DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Tracking de esferas mediante procesamiento de imágenes

Aplicando reconocimiento de bordes y la transformada de Hough en C++ y SIMD

26 de junio de 2016 — Organización del computador II

Integrante	LU	Correo electrónico
Heredia Favieri,Martin Ariel	146/11	martin.herediaf@gmail.com
Belloli, Laouen Mayal Louan	134/11	laouen.belloli@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

Las imágenes son muy bien estudiadas por el ser humano desde hace siglos, y tal vez podemos decir que la vista es uno de los sentidos mejor comprendidos por el ser humano si lo comparamos con el olfato, el tacto o el audio. Mientras que para el audio y para las imágenes se implementaron distintas codificaciones que permiten digitalizarlo y estudiarlo, para otros como el tacto o el olfato no hay ni siquiera una representación que permita su estudio dentro de la computación. Mismo así, el estudio y procesamiento del audio está mucho más limitado que el estudio y se emplean para el mismo técnicas más complejas con peores resultados que las técnicas empleadas para el estudio y procesamiento de imágenes el cual fue utilizado para múltiples propósitos (visión robótica, herramientas para estudios geográficos, estudios médicos, efectos especiales, etc). La importancia del procesamiento de imágenes en el mundo de la computación es tan alta que el día de hoy los procesadores dedicados a procesamientos gráficos (GPU) son utilizados para resolver todo tipo de algoritmos computacionalmente complejos como lo son algunas técnicas de deep-learning.

La representaciones matemáticas utilizadas para representar imágenes son múltiples, pero casi todas tienen en común la utilización de píxeles como unidad de información y la utilización de un orden y color para cada píxel. Para dar un orden a los píxeles, dado que la mayoría de las veces se consideran imágenes rectangulares, es común la utilización de matrices. La representaciones de los colores son variadas (RGB, CMYK, YUV, etc), una de las más utilizadas es RGB el cual consiste de tres canales donde cada canal guarda la intensidad de cada uno de los tres colores principales utilizados para general el resto de los colores de la paleta, rojo (R) verde (G) y azul (B) por sus nombres en inglés Red, Green and Blue. De la misma forma para representar imágenes en escala de grises se suele utilizar un sistema similar con un solo canal el cual guarda la intensidad de la luz del pixel.

Por una cuestión de la representación numérica de las computadoras, las intensidades de cada canal en RGB o del canal único en escala de grises suele representarse con enteros entre 0 y $255 = 2^8 - 1$ lo cual puede ser representado por 8 bits. Muchas veces los colores en RGB son representados en hexadecimal, por ejemplo el rojo, es representado por $ff0000$ ya que esto representa la máxima intensidad en el canal R y nada de intensidad para los canales verdes y azules $ff = 255$. De esta forma el negro, conocido por ser la ausencia de todos los colores, se representa con 000 y el blanco, conocido por ser la presencia de todos los colores con $ffffff$ o sea $(255, 255, 255)$. Mezclando entonces el orden matricial de los pixeles y su color se consigue como resultado un mapéo casi directo entre las imágenes reales y las imágenes digitales y un mapéo directo entre la representación digital de una imagen y la proyección de la misma en las pantallas.

Por otro lado, una vez que conseguimos tener una representación matemática/computacional para las imágenes, los videos pueden ser representados de manera casi trivial por una secuencia de imágenes. Esta representación se la puede pensar como la representación computacional de un flipbook el cual consiste en un libro con imágenes ordenadas de manera que al pasar sus páginas rápidamente se consigue ver los dibujos animados.

El procesamiento de imágenes, es un área de la computación dedicada a procesar las representaciones computacionales de las imágenes con algún fin particular. Ejemplos de esto son OCR (Optical Character Recognition), reconocimiento facial, filtros, reconocimiento de bordes, reconocimiento de figuras, diagnósticos médicos por imágenes, visión robótica, etc.

Una técnica de procesamiento de imágenes utilizada comúnmente para visión robótica es el reconocimiento de figuras mediante la transformada de hough. Esta técnica permite reconocer figuras parametrizables mediante una transformación del espacio de colores de los pixeles a un espacio de votaciones realizada en base a los bordes detectados previamente en la imágen. La transformación considera las posibles figuras existentes cuyos bordes son un subconjunto de los bordes detectados en la imagen. De esta manera se puede utilizar el espacio transformado para

determinar posibles figuras como líneas, cuadrados, círculos, etc.

Por otro lado, los lenguajes ensambladores suelen contar con instrucciones SIMD (Single Instruction Multiple Data) que permiten realizar una misma instrucción en paralelo (pero no en simultáneo) a varios datos. Intel lanzó un set de instrucciones llamado SSE (Streaming SIMD extensions) que operan sobre un conjunto de registros especiales (XMM) de 128 bits. SSE Implementó nuevas operaciones SIMD.

Dado que muchas de las técnicas de procesamiento de imágenes consisten en aplicar filtros, su implementación se basa en realizar las mismas operaciones para cada pixel de la imagen. Es en esos casos en donde SIMD permite paralelizar el procesamiento de las mismas reduciendo su tiempo de cómputo considerablemente pero siempre de manera lineal sin mejorar la complejidad algorítmica.

2. Objetivos

El objetivo de este trabajo es realizar un programa implementado en C++/Intel Assembly que utiliza técnicas de procesamiento de imágenes para rastrear mediante una webcam y dibujar en la pantalla, la trayectoria de una esfera (joystick) en tiempo real. Dibujar la trayectoria en tiempo real o más generalmente procesar imágenes en tiempo real, significa que: Las imágenes se van procesando a medida que son obtenidas de la webcam y se las van mostrando post procesamiento en el mismo momento. En otras palabras, se obtiene la primer imagen, se la procesa, se la muestra, se obtiene la segunda imagen, se la procesa, se la muestra, y así sucesivamente. La idea general del método consiste en procesar cada imagen recibida desde una webcam mediante tres pasos importantes:

- **Aplicación de un filtro de suavizado para eliminar ruidos:** Se trabaja sobre la hipótesis fuerte de que los saltos significativos de colores son generados por los bordes de los objetos reales proyectados en la imágen. Con esta hipótesis, al suavizar la imagen, se reduce la probabilidad de detectar bordes generados por texturas, contrastes, u otros fenómenos no deseados.
- **Detección de bordes:** La imagen es procesada por un método de detección de bordes que devuelve una lista con todos los bordes detectados. Este método es conocido como Canny.
- **Transformada de Hough:** Los bordes obtenidos tras procesar la imagen con Canny son utilizados para determinar qué puntos de la imagen son los más probables de pertenecer al centro del circulo, utilizando para esto como hipótesis, que los bordes del círculos son un subconjunto de los puntos detectados como bordes.

También se pretende comprender mejor el funcionamiento y performance de SIMD para lo cual, en este trabajo se implementan en Intel Assembly y utilizando el set de instrucciones SSE junto con los registros XMM un subconjunto de las funciones del programa que trabajan sobre operaciones paralelizables. Estas operaciones son en particular aquellas que requieren la realización de las mismas operaciones en todos o un subconjunto significativo de píxeles de la imagen a procesar.

Para poder comprender mejor SIMD, se implementan todas las funciones en C++ y se realizan análisis comparativos para medir la performance del programa al utilizar SIMD. De la misma manera, se realizan análisis de sensibilidad del método a los ruidos. En particular, se experimenta con diferentes fondos variando la cantidad de bordes, de esta forma, podemos medir cómo varía la performance del método al aumentar la cantidad de objetos en el ambiente. Por último, Para comprender mejor los métodos implementados, realizamos experimentación variando ciertos parámetros del programa que son utilizados por los 3 pasos principales mencionados previamente

para analizar el comportamiento del programa respecto de esos parámetros. Estos parámetros son explicados a continuación en la sección "Especificaciones del programa".

3. Especificaciones del programa

3.1. Input

El input del programa consiste en la conexión a una webcam mediante la utilización de OpenCV para recibir las imágenes capturadas por la webcam a ser procesadas.

3.2. Output

El output del programa es una secuencia de imágenes que contienen la trayectoria de la esfera de forma incremental. Esto significa, que la primer imagen contiene un único punto en el lugar donde la esfera es detectada en la primer imagen procesada, La segunda imagen, contiene lo mismo que la primer imagen más un nuevo punto en el nuevo lugar donde es encontrada la esfera en la segunda imagen obtenida procesada y una recta que une al primer y al segundo punto (a menos que sean el mismo), y de esta manera sucesivamente. En caso de detectar que dos puntos sucesivos están muy alejados, el programa considera primeramente que es un error (no se detectó la esfera correctamente) y no agrega nada a la imagen anterior. Si este fenómeno se repite sucesivamente más de una cierta cantidad de veces (ajustable) se considera que no era un error y que sucedió alguna de las siguientes dos situaciones: la esfera se desplazó demasiado rápido sin dar tiempo a detectar los puntos intermedios o la esfera salió del ángulo de visión de la webcam y volvió a entrar.

3.3. Parámetros

El Programa cuenta con una serie de parámetros utilizados para ajustar ciertas propiedades de los métodos implementados. Explicaciones más detalladas del efecto de cada parámetro están dadas en la sección correspondiente a la explicación de cada método implementado. Los parámetros del programa son:

1. **Umbral para la detección de bordes:** Este es utilizado para decidir qué píxeles se consideran bordes y cuáles no. A umbrales más altos, menos píxeles son considerados bordes mientras que a valores más bajos, más píxeles son considerados bordes. Si el umbral está seteado en cero, todos los píxeles se consideran bordes.

2. **Cantidad de radios:** este parámetro determina la cantidad de radios a considerar, permite ajustar la flexibilidad del programa a acercamientos y alejamientos de la esfera a la webcam. Si la cantidad de radios es uno, el programa solo busca esferas con un único radio esperado. Mientras que si la cantidad de radios es mayor a uno, el programa busca en un conjunto de radios.

3. **Radio mínimo:** este parámetro sirve para especificar cual es el radio mínimo a ser considerado. En el caso de considerar un solo radio, el radio mínimo es el único radio considerado.

4. **Radio step:** este parámetro determina la distancia entre los radios considerados. El programa arranca con el radio mínimo y luego va incrementando el radio utilizando el radio step tantas veces como la cantidad de radios indicados por el parámetro 2.

5. **Relación radios votaciones:** Este parámetro es utilizado para obtener el umbral que decide cuando el pixel con mayor votaciones a ser el centro del círculo buscado, es efectivamente un círculo o es simplemente el máximo de un conjunto de votaciones donde no se encontró ningún círculo. Este umbral representa el porcentaje en relación al radio del círculo.

6. **Distancia máxima:** Este parámetro es un umbral que determina cuándo dos puntos consecutivos están demasiado alejados para ser considerados como puntos sucesivos de la trayectoria de la esfera.

7. **Cantidad máxima de círculos lejanos:** Este parámetro es un Umbral que determina cuántos puntos lejanos se necesitan para decidir que efectivamente hubo un salto en la trayectoria de la esfera

3.4. Comandos

El programa cuenta con una lista de comandos destinados a: ajustar parámetros, pausar/reanudar el programa, mostrar el estado de los parámetros y terminar la ejecución del programa. La Tabla 1 muestra el listados de comandos disponibles en el programa:

Comando	Descripción
Comandos de control de flujos	
"Espacio"	Pausar/Retomar
"Esc"	Terminar programa/tarea.
[1] "h"	Ajustar Hough
[1] "b"	Ajustar Bordes
[1] "p"	Pintar trayectoria de la esfera
[1] "g"	C++/Assembly
Comandos de ajuste de parámetros.	
"+"	Aumentar el parámetro "Relación radios votaciones" en 0.01
"-"	Disminuir el parámetro "Relación radios votaciones" en 0.01
"w"	Aumentar el parámetro "Umbral para la detección de bordes" en 1
"s"	Disminuir el parámetro "Umbral para la detección de bordes" en 1
"6"	Aumentar el parámetro "Radio mínimo" en 1.
"4"	Disminuir el parámetro "Radio mínimo" en 1.
"d"	Aumentar el parámetro "cantidad de radios" en 1.
"a"	Disminuir el parámetro "cantidad de radios" en 1.
"8"	Aumentar el parámetro "Radio step" en 1.
"2"	Disminuir el parámetro "Radio step" en 1.
"v"	Aumentar el ancho de la línea dibujada.
"c"	Disminuir el ancho de la línea dibujada.
"l"	Aumentar la cantidad de threads a usar.
"k"	Disminuir la cantidad de threads a usar.

Tabla 1: Comandos disponibles.

Las imágenes son solicitadas por el programa cada vez que es necesario mediante la utilización de la librería OpenCV-2.4.9; de esta manera la fluidez del programa depende del tiempo que tarda en procesar cada imagen pero no genera un apilamiento de imágenes cuando baja la performance del programa. Si la performance del programa baja demasiado, la fluidez del programa se reduce junto con una trayectoria menos continua de la esfera en la pantalla.

4. Flujo general del programa

La Figura 1 Muestra el flujo general del programa desde que comienza su ejecución hasta que finaliza su ejecución. Como se puede observar, el programa cuenta con un ciclo general en el cual decide cual es la tarea a realizar en cada iteración, las tareas pueden ser acciones para ajustar parámetros, como lo son Ajustar bordes y Ajustar Hough o dibujar la trayectoria de la esfera. Cada vez que el programa entra en una tarea, se mantiene en esa tarea hasta que el comando "escape" sea presionado, en cuyo caso vuelve al ciclo general donde se mantiene ciclando a la espera de alguno de los comandos de control de flujo para poder decidir cual tarea realizar a continuación o

[1] El comando solo tiene efecto si el programa no está ejecutando ninguna tarea.

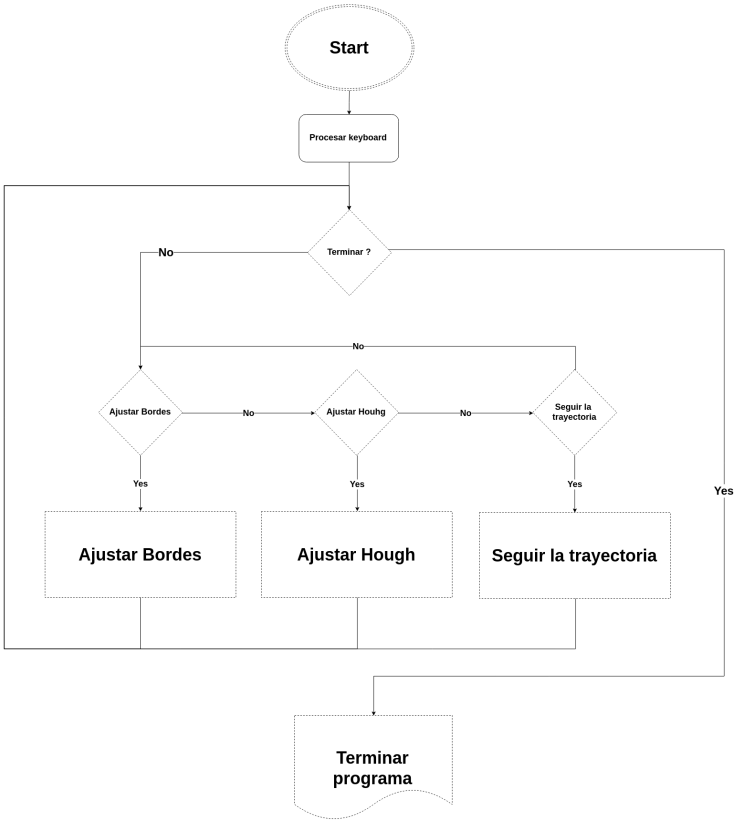


Figura 1: Flujo general del programa, el ciclo principal y sus subtareas principales.

si finalizar el programa. Las Figuras 2, 3 y 4 muestran cada uno de las tres tareas principales del programa mencionadas previamente.

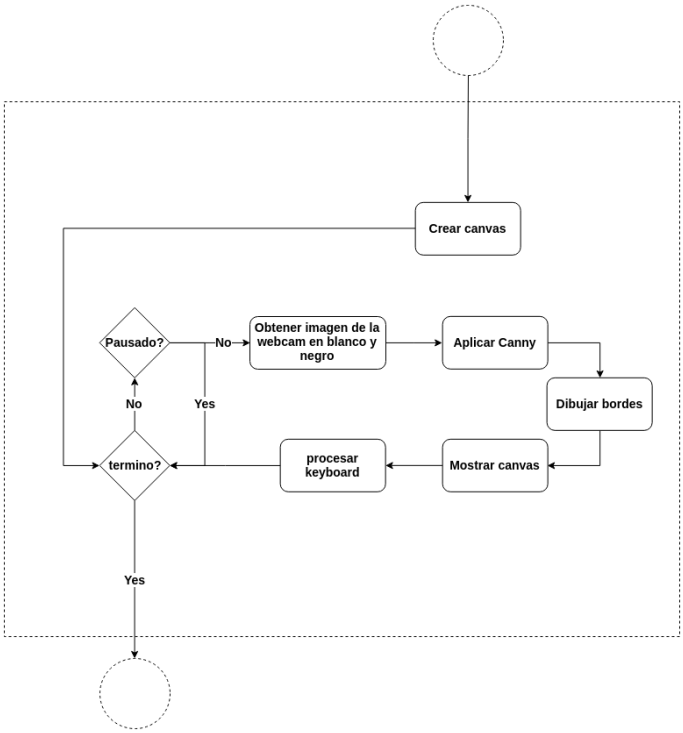


Figura 2: Flujo de la subtarea principal Ajustar Bordes.

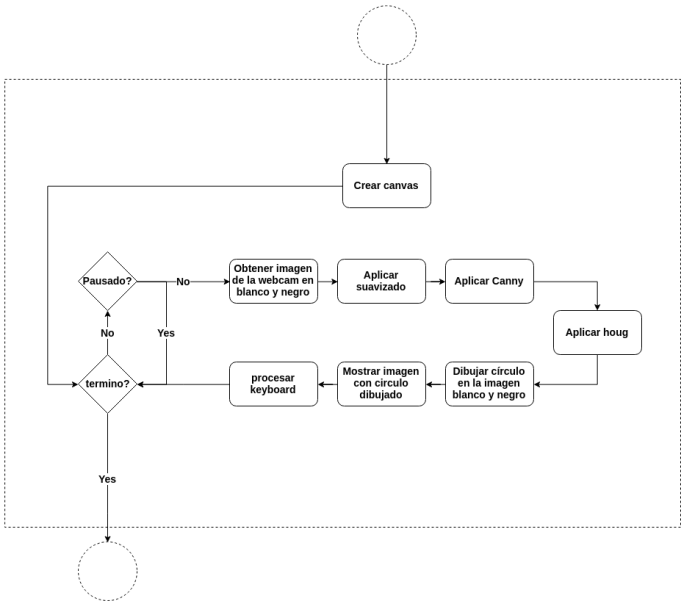


Figura 3: Flujo de la subtarea principal dibujar Ajustar Hough.

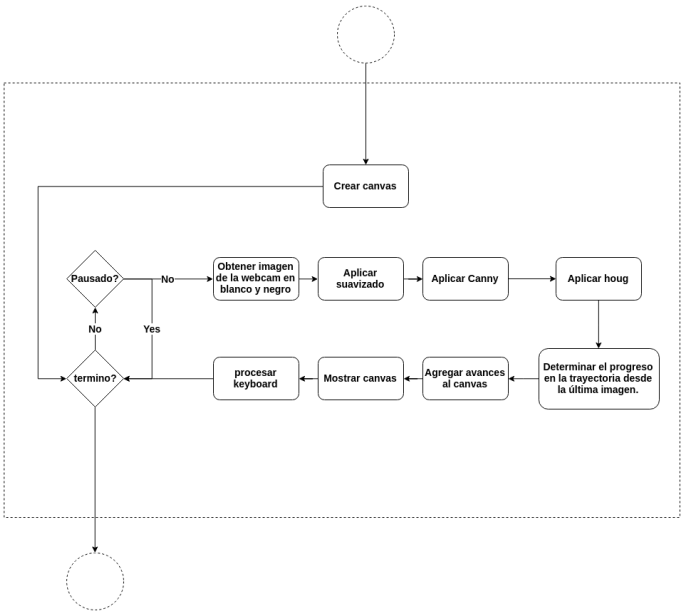


Figura 4: Flujo de la subtarea principal Dibujar Trayectoria.

1. **Ajustar Bordes:** Como se puede ver en la Figura 2, el programa procesa las imágenes de la webcam en tiempo real, aplicando el filtro de Canny a cada imagen y dibujando los bordes detectados por el filtro en el canvas (la imagen mostrada como output del programa). En cada ciclo que se procesa una imagen, se realiza una pausa de 10 milisegundos esperando una acción de teclado a procesar. Si el usuario presiona un comando de seteo de parámetros, se setea correctamente el parámetro correspondiente y se pasa a procesar la siguiente imagen con los nuevos parámetros. Si por el contrario, se presionó una tecla de modificación del flujo ("escape" o "espacio"), el flujo del programa se modifica de las siguientes maneras correspondientemente: finaliza la tarea y vuelve al ciclo general mostrado en la Figura 1, pausa la tarea a la espera de otro comando de flujo. **Nota:** Pausar la tarea significa mantener el flujo del programa corriendo pero sin procesar ninguna imagen.

2. **Ajustar Hough:** El flujo de ejecución del programa dentro de la tarea Ajustar hough es muy similar al de Ajustar Bordes, ya que procesa las imágenes de la webcam en tiempo real, utilizando un ciclo que se mantiene activo hasta que se presione "escape", momento en el cual se termina la tarea y se vuelve al ciclo general del programa. Como se ve en la Figura 3, en cada ciclo de la tarea, se obtiene una imagen de la webcam, se la procesa con los tres pasos necesarios para la detección de círculos (suavizado de la imagen, detección de bordes y transformación de hough) y se pinta el círculo encontrado sobre la misma imagen obtenida de la webcam y es mostrada como output del programa. Al mostrar el círculo encontrado sobre la imagen y no sobre un canvas vacío. El usuario puede comparar fácilmente el resultado del programa con la posición real de la esfera y ajustar mediante los comandos, los parámetros que desee.

3. **Dibujar la trayectoria de la esfera:** Esta tarea cuenta con la misma lógica para su flujo de ejecución de las otras tareas, donde las imágenes son procesadas en tiempo real mediante un ciclo de procesamiento a menos que el programa sea pausado con el comando "espacio" o la tarea sea finalizada "escape" y se retorna al flujo general del programa. En cada ciclo en el cual se procesa una imagen, se aplica el método de detección de círculos utilizado en la tarea Ajustar Hough pero en vez de dibujar el círculo encontrado sobre la imagen, se calcula el avance realizado en la trayectoria de la esfera entre la iteración anterior y la iteración actual. Una vez determinado el avance de la esfera en la trayectoria desde la última imagen,

el avance es dibujado sobre el mismo canvas donde se viene dibujando la trayectoria de la esfera hasta el momento y el canvas es mostrado como el output del programa por pantalla. De esta manera, lo que se consigue es mostrar una imagen que va dibujando la trayectoria de la esfera en el mismo momento en que la esfera produce la trayectoria en el aire. La Figura 4 muestra el flujo de esta tarea.

5. Estructuras de datos

Para la implementación del programa se implementaron estructuras de datos con la finalidad de organizar los datos manejados de forma ordenada, las estructuras de datos implementadas son las siguientes:

5.1. Borde

Esta estructura es utilizada para guardar la información obtenida sobre los bordes encontrados por el filtro Canny. una variable de tipo Borde, representa un pixel de la imagen que es considerado borde. Un borde está conformado por:

1. *posicion*: Una tupla de Integers que determinan la posición correspondiente a la posición del pixel en la imagen.
2. *gradiente*: Un double que representa el modulo del gradiente del borde.
3. *theta*: un double que representa el valor del angulo θ de la dirección del gradiente del borde.

5.2. Círculo

Esta estructura representa un círculo y es utilizado para guardar los círculos de la imágen detectados por hough. Un círculo está conformado por:

1. x: Un integer que representa la posición horizontal del pixel del centro del círculo.
2. y: Un integer que representa la posición vertical del pixel del centro del círculo.
3. *radio*: In integer que representa el radio en píxeles del círculo.
4. *valido*: Un booleano que indica si el círculo encontrado es válido o no. Este parámetro es utilizado para indicar, cuando hay o no suficientes votos sobre el círculo encontrado por la transformada de hough. La cantidad de votos indica de alguna manera cuanta probabilidad hay de que el círculo provenga realmente de un círculo en la imagen.

Además, la estructura de datos del círculo implementa el operador = que permite trabajar con asignaciones de círculos.

5.3. Recorte

Esta estructura de datos representa una zona rectangular de la imagen que es utilizada para informar a las distintas funciones del programa que sectores de la imagen deben procesar. Esto es utilizado como parte de una optimización que será explicado en la sección siguiente "Optimizaciones realizada". Esta estructura está compuesta por:

1. $j1$: La posición horizontal del borde izquierdo de la zona a procesar.
2. $j2$: La posición horizontal del borde derecho de la zona a procesar.
3. $i1$: La posición vertical del borde superior de la zona a procesar.
4. $i2$: La posición vertical del borde inferior de la zona a procesar.

5.4. Parámetros

Esta estructura sirve para manejar los distintos parámetros del programa. La estructura cuenta con un conjunto de métodos setters que sirven para setear los parámetros correctamente. Cada vez que un nuevo parámetro es actualizado, primero se realiza un chequeo de verificación que valida que los nuevos valores de los parámetros sean correctos, en caso contrario la actualización no se realiza. De esta manera se evita que el usuario pueda setear parámetros incorrectos en el programa. Los atributos de esta estructura de datos son explicados de manera más precisa en la sección "Implementación de los método", en esta sección solo los nombraremos con una explicación breve de los mismos.

1. *terminado*: Si es true, significa que el comando "escape" fue presionado y que se debe finalizar la tarea que se esté ejecutando si se está ejecutando una tarea, o el programa entero en caso contrario.
2. *pausado*: Determina si la tarea actual está en pausa o no.
3. *relacion_votaciones_radio*: Determina el porcentaje de votaciones que se necesita en relación al radio del círculo buscado para determinar que el círculo encontrado es altamente probable que provenga de un círculo real en la imagen o no.
4. *umbral_bordes*: Es el umbral utilizado en Canny para determinar cuando un punto es considerado borde o no.
5. *min_radio*: Es el radio mínimo a considerar.
6. *cantidad_radios*: Es la cantidad de radios a considerar a partir del radio mínimo.
7. *radio_step*: Es la distancia que hay entre dos radios consecutivos de los radios considerados.
8. *ancho_linea*: Determina el ancho de la linea que dibuja la trayectoria de la esfera.
9. *distancia_maxima*: Es la distancia máxima en la cual se considera que dos puntos sucesivos pertenecen a una misma trayectoria o no.
10. *cantidad_lejanos_maximos*: Determina la cantidad máxima de puntos sucesivos que pueden ser determinados lejanos por el parámetro anterior antes de decidir que los puntos lejanos no son errores y que son en realidad un salto abrupto de la esfera.

6. Optimixaciones realizadas

El programa no solo cuenta con las optimizaciones realizadas al implementar ciertas funciones con SIMD, sino que también cuenta con 2 enfoques principales que sirven para optimizar aquellos puntos del programa que pueden ser cuellos de botella para la aplicación. Estos dos enfoques serán explicados a continuación.

6.1. Paralelización con concurrencia

El programa contiene varios puntos en los cuales se realizan ciclos anidados. Los ciclos anidados, aparecen cuando se recorre una imagen, utilizando un iterador para cada una de las dimensiones de la misma (filas y columnas). Cuando esto sucede, se puede paralelizar mediante la utilización de threads de manera de partir la imagen en bloques, los cuales pueden ser procesados de manera concurrente. Estos ciclos anidados ocurren en el primer filtro de suavizado, en el cual se le aplica una convolución a cada pixel de la imagen y en el filtro de Canny a cargo de detectar los bordes de la imagen.

Además de los ciclos anidados recién mencionados, existe un ciclo simple, utilizado por la transformada de Hough para determinar la transformación correspondiente para cada radio considerado. En este proceso, es difícil partir el problema en partes como en el caso de procesar una imagen, ya que las votaciones se realizan sobre una matriz en la cual se desconoce en un principio que puntos va a necesitar cada threads, teniendo entonces que compartir esa memoria resultando en posibles problemas. De todas maneras, lo que sí se puede paralelizar es la realización de la transformación completa para cada radio, de esta forma, para cada radio considerado, se realiza la transformación correspondiente, en un thread diferente sin compartir memoria.

Implementamos esta optimización para poder comparar la eficiencia de SIMD contra la eficiencia de programación concurrente, y también para poder analizar su rendimiento en conjunto.
La Figura 5 muestra como se dividen y procesan las imágenes de forma concurrente mediante la utilización de threads.

Para la implementación de threads se utilizo la libreria "thread" de la STD. Para paralelizar utilizando esta librería, se necesita un vector de threads a los cuales se les asigna una función a ejecutar y los parámetros de las funciones. Luego, una vez inicializados los threads en el vector, se los recorre para realizar un join, el cual espera a que cada thread termine su ejecución antes de avanzar con la ejecución del programa principal.

Las imágenes a procesar en threads son divididas en bloques, primero se determina una partición de la misma en líneas horizontales y verticales, obteniendo una imagen dividida en bloques, cada bloque es procesado por un thread distintos.

Una vez obtenidos los índices de la partición, se inicializa cada thread pasándole una zona a procesar como muestra la Figura 5 que especifica que parte de la imagen tiene que procesar. De esta manera, la imagen no es cortada y es pasada como referencia a cada thread y los threads utilizan la zona a procesar para saber cuales son los límites que tienen que considerar. De esta manera, se evita el costo de copiado de cada bloque de la imagen.

Una zona a procesar es una variable del tipo recorte, explicada previamente en la sección "estructuras de datos" que contiene un índice i1 para la primer fila a considerar, i2 para la última fila a considerar, j1 para la primer columna a considerar y j2 para la última columna a considerar.

Las funciones que utilizan recortes para determinar los índices del ciclo de procesamiento, no realizan ningún tipo de chequeo previo para determinar si el recorte pasado por parámetro pertenece a una zona válida, dejándole la responsabilidad total de pasar una zona a procesar válida al usuario de la función.

Dado que no conocemos las dimensiones de las imágenes a procesar desde un comienzo, y que la cantidad de threads son parte de los parámetros del programa, es posible que queden sectores de la imagen no procesadas. Esto sucede cuando el alto y/o ancho de la imagen no es divisible por la cantidad de bloques especificados, y al ser una división entera el resultado es que el resto de dicha división equivale a las columnas (para el caso del ancho) y filas (para el caso del alto) de la imagen que no serán procesadas. Sin embargo, podemos controlar cuan grande es este resto y de esa forma, dado que buscamos una esfera consideramos todo su perímetro, podemos determinar que la cantidad de pixeles de la imagen descartada al paralelizar en threads, es lo suficientemente chica para no afectar el método aún si algunas partes de la esfera se encuentran justo sobre una de estas secciones no procesadas de la imagen.

6.2. Procesamiento por zonas

Este enfoque utiliza la fuerte asunción de que el movimiento de la esfera no se genera a más de una determinada velocidad. O lo que es lo mismo, que dado dos imágenes consecutivas, dado que

el intervalo de tiempo entre cada imagen es muy pequeño, la distancia entre las posiciones de la esfera en ambas imágenes no puede ser demasiado grande. Cuan grande puede ser esa distancia es un resultado al que se llegó mediante experimentación, y el resultado para un movimiento normal es inferior a 2 veces el radio de la esfera.

Dado que sabemos que la posición de la esfera en la imagen que está siendo procesada en un ciclo, determina la zona de la imagen con mayor probabilidad de que este la esfera en la siguiente imagen, se utiliza esta zona con mayor probabilidad como zona a procesar en la siguiente iteración en vez de procesar toda la imagen.

Dado que por más chica que sea, siempre existe la probabilidad de que la esfera no se encuentre en la zona definida como altamente probable. Si efectivamente no se encuentra el círculo en esa zona, en la siguiente iteración, se vuelve a buscar la esfera en toda la imagen.

Haciendo un análisis sobre la performance ganada contra la cantidad de imágenes no procesadas exitosamente con este enfoque, podemos considerar los siguientes puntos:

Si en general, el método es exitoso y se encuentra la esfera, entonces en todos esos casos, el procesamiento es más rápido que el procesamiento considerando toda la imagen, entonces, al ser más rápido, se procesan más imágenes por segundo, disminuyendo el intervalo de tiempo entre dos imágenes consecutivas, y por ende, dado que la distancia de la esfera entre dos imágenes consecutivas es proporcional al intervalo de tiempo entre dichas imágenes, también se reduce esta distancia y aumenta la probabilidad de que la esfera se encuentre en la zona determinada como la zona con mayor probabilidad. Por otro lado, sin importar la tasa de aciertos del método, el peor caso es que la esfera nunca se encuentre en la zona determinada, en cuyo caso, la velocidad disminuye, pero no al doble de lento, ya que en este caso, la esfera es encontrada una de cada dos veces, ya que si falla, en el ciclo siguiente busca en toda la imagen y la encuentra. Como

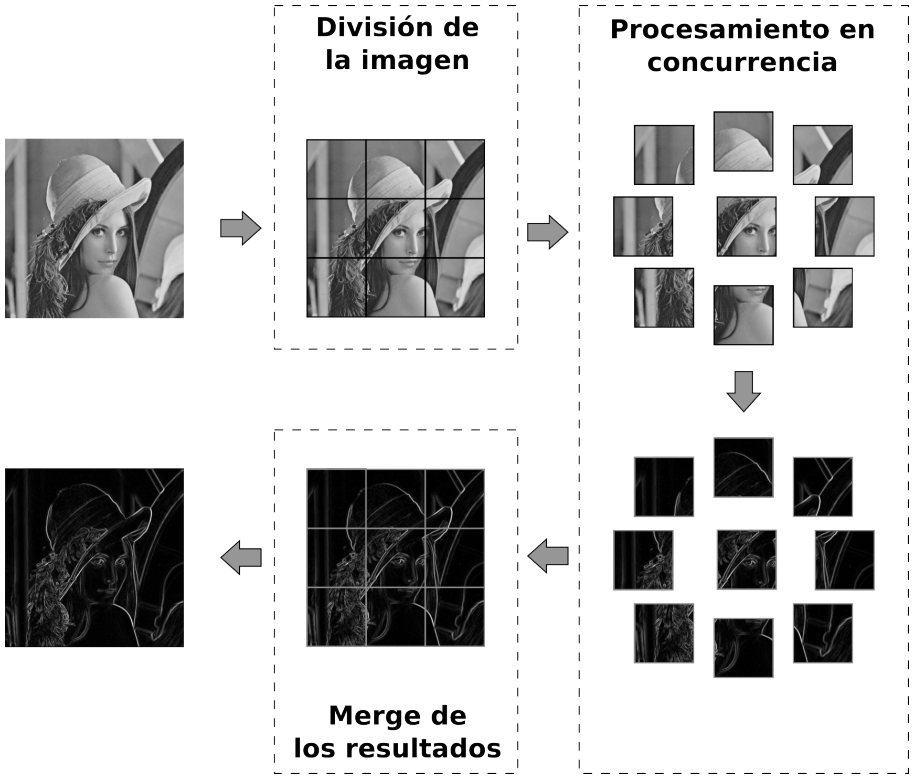


Figura 5: Diagrama del flujo de paralelización.

cuando falla no proceso toda la imagen, los ciclos descartados que son la mitad, consume considerablemente menos tiempo de procesamiento, disminuyendo la velocidad de manera no considerable.

Además, como esta optimización determina la zona a procesar en el siguiente ciclo de forma proporcional al radio de la esfera, la performance del programa se ve afectada por la misma, tardando más en procesar cada imagen cuando el radio es grande que cuando es pequeño.

Para implementar esta optimización por zonas, se utiliza la misma estructura de datos de recortes utilizada para los threads, en cada ciclo que se realiza hough, al finalizar, en caso de haber encontrado un círculo en la imagen (i.e. si el círculo encontrado está marcado como válido) entonces se setea el recorte con:

- $i1 = -2 * radio$
- $i2 = 2 * radio$
- $j1 = -2 * radio$
- $j2 = 2 * radio$

Esta nueva configuración del recorte es utilizada en la siguiente iteración para saber que zona de la imagen debe considerarse. En caso de que no se encuentre un círculo válido, el recorte se setea con las dimensiones de la imagen para considerar la imagen entera.

Al igual que para los threads, la zona a procesar no es copiada de ninguna forma física, en cambio, se utilizan los límites marcados por la zona a procesar como los límites de la imagen. Además, tampoco se chequea que los límites seteados en que el recorte sean válidos respecto de las dimensiones de la imagen a procesar. Se utiliza fuertemente el hecho de que las dimensiones de las imágenes devueltas por la webcam no varían, siendo así todas las imágenes procesadas, de la misma dimensión.

7. OpenCV-2.4.9

OpenCV-2.4.9 es una librería implementada para C++, Python y Matlab que brinda un set de métodos útiles para la carga de imágenes, así como para el procesamiento de las mismas y su posterior visualización.

Las imágenes pueden ser cargadas desde archivos o desde dispositivos como lo son las webcam y otros dispositivos conectados a la computadora mediante USB. Esta librería ofrece estructuras de datos para la representación de las imágenes que permiten un rápido acceso a los píxeles de la imagen. Además tiene implementados los métodos que desarrollamos en este trabajo (suavizado, Canny y transformada de Hough) permitiéndonos comparar no solo entre C++ y Assembler, sino también, entre nuestra implementación y la implementación de OpenCV-2.4.9.

La estructura de datos principal de OpenCV que utilizamos en este trabajo para guardar las imágenes es las estructura cv::Mat. La misma consiste principalmente de un arreglo de píxeles y una cantidad de columnas y filas. De esta forma, si se desea acceder a la posición (i, j) de la imagen, hay que acceder a la posición $i * cantidad\_de\_columnas * tamaño\_de\_pixel + j * tamateno\_pixel$, donde el tamaño del píxel varía dependiendo la cantidad de canales de los mismos. Para una imagen RGB, que tiene 3 canales R, G y B, el tamaño del píxel es $3 * 8$ bits, ya que cada canal está representado por un uchar de 8 bits. De la misma forma, para una imagen en escala de grises, el tamaño del píxel es de 8 bit, ya que tiene un solo canal.

La estructura de datos cv::Mat ofrece el método step1 que devuelve el ancho de las filas, evitando hacer la cuenta (cantidad de columnas)*(tamaño de píxel). Además, en algunos casos, las

imágenes contienen un padding que no sería considerado por la cuenta anterior y que si es considerado por el método step1 de cv::Mat.

Además. para comunicarnos con la webcam, se utiliza el tipo de datos VideoCapture que detecta en su inicialización la webcam de la computadora y posee el operador >> que permite obtener una imagen desde la webcam.

La API de openCV-2.4.9 que utilizamos en este trabajo consta de los siguientes métodos:

1. Bool $VideoCapture :: isOpened()$: Devuelve true si la variable de tipo $VideoCapture$ efectivamente pudo conectarse a una webcam en su inicialización.

2. void $Operator >> (VideoCapture\ c,\ Mat\&dst)$: c obtiene una imagen en el momento de la webcam, e inicializa dst copiando, la imagen recién obtenida de la webcam en dst. la estructura de VideoCapture. Mantiene una referencia a la imagen obtenida en dst, por lo que no hay que encargarse de liberar la memoria de dst, ya que lo hace automáticamente c en su destructor.

3. void $cv :: namedWindows(String\ n)$: construye una ventana gráfica que cuenta con múltiples funcionalidades como zoom, guardar, copiar, etc. En esta ventana se pueden mostrar imágenes de tipo Mat mediante el método $imshow$ explicado a continuación.

4. void $cv : imshow(String\ n,\ const\ Mat\&\ img)$: Muestra la imagen guardada en img en la ventana con nombre n. Nota: este método trabaja en un subproceso y requiere la utilización de $waitKey$, quien duerme el proceso principal a la espera de una interrupción del teclado para ceder el procesador y poder mostrar la imagen rápido. De otra manera, se genera un delay, y en ocasiones, la imagen directamente no es mostrada.

5. void $cv :: destroyWindow(String\ n)$: Este es el destructor de las ventanas gráficas, que no es automático. Necesita ser llamado para liberar la memoria.

6. int $cv :: waitKey(Int\ ms)$: Interrumpe el proceso, durmiendo el proceso actual por un máximo de ms milisegundos a la espera de una interrupción del teclado, si una interrupción aparece, la función devuelve el código ASCII del comando presionado. Si no aparece ninguna interrupción, la función devuelve -1.

7. void $cv :: cvtColor(Mat\&\ src,\ Ma\&\ dst,\ int\ cde)$: Genera una transformación en la codificación de los píxeles de la imagen imagen guardada en src y lo guarda en dst. Por ejemplo, se pueden pasar imágenes de RGB a imágenes en escala de grises con un solo canal. El último parámetro cde determina la transformación a realizar. Para pasar de RGB a escala de grises, se utiliza la constante $COLOR\_BGR2GRAY$.

8. void $cv :: flip(cv :: Mat\&\ src,\ cv :: Mat\&\ dst,\ Int\ cde)$: Genera una rotación de la imagen src y guarda la nueva imagen obtenida mediante la rotación en dst. El último parámetro cde, indica en qué sentido y cuantas veces rotar la imagen. Por ejemplo, pasándole como cde 1, el método genera un espejamiento horizontal de la imagen.

9. void $cv :: circle(cv :: Mat\&img, cv :: Pointp, Intr, cv :: Scalarclr, Int\ t, Int\ cde)$: Dibuja un círculo en img con centro en p y radio r, el círculo es esta pintado de color clr, tiene un ancho de borde de tamaño t. El estilo del borde puede ser determina en el último parámetro cde quien indica si es una línea continua, partida, etc.

10. void $cv :: line(cv :: Mat\&\ img,\ cv :: Point\ p1,\ cv :: Point\ p2,\ cv :: Scalar\ clr,\ Int\ t)$: Dibuja una línea en la imagen img que une los puntos $p1$ y $p2$, tiene color clr y el ancho de la línea está determinado por t.

11. void $cv :: putText(cv :: Mat\&img,\ String\ text,\ cv :: Point\ orig,\ Int\ f,\ Double\ s,\ cv :: Scalar\ clr)$: Escribe el texto especificado por el String *text* en la imagen *img* utilizando la fuente especificada por f (OpenCV-2.4.9 tiene su propio set de fuentes). El texto tiene como punto de partida *orig*, esta escalado por s (el tamaño de la fuente es $s * tamaño\_estandar$) y tiene el color *clr*. Nota: Esta función no reconoce los caracteres especiales como el salto de línea o tab. Para conseguir estos formatos es necesario crear un función propia que reciba el texto y maneje estos caracteres especiales, separando el texto en partes y realizando múltiples llamadas a putText.

12. void $Mat :: release()$: Este es el destructor de la estructura Mat que tiene que ser explícitamente llamado para liberar la memoria utilizada por esta estructura, a menos que la misma esté también referenciada desde otro sitio, como es el caso cuando se utiliza el operador >> de $VideoCapture$ en donde la estructura $VideoCapture$ se responsabiliza por destruirlo en su destructor.

8. Métodos utilizados

8.1. Suavizado

Para el suavizado de las imágenes, se implementó un método de suavizado lineal por matriz de convolución gaussiana. Este método permite suavizar la imagen, eliminando los detalles que podrían generar ruido al ser considerados bordes, ejemplos de esto, son todos los cambios de colores generados por texturas de distintos materiales o los reflejos de la luz sobre los objetos. Este método de suavizado es robusto y preserva los saltos significativos de colores de manera de no perder los bordes de los objetos. El método se llama así, pues se aplica una función lineal con coeficientes gaussianos sobre los píxeles vecinos al píxel que se está procesando, los coeficientes gaussianos son determinados considerando la distancia de cada píxel vecino al pixel que se está procesando.

Se llama kernel a la matriz de coeficientes que considera el entorno al píxel procesado. Esta matriz puede ser de diferentes dimensiones, pero generalmente es una matriz cuadrada de nxn. Para el fin de este trabajo, se considero una matriz de 5x5 por ser el estándar ya que no existe ningún requerimiento especial a considerar. La matriz de convolución utilizada es la mostrada en la Tabla 2

2	4	5	4	2
4	9	12	9	4
5	12	15	12	5
4	9	12	9	4
2	4	5	4	2

Tabla 2: Matriz de convolución para el filtro de suavizado de imágenes.

La función lineal resultante de la matriz de convolución se muestra en la Ecuación 1:

$$G(x,y) = \frac{\sum_{i=y-2}^{y+2} \sum_{j=x-2}^{x+2} M(i,j) * I(i,j)}{159} \tag{1}$$

Donde M es el kernel de coeficientes gaussianos y I es la imagen. Para aplicar el filtro, se realiza una convolución de la imagen, reemplazando el valor de cada píxel de la imagen original por el valor de $G(x,y)$, con (x,y) la posición del píxel. Las Figuras 6 y 7 muestran un ejemplo de una imagen procesada con el filtro de suavizado.

Figura 6: Imagen original.

Figura 7: Imagen suavizada.

8.2. Detección de bordes (Canny)

Para la detección de bordes utilizamos un filtro llamado Canny. El mismo utiliza el gradiente de cada píxel para determinar si el píxel es un máximo respecto de su vecindad (Ver Tabla 3) o no y marcarlo como borde, ya que los bordes están caracterizados por saltos significativos en la intensidad de los píxeles. La principal diferencia entre Canny y otros filtros de detección de bordes, es el hecho de que Canny utiliza más información, de forma de comparar los pixeles solo con aquellos vecinos que están en la dirección del gradiente y utilizar un double-threshold para marcar bordes fuertes y débiles con el fin de luego determinar cuáles de los bordes débiles, son bordes y cuales son ruido. De esta forma, consigue obtener bordes más finos y mejor definidos que otros filtros. A continuación explicaremos cada uno de los pasos de canny:

1	2	3
4	X	5
6	7	8

Tabla 3: Vecindad de un píxel cualquiera X.

1. **Finding gradients:** En esta etapa, se obtiene el gradiente de cada píxel de la imagen mediante el operador de sobel: Este operador, aplica dos matrices de convolución para determinar de forma aproximada el gradiente del pixel en las direcciones x e y, utilizando para esto la diferencia de intensidad entre el píxel procesado y su vecindad (Tabla 3, vecindad de un píxel X). Las matrices de convolución utilizadas se muestran en las Tablas 4 y 5. Una vez obtenidos los gradientes en las direcciones x e y, se utiliza el teorema de pitágoras para obtener el ángulo del mismo; sea entonces Gx y Gy los gradientes obtenidos en las direcciones x e y respectivamente, la Ecuación 2 muestra cómo se obtiene el ángulo θ.

$$\theta = \arctan \frac{|Gy|}{|Gx|} \tag{2}$$

   La idea de obtener el ángulo del gradiente, es obtener la dirección del mismo ya que esta determina también la dirección del borde y permite entonces saber cuáles vecinos son aquellos que deberían ser significativamente distintos si el pixel del gradiente es efectivamente un borde. El ángulo es entonces discretizado midiendo su distancia respecto de los siguientes ángulos: 0°, 45°, 90°, 135°, 180°, 225°, 270° y 315°:

   - Si está más cerca de 0° o 180°, entonces la dirección es horizontal (—) y se compara al píxel x con los vecinos 4 y 5 (ver Tabla 3).
   - Si está más cerca del 45° o 225°, la dirección del gradiente es diagonal (\) y se realiza la comparación utilizando los píxeles 3 y 4 de su vecindad.

- Si está más cerca a los ángulos 90° o 270° el gradiente tiene dirección vertical (|) y se compara utilizando los píxeles 2 y 7 de su vecindad.
- Si está más cerca de los ángulos 135° o 315°, el gradiente tiene dirección diagonal (/) y se utilizan en la comparación, los píxeles 1 y 8 de su vecindad.

La Figura 8 muestra los resultados al obtener los módulos de los gradientes de los píxeles de la imagen suavizada anteriormente.

-1	0	1
-2	0	2
-1	0	1

Tabla 4: convolución soble en dirección X.

-1	-2	-1
0	0	0
1	2	1

Tabla 5: matriz sobel en dirección Y.

2. **Non-maximum suppression — double-thresholding:** En esta etapa se utiliza la información obtenida en la etapa anterior para determinar qué píxeles son bordes fuertes, cuales bordes débiles y cuales no son bordes. Para esto se recorren todos los píxeles y para cada píxel se compara el módulo de su gradiente con el de sus vecinos correspondientes según el criterio del punto anterior. Si el pixel es un máximo local (su gradiente es mayor al del de sus vecinos correspondientes) entonces se lo compara con dos umbrales. Si el módulo del gradiente es mayor al umbral más alto, se lo marca como borde fuerte. Si es menor al umbral más alto pero mayor al umbral más bajo, se lo marca como borde débil y si es menor a ambos umbrales se lo marca como no borde. Si el píxel no es un máximo local, también se lo marca como no borde. La idea detrás de esto, es evitar marcar como bordes a píxeles que son ruido de la imagen sin perder aquellos puntos de bordes que se puedan confundir con el ruido, asumiendo que la diferencia entre el ruido y el borde no muy definido es que el borde no muy definido tiene que estar en algún lugar unido a un borde bien definido. La Figura 9 muestra el resultado tras aplicar non-maximum suppression double-thresholding, la imágen tiene pixeles en negro (no son bordes) en gris (bordes débiles) y en blanco (bordes fuertes).
3. **Edge-tracking by histeresis:** Como paso final, se realiza un filtrado de los bordes débiles para determinar cuáles de ellos son bordes aunque su intensidad no sea tan alta y cuales son ruidos. Para esto, se buscan aquellos puntos marcados como bordes débiles que tengan un camino de píxeles contiguos que también sean bordes débiles y que estén en contacto con al menos 1 pixel de borde fuerte. De esta forma, cualquier cluster de píxeles de bordes débiles aislado de todo borde fuerte, es considerado ruido. Mientras que aquellos bordes débiles en contacto (directo o indirecto) con bordes fuertes se lo considera la continuación del borde con el cual están en contacto.

La Figura 10 muestra el resultado final luego de aplicar Canny la imagen.

Figura 8: Módulo de gradiente de cada píxel de la imagen.

Figura 9: Non maximum suppression.

Figura 10: Resultado final canny.

8.3. Detección de círculos (Transformada de Hough)

Para esto, utilizamos la transformada de hough que permite transformar un espacio de bordes de una imágen en un espacio de acumulación en el cual se acumulan votos. El espacio de acumulación es una matriz del mismo tamaño que la imagen original donde cada posición representa el píxel de la imagen y cada voto asignado en el espacio de acumulacion, representa un voto a favor de que la posición del voto en el espacio de acumulacion, sea el valor de un parámetro en la imagen para alguna figura parametrizable que se desea encontrar.

La transformada de hough, es utilizada para encontrar figuras tales como líneas, círculos, elipses, rectángulos, etc en imágenes mediante una previa detección de bordes. El algoritmo asume que cada borde detectado podría ser el borde de la figura buscada y despejando parámetros desconocidos de la parametrización de la figura obtiene el pixel de la imagen para el cual, si ese píxel es el verdadero valor de los parámetros desconocidos, entonces el borde detectado sería el borde de esa figura en la imágen. Este concepto quedará más claro a continuación donde explicamos la transformada de hough para el caso particular de la detección de círculos que es el que nos interesa en este trabajo.

En el caso particular de los círculos, sabemos que los mismos pueden ser parametrizados mediante la Ecuación 3.

$$Circulo(x, y, r) = \{(x + r * \cos\theta, y + r * \operatorname{sen}\theta) \mid [0, 2)\} \tag{3}$$

En esta ecuación, x, y y r son parámetros mientras que no, ya que el mismo tiene que recorrer todos sus valores posibles para terminar de formar el círculo. La posición (x, y) es el centro del círculo y r es el radio del mismo. Utilizando entonces la transformada de hough, despejamos x e y como lo muestran las Ecuaciones 3 y 4 y asumiendo un r ya dado; para cada borde detectado en la imagen, si el mismo es el borde de un círculo de radio r, entonces, el centro del círculo tiene que ser el obtenido al despejar x e y, por lo que el borde le asigna un voto al punto (x, y) encontrado.

El problema que surge, es que para cada valor de $[0, 2)$, se obtiene un valor distinto de (x, y), es por esto que cada punto de borde podría votar a muchos puntos como los posibles centros del círculo buscado. Sin embargo, para evitar que cada punto de borde detectado genere tantos bordes, se utiliza la dirección del gradiente obtenida junto con el borde para obtener los potenciales que tienen sentido , ya que si un punto es el borde de un círculo, el centro del círculo tiene que estar en dirección del gradiente del borde. De esta forma, se reduce la cantidad de votos para cada punto de borde a 2, uno para cada sentido de la dirección del gradiente. La Figura 11 muestra un gráfico donde se puede ver un círculo y muestra para un punto específico del borde, la dirección del gradiente y los puntos que vota el mismo en la transformada de hough.

Una vez recorridos todos los bordes de la imagen y realizadas las votaciones, se obtiene una imagen en la cual, cada píxel contiene la sumatoria de todos los bordes que lo votaron como centro, es natural entonces que todos los bordes correspondientes a un círculo de radio r, van a coincidir

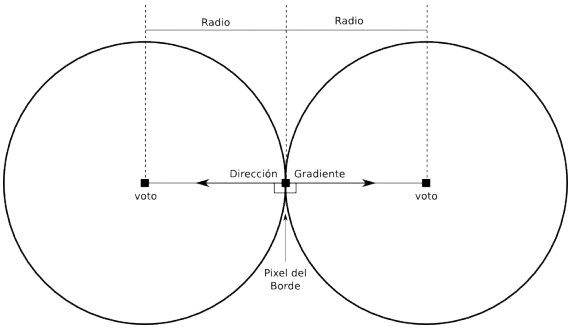
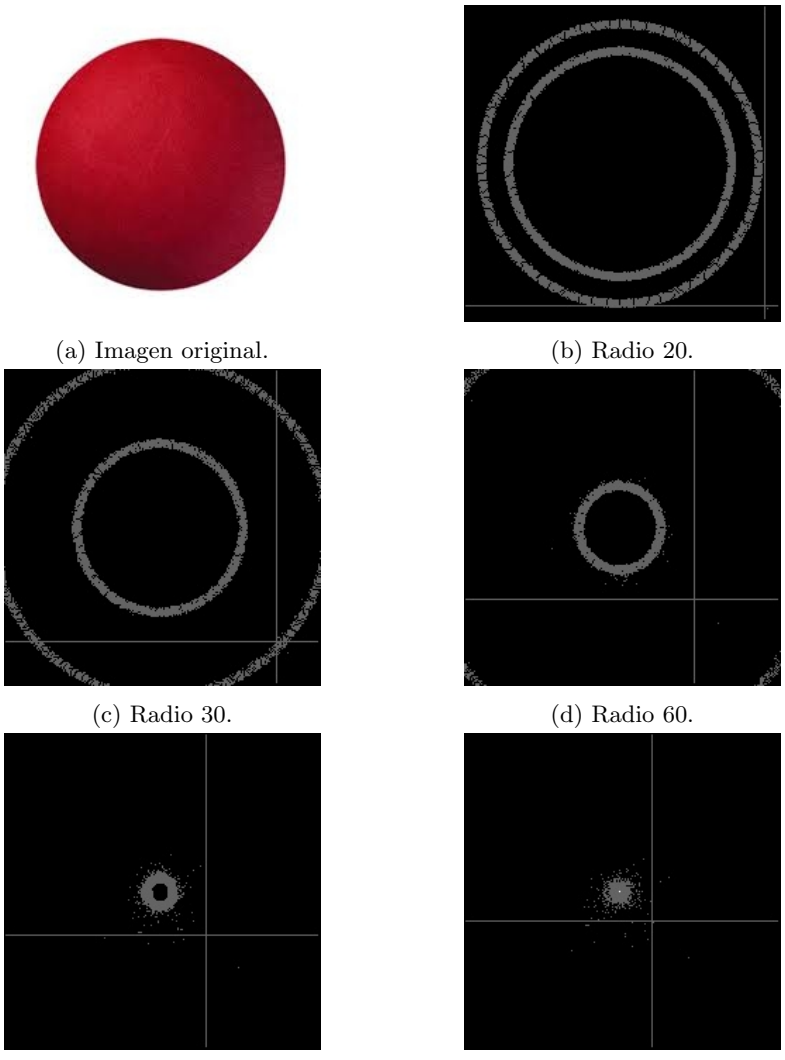


Figura 11: Píxel del borde y los dos votos generados mediante la dirección del gradiente y el radio.

en uno de sus votos, que es el centro del círculo real en la imagen. El último paso, consiste en la utilización de un umbral, para determinar qué píxeles de la imagen de votaciones, tiene suficientes votos para ser considerado un centro de un círculo de radio r. La Figura 12 muestra una imagen con un círculo y sus respectivos espacios de acumulación de votos para distintos radios. Se puede ver para cada radio, los dos círculos formados por las votaciones, a medida que el radio crece, el circulo externo se hace más grande y el circulo interno más pequeño hasta llegar al radio real del circulo de la imagen, en donde el circulo interno se convierte en un punto, para ese radio, las votaciones coinciden en el centro del círculo obteniendo el centro como el punto más votado (el único punto blanco en la imagen). **Nota:** las lineas que se generan, se deben a que el borde inferior y derecho de la imagen son detectados por canny, por lo que ellos también votan en dirección a su gradiente formando un recta, la distancia de las rectas a los bordes, esta determinada nuevamente por el radio.



(a) Imagen original. (b) Radio 20.

(c) Radio 30. (d) Radio 60.

(e) Radio 80. (f) Radio 90.

Figura 12: Espacio de acumulación de la transformada de Hough para distintos radios. Los puntos en gris, son aquellos puntos que recibieron menos de 150 votos, los puntos blancos, recibieron más de 150 votos.

9. Implementación de los métodos

9.1. Implementación en C++

9.1.1. Suavizado

Para implementar el suavizado en c++ se colocó en memoria la matriz de la Tabla 2 utilizando para esto un $int**$ que es pasado como parámetro. Luego para realizar la convolución, se utilizan cuatro ciclos, dos ciclos son utilizados para recorrer la imagen por filas y columnas y dos ciclos son utilizados para calcular en cada píxel de la imagen la convolución de suavizado.

Dado que la convolución utiliza un vecindario al rededor del píxel procesado, puede pasar que al realizar la convolución algunos de los píxeles a los que se necesita acceder estén fuera de la imagen generando un error de $segmentationfault$. Para evitar esto, tenemos en cuenta que el vecindario es de tamaño $5x5$ al rededor del pixel procesado, sabemos entonces, que los píxeles a los que se

van a acceder están como máximo a distancia dos del mismo, es por esto, que lo que necesitamos, es asegurarnos que el pixel que se esta procesando, este a una distancia mayor a dos de cualquier borde de la imagen para que su vecindario esté completamente contenido en la imagen. Es por esto, que en vez de procesar la convolución en la imagen entera, dejamos un borde de dos píxeles sin procesar al rededor de la imagen.

Pensando en las optimizaciones realizadas, el algoritmo toma como parámetro un recorte que indica que zona de la imagen debe suavizar, de esta manera si se procesa en concurrencia, a cada thread se le asigna una zona diferente y además solo suavizamos la zona de la imagen que queremos procesar y no la imagen entera (ver la sección *Optimizaciones realizadas*).

A continuación, presentamos el algoritmo de suavizado, el mismo fue dividido en dos partes que son presentadas a continuación. El Algoritmo 2 muestra la realización de la convolución para un pixel (i, j), el Algoritmo 2 muestra el recorrido de la imagen y aplicación de la convolución a cada píxel dejando un borde de dos píxeles sin procesar al rededor de la imagen.

Algoritmo 1 Barrido de la imagen en el rango correcto aplicando la convolución de suavizado

Entrada: Mat img, recorte $zonaAProcesar$.

$primera\_fila \leftarrow \text{mín}\{zonaAprocesar.i1, 3\}$
$ultima\_fila \leftarrow \text{máx}\{zonaAprocesar.i1, img.alto - 3\}$
$primera\_columna \leftarrow \text{mín}\{zonaAprocesar.j2, 3\}$
$ultima\_columna \leftarrow \text{mín}\{zonaAprocesar.j2, img.ancho - 3\}$

para $i \in [primera\_fila \ \ldots \ ultima\_fila]$ **hacer**
  para $j \in [primera\_columna \ \ldots \ ultima\_columna]$ **hacer**
    $img.data[i][j] \leftarrow convolución\_suavizado(img, i, j)$
  fin para
fin para

Algoritmo 2 ($convolución\_suavizado$) Convolución de suavizado para el píxel (x, y) de una imagen

Entrada: Mat img, int x, int y, int** $matriz\_convolución$..
Salida: El resultado de la convolución para el píxel (x, y) de la imagen img

$acumulador \leftarrow 0$
para $i \in [0 \ \ldots \ 5]$ **hacer**
  para $j \in [0 \ \ldots \ 5]$ **hacer**
    $acumulador \leftarrow acumulador + img.data[y - 2 + i][x - 2 + j] * matriz\_convolución[i][j]$
  fin para
fin para
devolver $acumulador/159$

9.1.2. Detección de bordes (Canny)

Una vez que la imagen ya está suavizada, Canny procesa la imagen nuevamente aplicando dos matrices más de convolución. La matriz *convolucion_x* sirve para obtener la intensidad del gradiente en la dirección horizontal de la imagen y la matriz *convolucion_y* sirve para obtener la intensidad del gradiente en la dirección vertical de la imagen.

Dado que las convoluciones que se aplican en el algoritmo de Canny se aplican en un vecindario de $3x3$ al rededor del pixel a procesar, hay que tener en cuenta las mismas consideraciones que

en el Algoritmo 1 de suavizado pero dejando esta vez un borde de un pixel ya que el vecindario utilizado es mas pequeño que el anterior.

Una vez obtenidos los gradientes de cada pixel de la imagen, se utiliza un umbral para determinar cuales pixeles son bordes y cuales no. Es importante recalcar que si bien, el algoritmo original de Canny realiza non-maximum suppression - double thresholding y Edge-tracking by histeresis, en la implementación, estos pasos son reemplazados por un simple thresholding algorithm que consiste en comparar la intensidad del módulo del gradiente de cada pixel con un umbral para determinar si es borde o no. La decisión de modificar el algoritmo de Canny fue tomada considerando que:

- Canny genera bordes finos (un pixel de ancho) que generan menos votaciones, lo cual significa una perdidas en la redundancia de las votaciones. Si bien, podría parecer beneficioso tener menos bordes, menos redundancia y menos votaciones, esto es más propenso a errores numéricos que generen que los votos no se efectúen siempre en el centro del circulo, sino en alguno de los píxeles vecinos al mismo. Más aun, podría suceder que el centro del círculo esté entre dos píxeles y en tal caso los votos se reparten entre esos dos píxeles y sus vecinos (por errores numéricos). Es por esto que tener bordes anchos (mas de un pixel) que genere redundancia ayuda a que el método de detección de bordes sea mas robusto a errores. Utilizando non-maximum suppression - double-thresholding un pixel es considerado borde solo si, además de pasar los umbrales (double-thresholding) es un máximo local (non-maximum suppression), por lo que los bordes del filtro Canny original siempre son de ancho un pixel.
- Realizar un BFS para cada pixel considerado borde fuerte de la imagen, tiene un costo demasiado elevado que genera que el tiempo de procesamiento empeore en varios ordenes de magnitud, generando un mal funcionamiento del mismo al ser aplicado a videos. Además, por la naturaleza del BFS es muy difícil paralelizar esta tarea sin generar conflictos de concurrencia y la utilización de optimizaciones con SIMD tampoco se puede realizar ya que su costo no proviene de realizar múltiples veces las mismas operaciones.

Por último, se obtiene la dirección del gradiente que es necesaria para la implementación de la transformada de Hough y se devuelven todos los bordes encontrados en un vector de bordes. Los Algoritmos 4 y 3 muestran la implementación de Canny modificado en C++.

Algoritmo 3 Barrido de la imagen en el rango correcto para detectar los bordes

Entrada: Mat *img*, recorte *zonaAProcesar*.
Salida: un vector de bordes con los bordes detectados y los ángulos de sus gradientes.

1: $primera\_fila \leftarrow \text{mín}\{zonaAprocesar.i1, 2\}$
2: $ultima\_fila \leftarrow \text{máx}\{zonaAprocesar.i1, img.alto - 2\}$
3: $primera\_columna \leftarrow \text{mín}\{zonaAprocesar.j2, 2\}$
4: $ultima\_columna \leftarrow \text{mín}\{zonaAprocesar.j2, img.ancho - 2\}$
5:
6: $modulo\_gradientes \leftarrow$ *nueva matriz de doubles*
7: $ángulos \leftarrow$ *nueva matriz de doubles*
8: $bordes \leftarrow$ *nuevo vector de bordes*
9:
10: **para** $i \in [primera\_fila \ \ldots \ ultima\_fila]$ **hacer**
11: **para** $j \in [primera\_columna \ \ldots \ ultima\_columna]$ **hacer**
12:
13: $Gx \leftarrow convolucion\_gradiente(img, i, j, convolucion_x)$
14: $Gy \leftarrow convolucion\_gradiente(img, i, j, convolucion_y)$
15:
16: $modulo\_gradiente[i][j] \leftarrow \sqrt{Gx^2 + Gy^2}$
17: $ángulos[i][j] \leftarrow \arctan \frac{|Gy|}{|Gx|}$
18:
19: **fin para**
20: **fin para**
21:
22: **para** $i \in [primera\_fila \ \ldots \ ultima\_fila]$ **hacer**
23: **para** $j \in [primera\_columna \ \ldots \ ultima\_columna]$ **hacer**
24:
25: **si** $modulo\_gradientes[i][j] > umbral$ **entonces**
26: $bordes.agregar(borde(i, i, ángulos[i][j]))$
27: **fin si**
28:
29: **fin para**
30: **fin para**
31:
32: **devolver** *bordes*

Algoritmo 4 (*convolución_gradiente*) Convolución de gradientes para el pixel (x, y) de una imagen

Entrada: Mat *img*, int x, int y, int** *matriz_convolución*.
Salida: El resultado de la convolución para el píxel (x, y) de la imagen *img*

1: $acumulador \leftarrow 0$
2: **para** $i \in [0 \ \ldots \ 3]$ **hacer**
3: **para** $j \in [0 \ \ldots \ 3]$ **hacer**
4: $acumulador \leftarrow acumulador + img.data[y - 1 + i][x - 1 + j] * matriz\_convolución[i][j]$
5: **fin para**
6: **fin para**
7: **devolver** *acumulador*

9.1.3. Detección de círculos (Transformada de Hough)

La transformada de Hough utiliza los bordes detectados por Canny para generar el espacio de votaciones y luego utiliza un umbral para determinar si el pixel con más votaciones es o no un círculo válido. Dado que la detección de bordes no siempre detecta los todos los bordes del circulo de manera perfecta debido a diferentes factores tales como la luz, los colores y las texturas de la imagen y debido a errores numéricos, es necesario considerar un umbral menor a la cantidad teórica de votos que debe recibir un pixel que es el centro de un círculo. Además, dado que los círculos de las imágenes son círculos generados por píxeles, podemos saber cual es la cantidad teórica de votaciones que debería recibir y multiplicarla por un factor del error realizado en la detección de bordes, este factor es siempre menor o igual a uno, ya que en caso contrario se estaría considerando un umbral mayor a la cantidad máxima de votos que puede recibir un pixel. La cantidad teórica de votos que recibe el pixel del centro de un círculo es cuatro veces el radio del circulo, ya que en cada cuadrante necesita una cantidad de píxeles igual al radio para dibujarlo. El Algoritmo 5 muestra la implementación de Hough en c++.

Algoritmo 5 Detección de cículos mediante la transformada de Hough

Entrada: vector de bordes *bordes*, vector de int *radios*, double $factor$.
Salida: Si lo encuentra, devuelve el círculo encontrado.

```
votos ← nuevo espacio de acumulación inicializado con ceros
máximo ← (0, 0)

para r ∈ radios hacer
    para b ∈ bordes hacer
        voto1.i ← b.i + sen(b.angulo) ∗ r
        voto1.j ← b.j + sen(b.angulo) ∗ r

        si voto1 está contenidoenlaimagen entonces
            votos[i][j] ← votos[voto1.i][voto1.j] + 1
        fin si

        voto2.i ← b.i + sen(b.angulo) ∗ r
        voto2.j ← b.j + sen(b.angulo) ∗ r

        si voto2 está contenidoenlaimagen entonces
            votos[i][j] ← votos[voto2.i][voto2.j] + 1
        fin si
    fin para

    para i ∈ filas de votaciones hacer
        para j ∈ columnas de votaciones hacer
            si votaciones[i][j] > votaciones[máximo[0]][máximo[1]] entonces
                máximo ← (i, j)
            fin si
        fin para
    fin para

fin para
si votaciones[máximo[0]]máximo[1]] > radio ∗ 4 ∗ factor entonces
    devolver círculo(máximo, radio)
fin si
```

9.2. Implementación en SIMD

En esta sección no detallaremos la utilización de los registros xmm ya que consideramos que redactar dicho detalle es tan declarativo como el código mismo que puede ser observado desde los archivo fuentes del trabajo. En cambio detallaremos las ideas principales de como utilizar SIMD para optimizar ciertas partes del código del programa. SIMD agrega optimizaciones cuando lo que se precisa es realizar múltiples veces una misma acción sobre un conjunto de datos que son levantados de memoria, paralelizando estas operaciones y minimizando las llamadas a memoria, lo cual es una operación costosa. Dado que se cuenta con una cantidad constante de registros xmm de 128 bits con capacidad de realizar operaciones en paralelo, la complejidad del algoritmo no se ve reducida más que por una constante, ya que la cantidad de operaciones en paralelo que se pueden realizar esta acotada por la cantidad de registros xmm.

Mirando los métodos utilizados, se puede ver que los mismos generan mucha lectura a memoria en dos lugares principales que son los filtros que aplican matrices de convolución (suavizado y búsqueda de gradientes de Canny). Además, en estos lugares, las operaciones a realizar son siempre las mismas aplicadas a puntos contiguos de la imagen levantados de memoria, lo que permite optimizar estas secciones mediante SIMD fácilmente. Además, dado que los píxeles de la imagen están representados por uchars de 8 bits, podemos levantar en una misma llamada a memoria 16 píxeles (8 Bits x 16 = 128 Bits). Aunque seria óptimo procesar 16 píxeles por iteración, solo podemos procesar 4 píxeles por vez. La perdida de los restantes 12 píxeles se debe a dos causas:

1. Para procesar la convolución de un pixel se precisa de una vecindad del mismo que puede ser de distintos tamaños. Esto genera que los x píxeles del comienzo y los x píxeles del final no puedan ser procesados por falta de sus vecinos.
2. Dado que las operaciones de SIMD trabajan en su mayoría con double words (32 Bits), es necesario desempaquetar los píxeles de 8 Bits a 32 Bits acotando la cantidad de píxeles que un registro xmm puede alojar de 16 a 4. Dado además la acotada cantidad de registros xmm disponibles, algunos píxeles se ven descartados por el proceso de desempaquetado.

Por otra parte, para la implementación de la transformada de Hough no sólo no se requiere implementar muchas llamadas a memoria ya que la imagen no es recorrida, sino que las operaciones que se realizan generan resultados que están lejos dentro del espacio de acumulación. Esto se debe a que cada borde vota dos píxeles que se encuentran a un diámetro de distancia entre si y en dirección del gradiente del borde que en el mejor caso (cuando se procesan los bordes del círculo buscado) varia para cada punto de borde. Es por esto que la función de la transformada de Hough no fue optimizada mediante la utilización de SIMD. También tuvimos problemas para implementar SIMD para obtener los ángulos de las direcciones de los gradientes ya que los mismos requieren la utilización de la función arco tangente, la cual no se encuentra disponible dentro del set de instrucciones SIMD de intel.

9.2.1. Suavizado mediante matriz de convolución

Con el fin de optimizar la cantidad de lecturas a memoria, y dado que los vecindarios utilizados para la convoluciones entre píxeles contiguos comparten muchas posiciones, se itera la imagen a procesar de a cuatro píxeles horizontales y para cada iteración, se levanta con una única llamada a memoria, todos los píxeles de la imagen necesarios para procesar la convolución de los siguientes cuatro píxeles de la imagen procesada. Uno de los inconvenientes que surgen, es que el set de instrucciones de SIMD que necesitamos usar trabaja con double words, mientras que los píxeles de la imagen estan en uchar (8 bits) es por esto que tenemos que desempaquetar dos veces y de los 16 pixeles que levantamos con una llamada a memoria, nos quedamos con los primeros 8 utilizando para esto dos registros xmm, ya que en cada registro xmm entran 4 double words. La Figura 13 muestra como es este proceso.

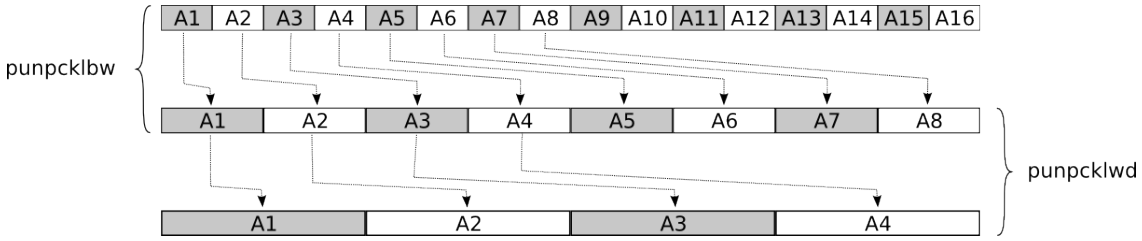


Figura 13: Desempaquetado de 16 píxeles levantados en 8 bits c/u a 4 píxeles de 32 bits c/u.

Dado que la fila uno de la matriz de convolución a utilizar es igual a la fila cinco de la misma matriz, y dado que la fila dos es igual a la fila cuatro, utilizando 6 registros xmm podemos levantar una sola vez para todo el proceso de suavizado la matriz de convolución de memoria. Se necesitan seis registros ya que la matriz tiene cinco columnas y cada registro puede guardar cuatro columnas, por lo que se necesitan dos registros por fila de la matriz, y dado que dos filas se repiten, en vez de tener cinco filas, tenemos tres.

Luego, para cada iteración del ciclo, una vez levantada la parte de la imagen a utilizar de la memoria se procede a multiplicar cada pixel del vecindario con cada pixel de la matriz de convolución y realizar el sumatorio de todos los resultados. Para esto se utilizan primero las multiplicaciones en vertical que permiten multiplicar los píxeles de la imagen con los valores de la matriz de convolución y luego se procede a realizar una combinación de sumas horizontales y verticales para sumar todos los resultados alojados horizontalmente en un mismo registro y entre registros. Una vez realizada la convolución para un pixel, se realiza un shift a izquierda sobre los registros que guardan los píxeles de la imagen, para de esta forma procesar la convolución en el pixel siguiente sin tener que volver a levantar dato de la imagen en la memoria, lo cual es muy costoso. Este proceso se realiza cuatro veces hasta tener que iterar y volver a levantar los siguientes píxeles de la imagen.

9.2.2. Obtencion de los gradientes para Canny

La idea de como utilizar SIMD para conseguir los gradientes de Canny es muy similar a la del procesamiento para suavizado ya que ambos métodos se realizan mediante convoluciones, lo que cambia es el tamaño del vecindario a utilizar, generando menos problemas con la cantidad de registros xmm disponibles. La figura 14 muestra el proceso entero de convolución para un pixel utilizando un vecindarion de $3x3$ como es el caso para las convoluciones de obtención de gradientes de canny.

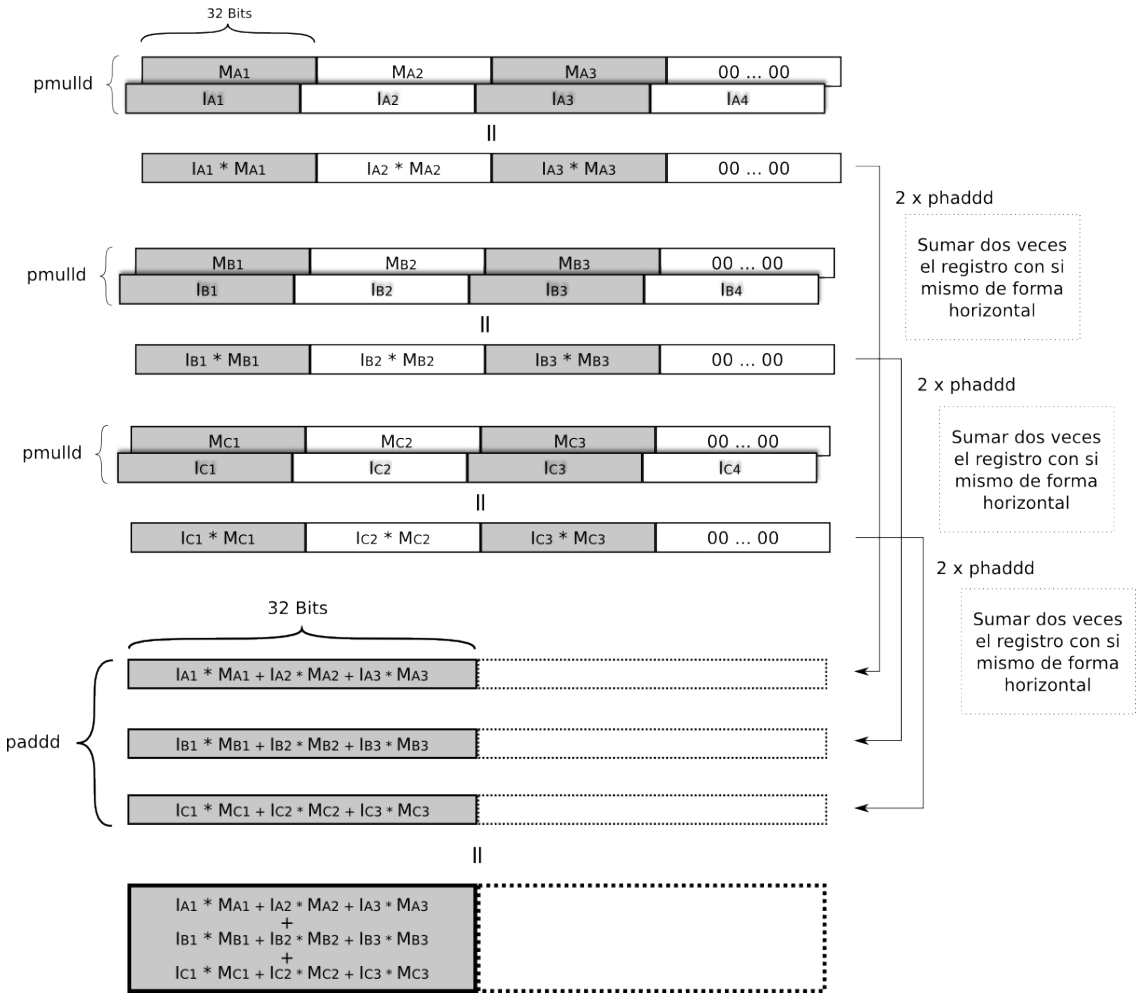


Figura 14: Proceso de convolución de un pixel en SIMD.

10. Experimentación

Toda la experimentación realizada fue ejecutada con una computadora con sistema operativo Ubuntu 14.04, 8 GB de memoria RAM y procesador Intel Core i7 - 4500 @ 1.80GHz x 4.

Todos los experimentos se realizaron utilizando los siguientes parámetros salvo especificaciones particulares de cada experimento.

- **Relación votaciones radio:** 0.3
- **Umbral bordes:** 30
- **Radio mínimo:** 100
- **Cantidad de radios:** 1
- **Radio step:** 1
- **cantidad de threads:** 1

10.1. Tiempos de ejecución

Para esta sección todos los experimentos salvo los de la sección "Tiempo de ejecución dependiendo la cantidad de threads", fueron realizados utilizando un solo thread para poder de esta forma obtener un análisis completo del algoritmo independiente de la optimización realizada utilizando programación concurrente.

10.1.1. Comparación entre C++ y Assembly

Como primera instancia de experimentación, quisimos ver dos aspectos fundamentales del algoritmo; su complejidad temporal respecto de las dimensiones y bordes de las imágenes y la mejora temporal obtenida en la implementación con SIMD (en adelante simplemente assembly). Dado que el algoritmo ejecuta convoluciones sobre los píxeles y luego votaciones sobre los bordes encontrados, se espera que la complejidad temporal sea lineal respecto del tamaño de la entrada (cantidad de píxeles) y dado que la implementación en assembly mejora el algoritmo solo en una constante esperamos que la complejidad de Assembly sea la misma que la complejidad en C++ dividida por alguna constante.

Para esto, utilizamos imágenes cuadradas grises con un círculo de radio la mitad del ancho de la imagen con una cruz en el medio. El círculo y la cruz tienen dimensión proporcional al ancho de la imagen para así obtener una cantidad de bordes proporcional a la dimensión de la imagen. Las dimensiones utilizadas van desde los $100x100$ hasta los $1000x1000$ píxeles, la Figura 15 muestra un ejemplo de las imágenes utilizadas.

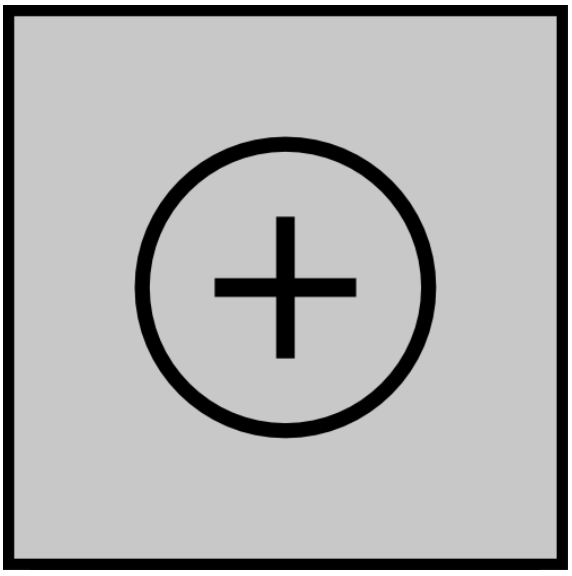


Figura 15: Imagen de dimensión $500x500$ utilizada para el experimento de Comparación entre C++ y Assembly.

Las Figuras 16, 17 fueron obtenidas con el mismo conjunto de datos de una misma corrida, ambos gráficos muestran los resultados de los tiempos de ejecución de C++ y Assembly para las distintas dimensiones/cantidad de píxeles de imágenes. La Figura 16 nos sirvió para corroborar que efectivamente tras efectuar una regresión lineal casi perfecta sobre los tiempos de corrida según la cantidad de píxeles de la imagen, el tiempo de ejecución aumenta de manera lineal. Esto nos permite reforzar (debido a la fuerte correlación existente entre complejidad temporal y tiempo de ejecución) que el algoritmo tiene complejidad temporal lineal sobre la cantidad de píxeles. Además de la Figura 17 podemos observar que la complejidad lineal de Assembly reduce la complejidad de C++ en una constante menor que 4 lo que tiene sentido ya que si bien en Assembly se procesan de a 4 píxeles por vez, la necesidad de computar la función trigonométrica arctan no disponible en SIMD genera un cuello de botella para el procesamiento en paralelo, impidiendo reducir en 4 la totalidad del algoritmo. Tambien podemos ver este resultado en las regresiones linealis (Figuras 16) ya que se puede ver como Assembly tiene una pendiente menor con punto de intersección en el origen rspecto de C++, lo cual nos muestra que efectivamente Assembly reduce la complejidad solamente en una constante.

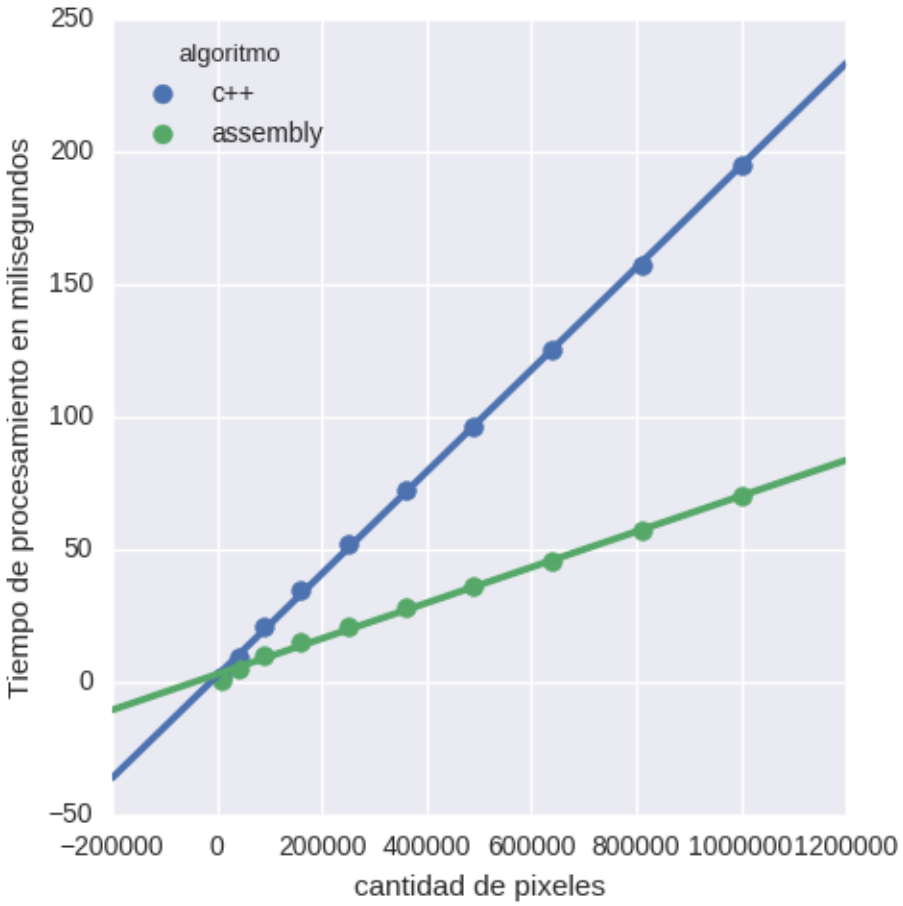


Figura 16: Regresión lineal del tiempo de ejecución en milisegundos para distintas cantidades de pixeles tanto en c+ como en assembly. Los tiempos se midieron corriendo 1000 instancias y tomando el promedio.

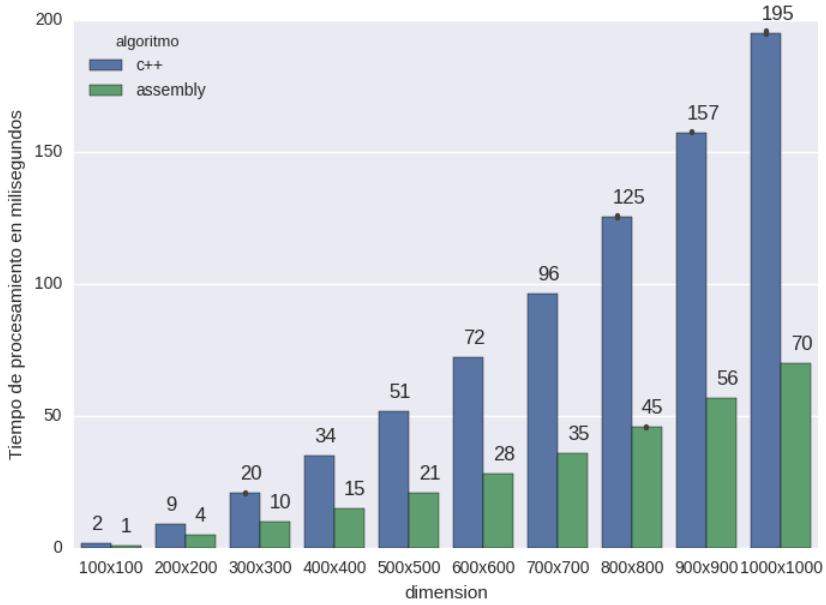
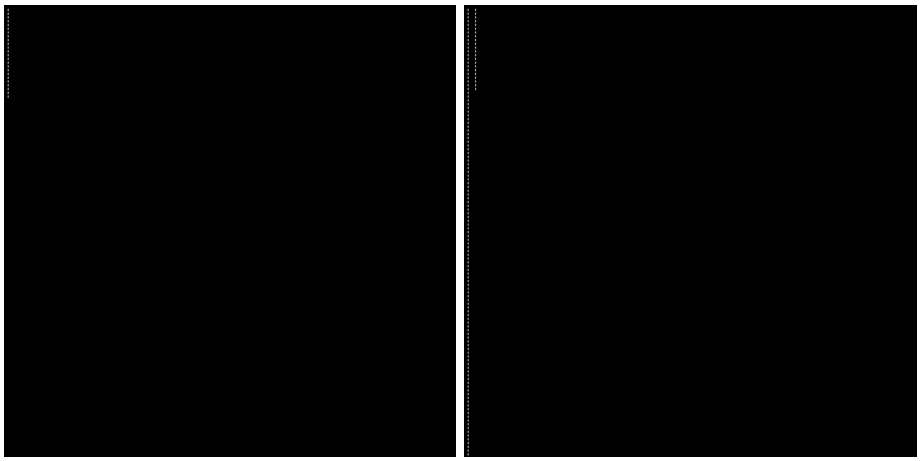


Figura 17: Tiempo de ejecución en milisegundos para distintas dimensiones de imágenes en píxeles tanto en c+ como en assembly. Los tiempos se midieron corriendo 1000 instancias y tomando el promedio.

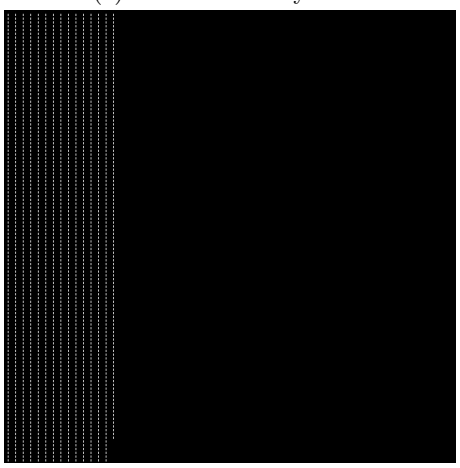
10.1.2. Tiempos respecto la cantidad de bordes

Dado que la transformada de hough utiliza los bordes detectados por Canny para generar un espacio de votaciones donde buscar el círculo en la imagen, decidimos corroborar los tiempos de procesamiento aumentando la cantidad de bordes de la imagen a procesar y corroborar que Assembly sigue mejorando C++ respecto de esta variable. Para poder medir los tiempos respecto de la cantidad de bordes generamos una imagen inicial toda en negro mediante OpenCV-2.4.9 de $600x600$ píxeles de dimensión e iteramos de 0 a 3000 agregando en cada iteración una raya vertical de 2 píxeles de ancho y 5 píxeles de alto. De esta forma sabemos que en cada iteración agregamos 10 píxeles marcados como bordes. La Figura 18 tiene algunos ejemplos de como se ven las imagenes con distintas cantidades de rayas que se generaron con OpenCV-2.4.9.

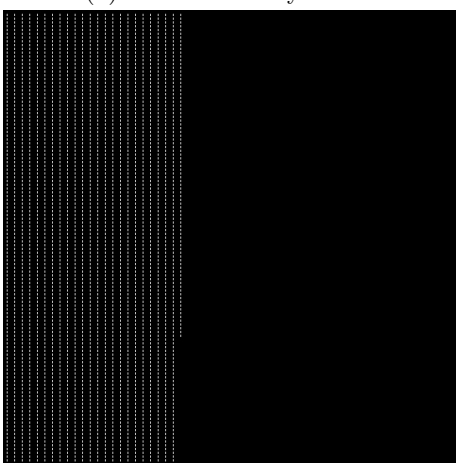


(a) Cantidad de rayas 23.

(b) Cantidad de rayas 140.



(c) Cantidad de rayas 1778.



(d) Cantidad de rayas 2822.

Figura 18: Imágenes con distintas cantidades de rayas de 2x5 píxeles.

Mirando la Figura 19 podemos corroborar que la complejidad del algoritmo es lineal respecto de la cantidad de bordes y que la mejora generada por Assembly es consistente a diferentes cantidades de bordes, mantendiendose siempre constante. Es importante notar que a diferencia de la Figura 16 en este gráfico las rectas generadas por C++ y Assembly tienen la misma pendiente, lo cual indica que Assemly no reduce la complejidad respecto de los bordes en una constante divisora sino que lo mejora en una constante que resta (sin importar la cantidad de bordes que tenga la imagen, la distancia entre C++ y Assembly es la misma). Esto se da porque la complejidad del algoritmo está dada por la dimensión de la imagen más la cantidad de bordes; Assembly reduce el tiempo de ejecución al procesar los píxeles de la imagen de a 4 por vez en la etapa de convoluciones para detectar bordes, dividiendo así el tiempo de ejecución de esta etapa en una constante para distintas dimensiones de imágenes, pero una vez obtenidos los bordes, el algoritmo en Assembly no mejora significantemente la complejidad, manteniendo la diferencia ya obtenida en las etapas de convoluciones.

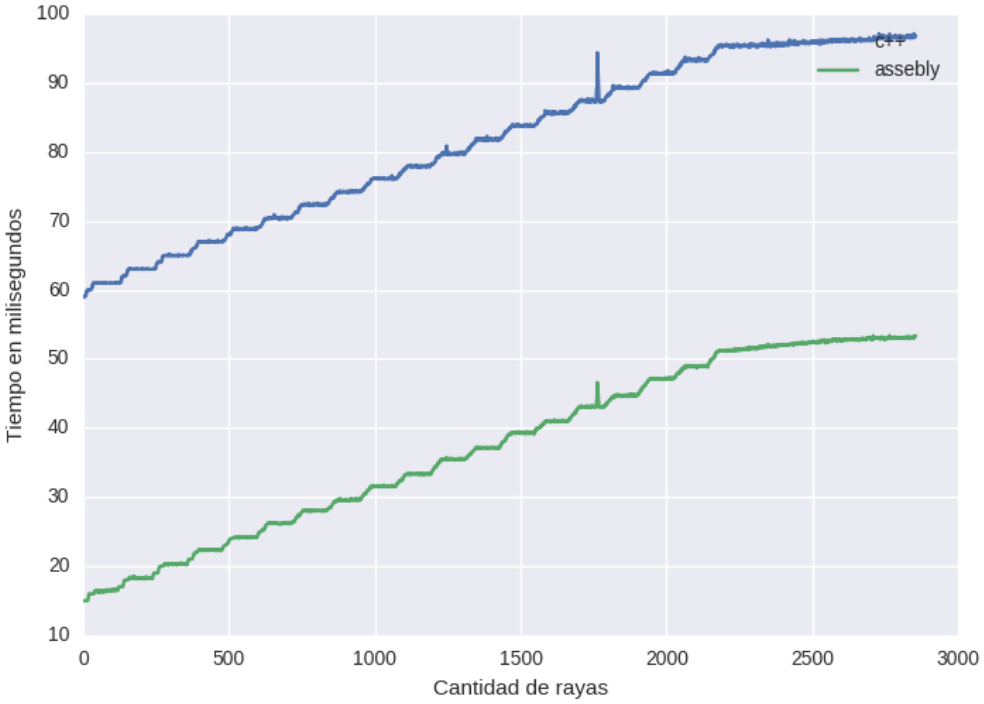


Figura 19: Tiempo de ejecución en milisegundos para distintas cantidades de rayas tanto en c+ como en assembly. Los tiempos se midieron corriendo 100 instancias y tomando el promedio.

10.1.3. Tiempo de ejecución dependiendo el umbral para la detección de bordes

Dado que como ya vimos en la experimentación anterior "Tiempos respecto la cantidad de bordes", la cantidad de bordes afecta el tiempo de ejecución pero mantiene la relación del tiempo de ejecución entre C++ y Assembly, nos preguntamos como podría afectar entonces a ambas implementaciones el umbral utilizado por Canny para determinar cuales píxeles de la imagen son bordes y cuales no. Si bien sabemos que a mayor valor de umbral menor cantidad de bordes detectados y por lo tanto menor tiempo de ejecución, queríamos corroborar que la diferencia entre C++ y Assembly nuevamente se mantuviese constante utilizando un razonamiento parecido al del experimento anterior "Tiempos respecto la cantidad de bordes". La imagen utilizada para este experimento es la ya bien conocida imagen de lena.

Podemos ver en la Figura 20 por un lado que el tiempo de ejecución para ambas implementaciones disminuye logaritmicamente. Esto puede deberse probablemente a los valores obtenidos de los módulos de los gradientes y podría variar para distintas imágenes. No seguimos experimentando con otras imágenes porque consideramos que de todas formas no es un resultado que nos pueda revelar demasiado sobre el algoritmo y sus implementaciones (C++, Assembly). Podemos ver por otro lado que para valores chicos del umbral la diferencia entre C++ y Assembler parece disminuir. Si bien hay muchas posibles explicaciones para este fenómeno, nosotros no pudimos concluir nada significativo y decidimos dejar este gráfico solo a modo de muestra de consistencia para mostrar que Assembly mejora C++ independientemente del valor de este parámetro.

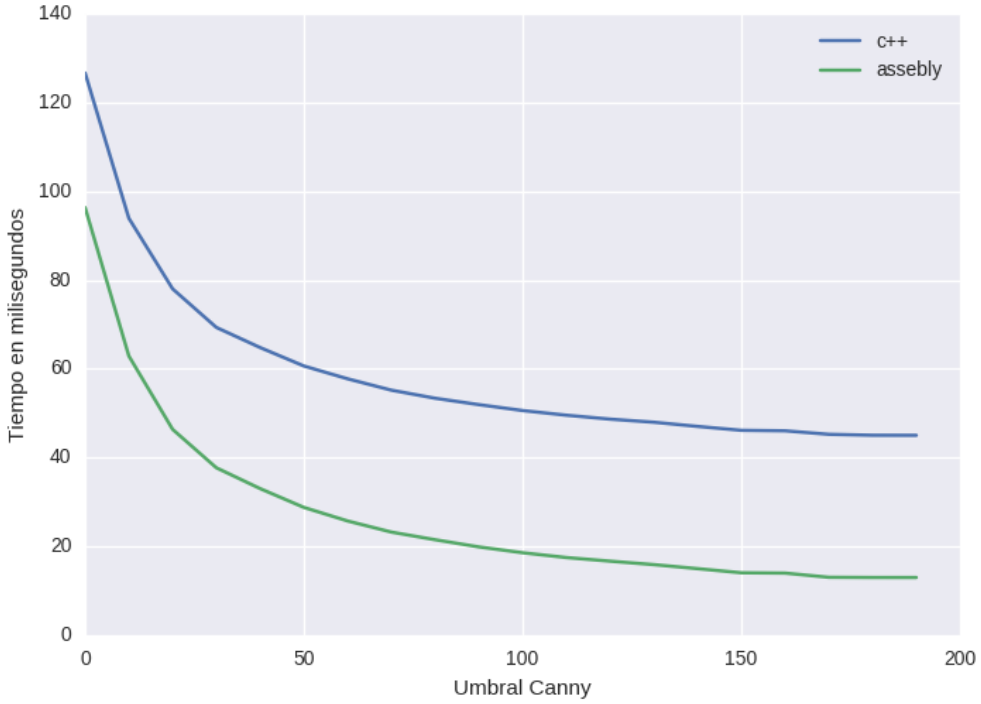


Figura 20: Tiempo de ejecución en milisegundos para distintos valores del umbral utilizado en Canny para la detección de bordes tanto en c+ como en assembly. Los tiempos se midieron corriendo 1000 instancias y tomando el promedio.

10.1.4. Tiempo de ejecución dependiendo la relación votaciones radio de la transformada de Hough

El parámetro de relación votaciones radio es utilizado por la transformada de hough para determinar cuales puntos del espacio de votaciones corresponden a un círculo y cuales no, Dado que este parámetro es utilizado sobre el final del algoritmo y que su resultado no repercute en etapas posteriores del mismo, no tiene mucho sentido esperar que halla variaciones en el tiempo de ejecución, pero nuevamente, realizamos este experimento a modo de verificación. Utilizamos nuevamente la imagen de lena para ir variando el parámetro relación votaciones radio y verificar que Assembly mejora C++ de forma consistente respecto del mismo. La Figura 21 muestra cómo el tiempo de ejecución para ambas implementaciones se mantiene constante y que la mejora de Assembly es independiente de este parámetro concluyendo una verificación exitosa.

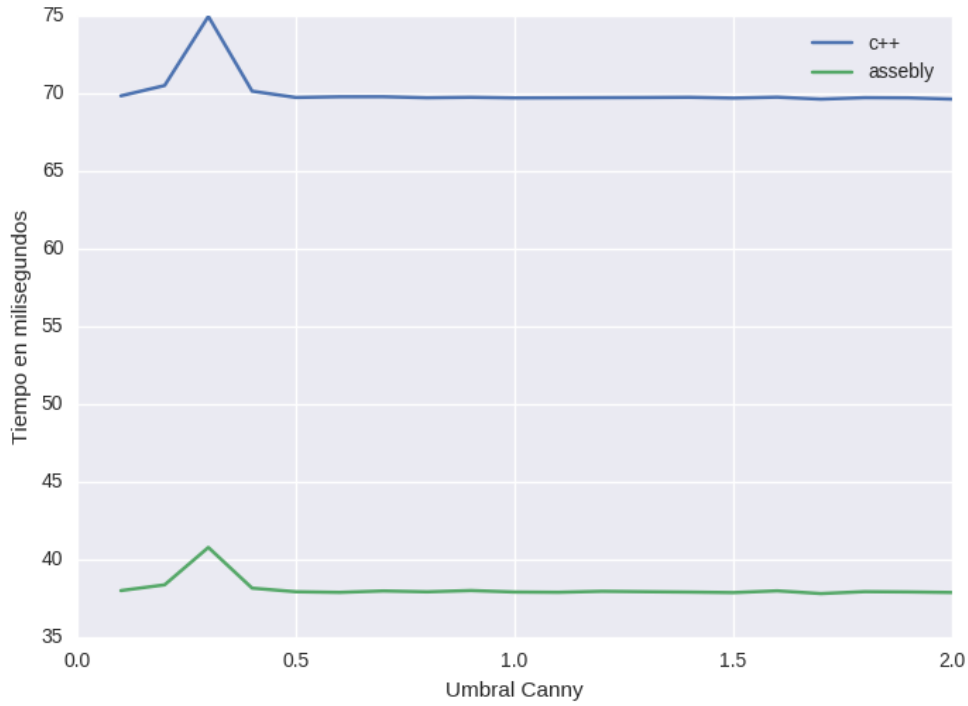


Figura 21: Tiempo de ejecución en milisegundos para distintos valores de la relación votaciones radio utilizado en la transformada de hough tanto en c+ como en assembly. Los tiempos se midieron corriendo 1000 instancias y tomando el promedio.

10.1.5. Tiempo de ejecución dependiendo la cantidad de threads

Como última experimentación, decidimos ver como varía el tiempo de ejecución tanto en C++ como en Assembly al variar la cantidad de threads utilizados. La idea de implementar threads en este trabajo fue poder analizar cómo afecta la utilización de programación concurrente a la utilización de SIMD, o si ambas técnicas pueden ser utilizadas en conjunto obteniendo mejoras provenientes de ambas optimizaciones. Como se especifica al comienzo de la sección "Experimentación", todos los experimentos fueron realizados en una procesador Intel Core i7 - 45000 @ 1.80GHz x 4, por lo que la mayor cantidad de procesos concurrentes que se pueden realizar son cuatro, uno por cada core. Para este experimento se corrió el algoritmo utilizando la imagen (a) de la Figura 23, ya que la misma tiene muchos bordes sobre el lado izquierdo y pocos sobre el lado derecho, generando diferencias en los bloques que procesan cada thread. Esto sirve para evitar realizar las mediciones en un escenario demasiado monótono.

La Figura 22 muestra cómo la utilización de programación concurrente mejora el tiempo de ejecución del algoritmo respecto de la cantidad de cores del procesador utilizados para la implementación en C++. Esto se ve ya que luego de cuatro cores, el tiempo de ejecución no mejora. Lo más destacable de este experimento es que la implementación en Assembly sigue siendo mejor que la de C++, aunque esta última mejora su performance y Assembly no. Esto podría dar un indicio de que ambas técnicas no son compatibles para ser utilizadas en conjunto y que no se obtiene ningún doble beneficio de utilizar estas dos técnicas en conjunto. Una posible explicación a este fenómeno podría ser la utilización de los registros xmm por los distintos cores. Desafortunadamente no conseguimos ningún procesador con más cores para evaluar la evolución de este fenómeno y comprobar si la técnica de programación concurrente y SIMD son incompatibles o no.

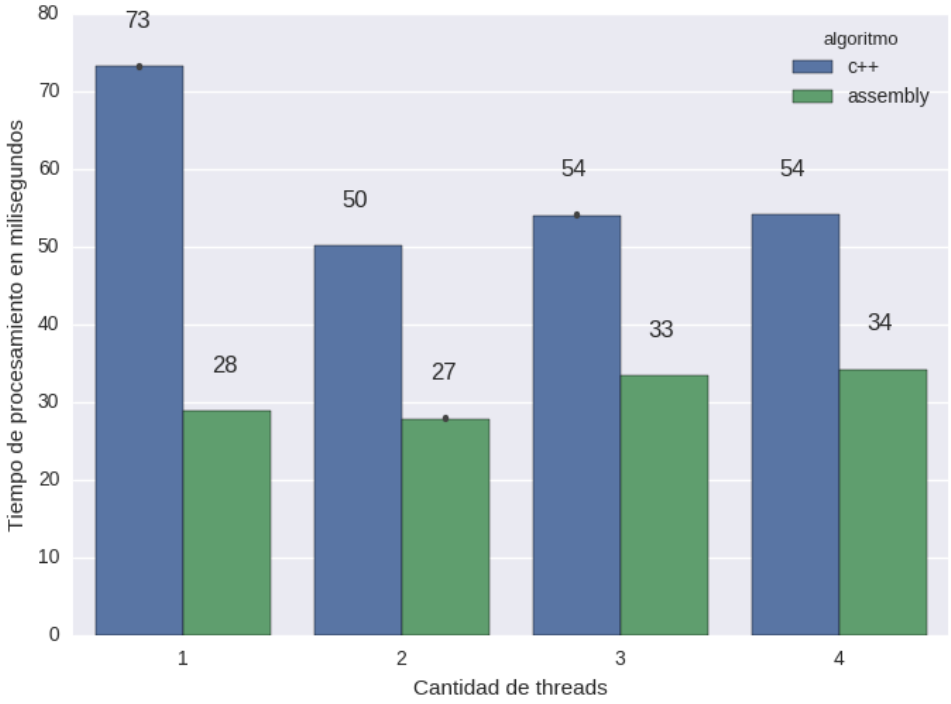


Figura 22: Tiempo de ejecución en milisegundos para distintas cantidades de threads en c+ como en assembly. Los tiempos se midieron corriendo 1000 instancias y tomando el promedio.

Si C++ puede seguir mejorando su performance a mayor cantidad de threads, tal vez pueda llegar a ganarle a Assembly en este caso podría ganarle o no a la utilización de SIMD.

10.2. Robustes del algoritmo

La idea de esta sección es estudiar cuán robusto es el algoritmo propuesto por la transformada de hough. Para esto realizamos un único experimento donde tomamos distintas imágenes con distintas cantidades de ruido y corrimos el algoritmo sobre esas imágenes para estudiar cuanto ruido es necesario para afectarlo. La Figura 23 muestra las imágenes utilizadas para este experimento.

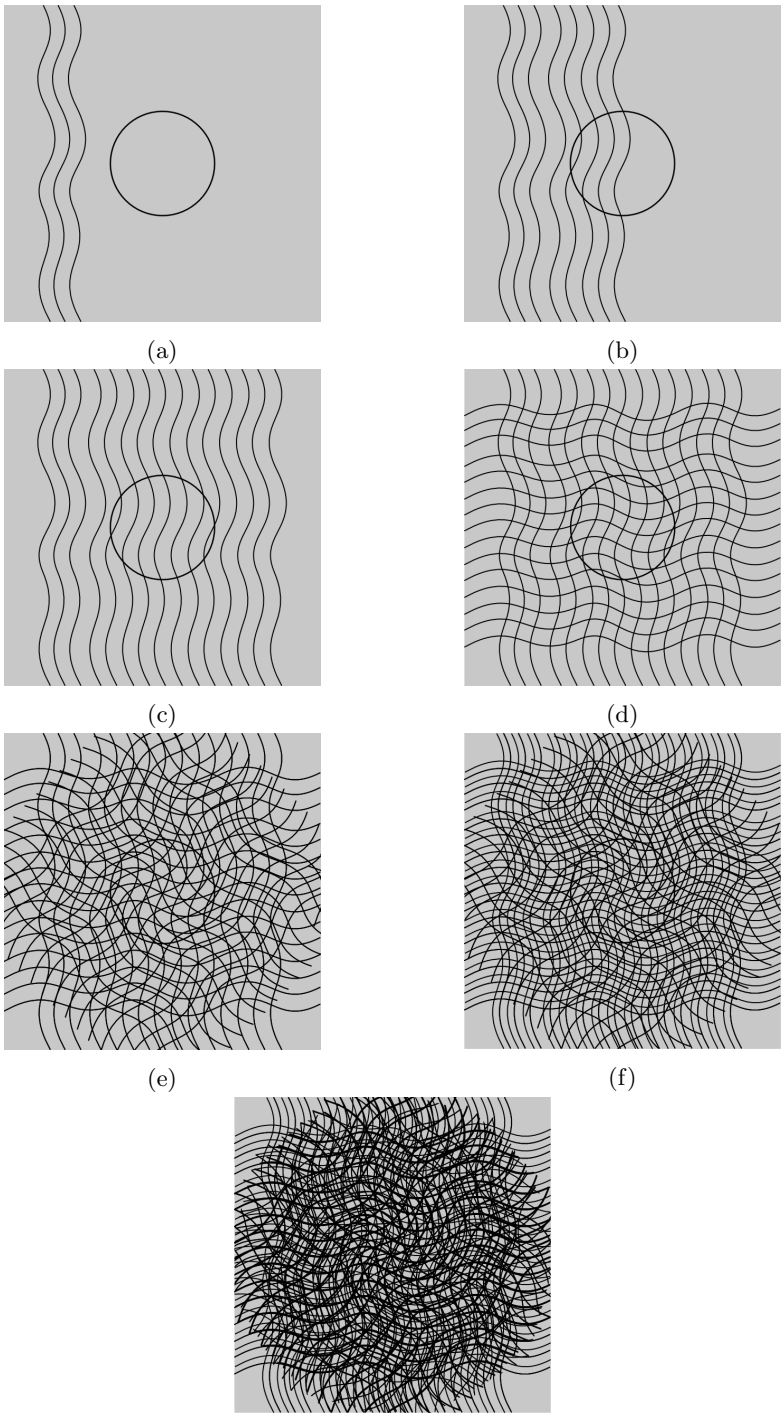


Figura 23: Imágenes con distintas cantidades de ruido y un círculo de radio 100 píxeles en el centro.

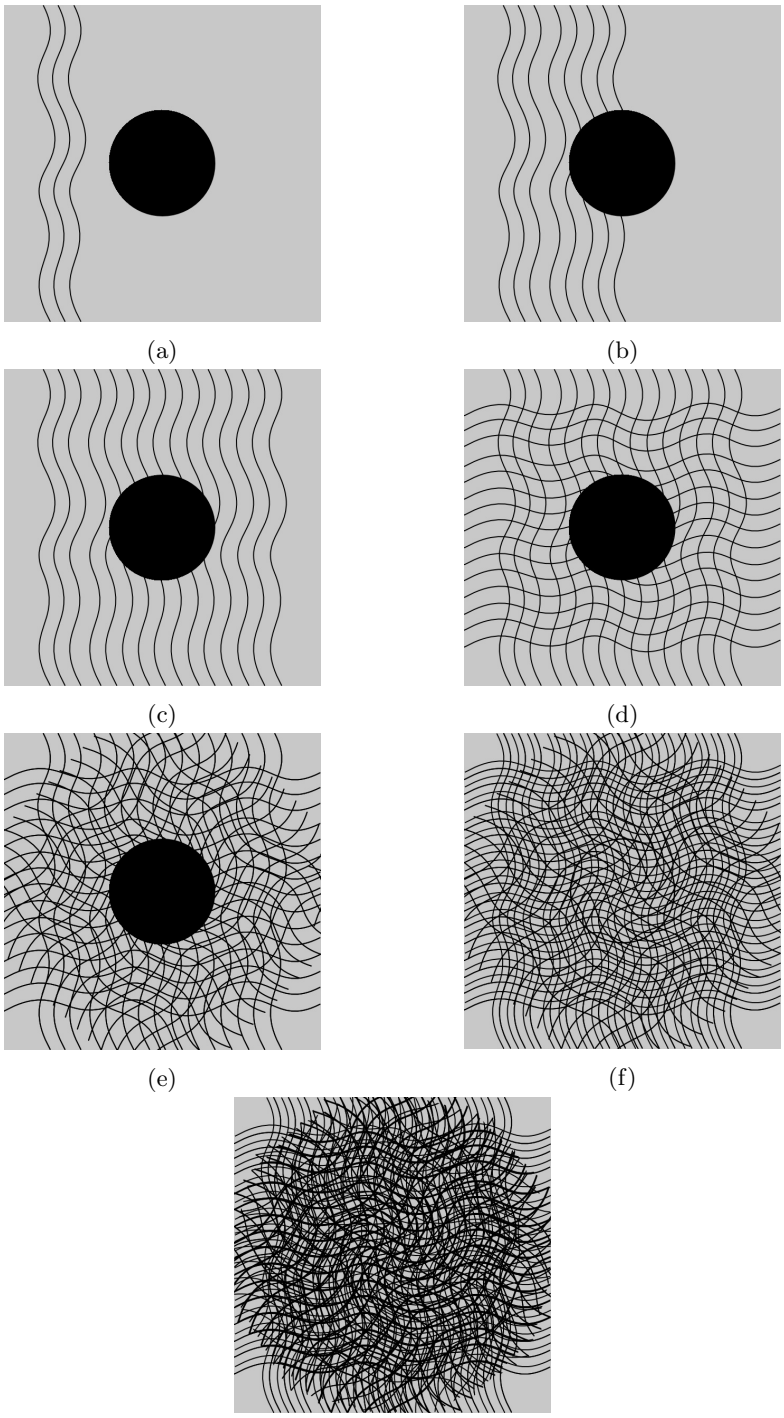


Figura 24: Resultado obtenido por el algoritmo utilizando la implementación de C++.

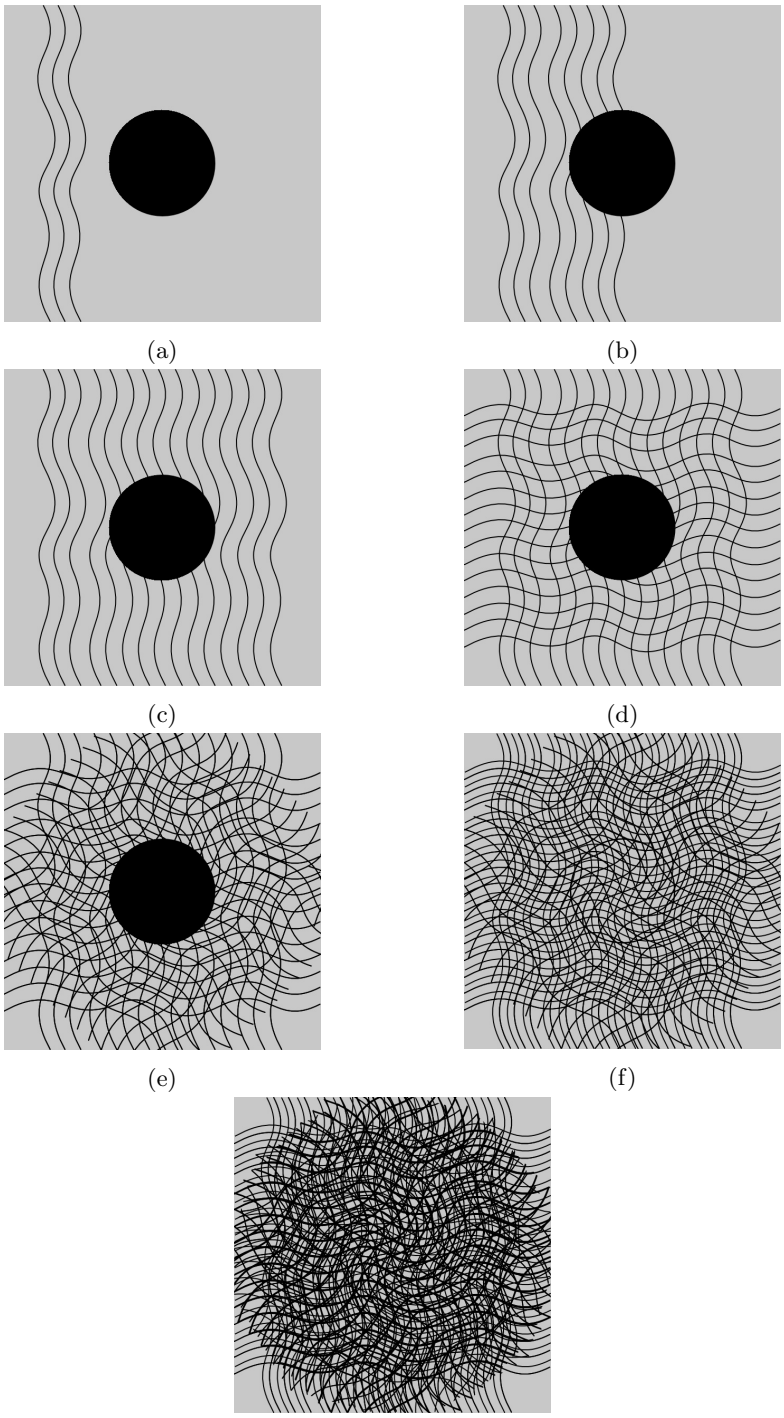


Figura 25: Resultado obtenido por el algoritmo utilizando la implementación en Assembly.

Como se puede observar en las Figuras 24, 25 el algoritmo es bastante robusto mismo cuando el círculo esta atravesado por lineas, no solo de forma horizontal y vertical sino también de forma diagonal. Si bien podría ser esperado que el algoritmo detecte el círculo de la imagen (f), también es razonable que no sea así pues la cantidad de ruido ya es muy elevada y el mismo no es un escenario común.

10.3. Experimentación visual

Para poder mostrar y estudiar como afectan los distintos parámetros del algoritmo implementado realizamos unas capturas de pantallas con el programa corriendo mientras se van modificando los parámetros del mismo. Los mismos están subidos a Youtube pero no aparecen visibles para aquellos que no tengan los links a los mismos. A continuación damos dichos links.

1. https://www.youtube.com/watch?v=RV6Hd-vPFzA
2. https://www.youtube.com/watch?v=tyOYZ9ylDFY
3. https://www.youtube.com/watch?v=idxXcaftM4w
4. https://www.youtube.com/watch?v=YkLsJANmK1Y

En los videos de los links 1 y 2 podemos ver como afectan los parámetros cantidad de radios y radio step a la fluidez del programa. Dado que la transformada de Hough utiliza un radio ya conocido para buscar círculos que fiteen con ese radio, si solo consideramos un radio fijo entonces el programa solo va a encontrar el círculo de forma perfecta cuando el mismo tenga este radio. Lo que se ve en estos dos experimentos (video 1 y 2) es que el programa también detecta el círculo mismo si el radio del mismo no es exactamente el mismo. En estos casos se ve que encuentra el círculo sobre un borde del círculo real y dado la diferencia entre los radios (real y esperado) el programa establece de forma errada el centro del círculo encontrado, aunque esto puede ser visto como un error, para los efectos de seguir la trayectoria de un círculo se puede considerar a este error como un error deseado, ya que encuentra el círculo solo que no lo parametriza correctamente lo cual puede no ser muy importante si por ejemplo, solo se desea utilizar el programa para dibujar en la pantalla. Este efecto se puede ver en las primeras pruebas en las cuales se utilizan una cantidad baja de radios a considerar. A medido que se va aumentando la cantidad de radios a considerar, el programa va encontrando cada vez mejor las parametrización del círculo real y el programa se vuelve mas adaptativo, permitiendo acercar y alejar la esfera de la cámara sin perdida de precisión en la detección del mismo y su correcta parametrización.

Otro factor que entra en juego es el radio step. Este parámetro indica cuál es la separación a considerar entre los distintos radios utilizados. Así por ejemplo, si se tiene un radio step de 1 pixel y se tiene una cantidad de 10 radios arrancando de un radio mínimo de 25 entonces se consideran todos los radios de $25, 26, 27, 28, ..., 35$ pero si se tiene un radio step de 5 se consideran los radios $25, 30, 35, 40, ..,75$. Como se puede ver, al agrandar el radio step se necesitan menos cantidad de radios para obtener una mayor varianza entre los radios considerados. Tomando esto en cuenta, y mirando los videos 1 y 2, podemos concluir que al variar la cantidad de radios manteniendo un radio step bajo, no se mejora mucho la precisión de la detección de círculos mientras que empeora considerablemente la fluidez del programa ya que para cada radio distinto considerado, es preciso correr la transformada de hough nuevamente. Por otro lado, considerando una cantidad razonable de radios con un radio step más amplio se consigue mayor precision en la parametrización del círculo encontrado sin tanta perdida en la fluidez del mismo y sin la necesidad de mantenerse siempre a la exacta misma distancia de la camara.

Es algo que vale la pena notar que para Assembly se necesita aumentar más la cantidad de radios para obtener la misma perdida de fluidez que se obtiene para valores mas bajos de cantidad de radios en C++. Este resultado es consistente con la experimentación de tiempos donde

Assembly mejora C++.

Por otro lado, los videos 3 y 4 muestran la performance para detectar círculos por el programa al ir aumentando los umbrales utilizados. Como se puede ver en los videos a medida que se aumentan estos parámetros, la fluidez no disminuye. Esto se condice con los experimentos realizados sobre el tiempo de ejecución variando estos umbrales. Lo que sí se puede observar, es que tanto en el video 3 aumentando la relación de votaciones radio como en el video 4 aumentando el umbral usado en Canny para la detección de bordes, cada vez cuesta más lograr que detecte el círculo teniendo que posicionar cada vez mejor la esfera, en una distancia correcta y teniendo cuidado de no generar ruidos, como por ejemplo el reflejo de la luz o el movimiento de la esfera. Eventualmente, se consigue llegar a valores para los cuales el programa deja de detectar círculos en su totalidad. Este resultado tiene sentido ya que aumentar los umbrales es ser mas estricto con los resultados que consideramos válidos y dado que uno puede ser tan estricto como quiera, entonces siempre hay un valor lo suficientemente alto como para que ningún resultado pueda ser considerado válido.

11. Conclusiones

En el presente trabajo introducimos un programa que se basa en la transformada de Hough para detección y seguimiento de círculos en tiempo real. Dado que es necesario poder procesar imágenes en tiempo real, se introducen optimizaciónes que permiten mejorar la velocidad del algoritmo. Una de esas optimizaciones y la más importante es la utilización de SIMD la cual nos permitió reducir considerablemente la constante de la complejidad del algoritmo al procesar las convoluciones en paralelo. De todas formas dado que el ojo humano deja de reconocer cambios a aproximadamente 30 FPS (Fotogramas por segundos) y dado que para una configuración de los parámetros razonables la implementación en C++ ya alcanza esta velocidad de procesamiento, no es estrictamente necesaria la implementación en SIMD si no se precisa gran nivel de exactitud en la detección del círculo y ambas implementaciones parecen performar igual a efectos visuales del ojo humano. Pero con la finalidad de poder ajustar los parámetros a fin de conseguir mayor precisión en el resultado y obtener un seguimiento de esferas mas flexible a variaciones como el radio de la esfera, la implementación en SIMD jugó un rol fundamental permitiendo aumentar la cantidad de radios a considerar sin perder la performance del programa.