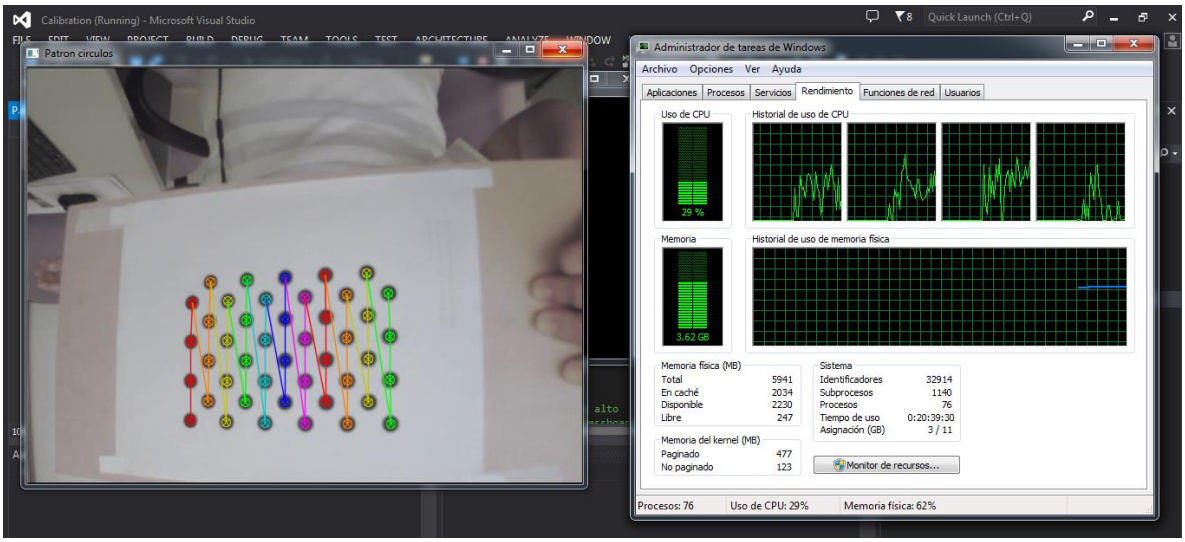
1.3. Resultados

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.







2. Implementación del Algoritmo de Detección de Patrón Desarrollado

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 pasmos 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.

2.2. Descripción del Algoritmo

La heurística del algoritmo desarrollado funciona de la siguiente manera.

Debemos tener en cuenta la cantidad de elementos del patrón que se va a procesar porque de esto depende parte del algoritmo desarrollado. Para nuestro caso son 44 círculos.

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 bordes que sean continuos, permitiendo así detectar las elipses que pertenecen a los círculos.

A continuación se deben aislar todos aquellos contornos que no formen parte de los elementos del patrón (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.

Ya habiendo obtenido todas las elipses propias de los círculos del patrón y habiéndolas separado del todo aquello que no es un elemento del patrón, aplicamos la función "*fiteEllipse*"

a cada uno de los contornos obtenidos, con lo cual se ajustan a una elipse, obteniendo así los centros de todas la elipses presentes en el patrón.

Para cada elipse después de aplicar "*fiteEllipse*" se obtiene 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.

Para el patrón 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.

Una vez hecho esto aplicamos la función "*convexHull*" para obtener los elementos que están en el contorno del patrón.

Con lo anterior, usando un concepto de algebra lineal, aplicamos el producto vectorial entre las rectas verticales con lo que podemos ordenar los elementos para poder dibujar e zigzag.

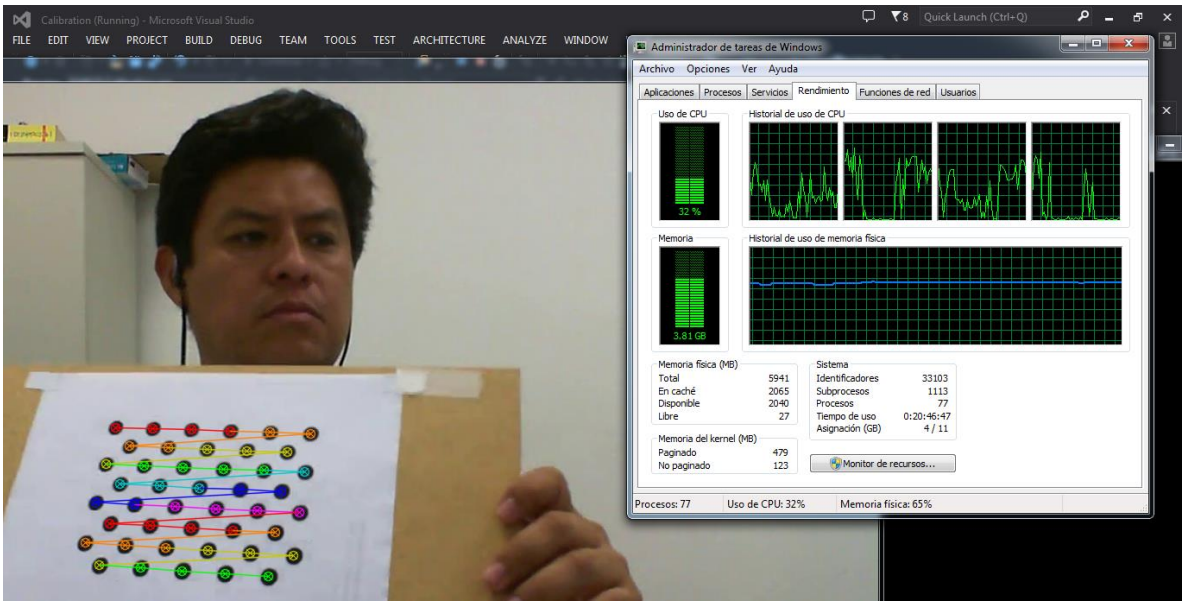
Una vez se hayan ordenados todos los elementos del patrón, solo resta dibujar el zigzag, que se logra usando la función "*drawChessboardCorners*".

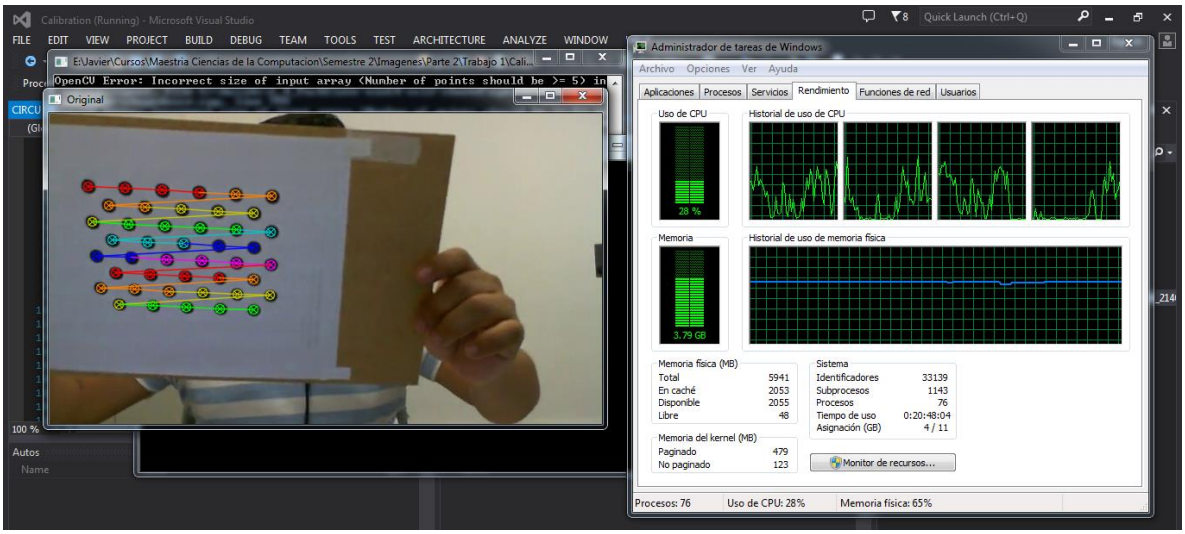
2.3. Resultados

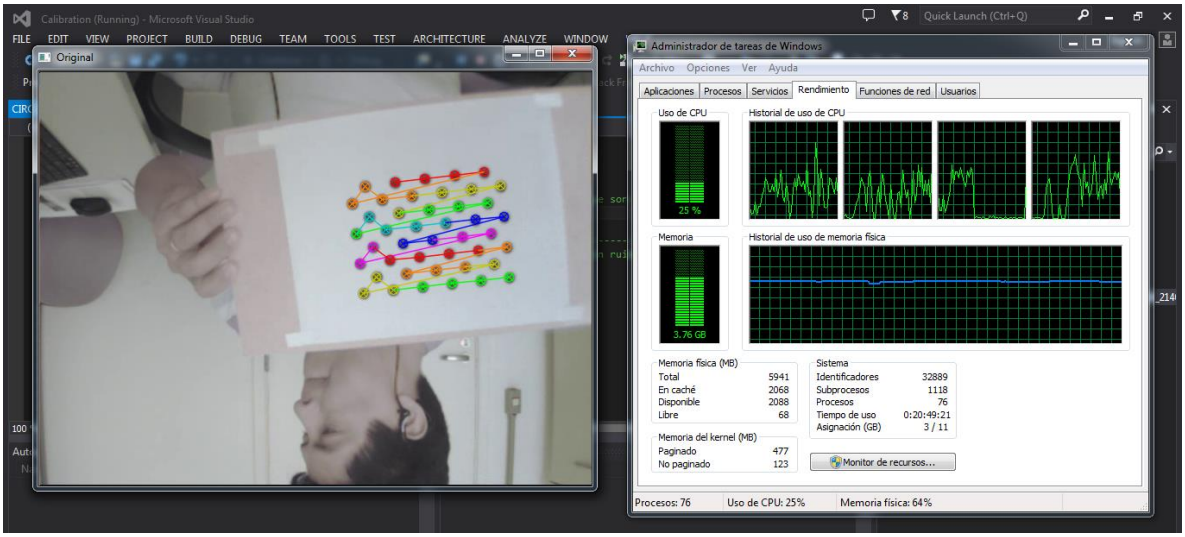
La heurística desarrollada ha sido corrida con los tres videos de prueba, y en todos ellos ha dado podido encontrar todos los centros y dibujar el zigzag con bastante fluidez y con un consumo de procesador moderado.

El algoritmo presenta una pequeña limitación, esta es que a pesar de que realiza adecuadamente la detección de los elementos del patrón en ocasiones los pierde y las líneas del zigzag desaparecen, y en algunas otras veces se deforma un poco. Sin embargo, aun con esto, el algoritmo realiza un buen trabajo ya que estos sucede menos del 40% del tiempo.

En las siguientes imágenes se muestra los resultados mencionados.


Calibration (Running) - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW
Quick Launch (Ctrl+Q)
Administrador de tareas de Windows
Archivo Opciones Ver Ayuda
Aplicaciones Procesos Servicios Rendimiento Funciones de red Usuarios
Uso de CPU
32 %
Historial de uso de CPU
Memoria
3.81 GB
Historial de uso de memoria física
Memoria física (MB)
Total 5941
En caché 2065
Disponible 2040
Libre 27
Sistema
Identificadores 33103
Subprocesos 1113
Procesos 77
Tiempo de uso 0:20:46:47
Asignación (GB) 4 / 11
Memoria del kernel (MB)
Paginado 479
No paginado 123
Monitor de recursos...
Procesos: 77 Uso de CPU: 32% Memoria física: 65%


Calibration (Running) - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW
Quick Launch (Ctrl+Q)
E:\Javier\Cursos\Maestria Ciencias de la Computacion\Semestre 2\Imagenes\Parte 2\Trabajo 1\Cali...
OpenCV Error: Incorrect size of input array (Number of points should be >= 5) in
Original
Administrador de tareas de Windows
Archivo Opciones Ver Ayuda
Aplicaciones Procesos Servicios Rendimiento Funciones de red Usuarios
Uso de CPU
28 %
Historial de uso de CPU
Memoria
3.79 GB
Historial de uso de memoria física
Memoria física (MB)
Total 5941
En caché 2053
Disponible 2055
Libre 48
Sistema
Identificadores 33139
Subprocesos 1143
Procesos 76
Tiempo de uso 0:20:48:04
Asignación (GB) 4 / 11
Memoria del kernel (MB)
Paginado 479
No paginado 123
Monitor de recursos...
Procesos: 76 Uso de CPU: 28% Memoria física: 65%


Calibration (Running) - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW
Quick Launch (Ctrl+Q)
Original
Administrador de tareas de Windows
Archivo Opciones Ver Ayuda
Aplicaciones Procesos Servicios Rendimiento Funciones de red Usuarios
Uso de CPU
25 %
Historial de uso de CPU
Memoria
3.76 GB
Historial de uso de memoria física
Memoria física (MB)
Total 5941
En caché 2068
Disponible 2088
Libre 68
Sistema
Identificadores 32889
Subprocesos 1118
Procesos 76
Tiempo de uso 0:20:49:21
Asignación (GB) 3 / 11
Memoria del kernel (MB)
Paginado 477
No paginado 123
Monitor de recursos...
Procesos: 76 Uso de CPU: 25% Memoria física: 64%

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.

II. Patrón de Anillos

1. Implementación del Algoritmo de Detección de Patrón Desarrollado

1.1. 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.

1.2. Descripción del Algoritmo

Para poder obtener los resultados esperados, la heurística desarrollada realiza las siguientes acciones.

Primeramente, se debe tener en consideración del tamaño del patrón que se va a procesar, es decir la cantidad de filas y columnas que tiene el grid de anillos (5x6 en enuestro caso), ya que esto es de mucha importancia en el funcionamiento del algoritmo.

Para poder extraer 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, a continuación 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 bordes que sean continuos, permitiendo así detectar las elipses que pertenecen a los anillos.

A continuación se deben aislar todos aquellos contornos que no formen parte de los elementos del patrón (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 anillos.

Ya habiendo obtenido todas las elipses propias de los anillos del patrón y habiéndolas separado del todo aquello que no es un elemento del patrón, aplicamos la función "*fiteEllipse*" a cada uno de los contornos obtenidos, con lo cual se ajustan a una elipse, obteniendo así los centros de todas la elipses presentes en el patrón.

Para cada elipse después de aplicar "*fiteEllipse*" se obtiene 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.

Para el patrón en uso, se puede considerar que los centros de las elipses son colieneales, lo que sirve para ordenar los centros de las elipses.

Una vez obtenidos todos los puntos del patrón (30 para nuestro caso), debemos encontrar las rectas por las cuales se dibujara el zigzag, para esto consideramos rectas que contengan 6 centros de elipse como puntos lo que permitirá generar las rectas en las horizontales del patrón.

Una vez obtenidas las 5 rectas por las que se trazaran las líneas del zigzag debemos ordenarlas para que este se mantenga constante en todo momento. Para ello, usando un concepto de algebra lineal, aplicamos el producto vectorial entre las rectas, lo que nos permitirá saber cuáles están arriba y cuales están abajo, consiguiendo así ordenarlas.

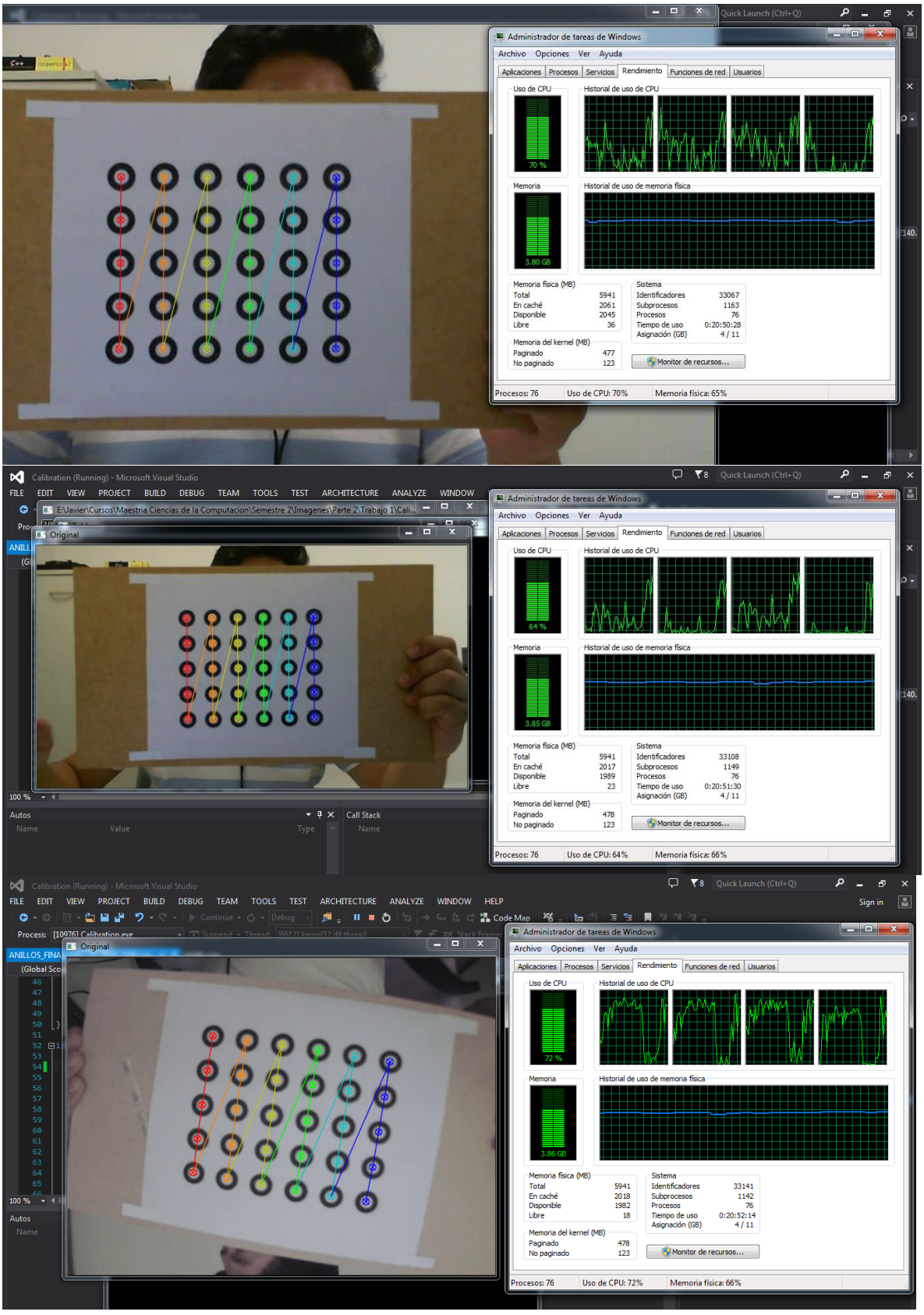
Una vez se hayan ordenado las rectas, solo resta dibujar el zigzag, que se logra usando la función "*drawChessboardCorners*".

1.3. 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.


Administrador de tareas de Windows
Archivo Opciones Ver Ayuda
Aplicaciones Procesos Servicios Rendimiento Funciones de red Usuarios
Uso de CPU
70 %
Historial de uso de CPU
Memoria
3.80 GB
Historial de uso de memoria física
Memoria física (MB)
Total 5941
En caché 2061
Disponible 2045
Libre 36
Sistema
Identificadores 33067
Subprocesos 1163
Procesos 76
Tiempo de uso 0:20:50:28
Asignación (GB) 4 / 11
Memoria del kernel (MB)
Paginado 477
No paginado 123
Monitor de recursos...
Procesos: 76 Uso de CPU: 70% Memoria física: 65%
Calibration (Running) - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW
Original
Autos
Call Stack
64 %
3.85 GB
Procesos: 76 Uso de CPU: 64% Memoria física: 66%
HELP
Sign in
72 %
3.86 GB
Procesos: 76 Uso de CPU: 72% Memoria física: 66%

III. 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 o posterior.

2. Preparar (linkear) el proyecto para que trabaje con la biblioteca OpenCV en cualquiera de las distribuciones de la versión 2 y activar la compatibilidad con OpenMP.

3. Acceder al enlace:

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

   Descargar los archivos "*in_VID5.xml*", "*Calibration.cpp*", "*Patterncircle.cpp*" y "*Pattern-Grid.cpp*".

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, el archivo .xml y los videos de los patrones con los que se harán las pruebas.

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 "*<Inut>*" 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.