Desarrollo de un sistema de asistencia visual para un Manipulador automático programable (robot industrial)

Ing. Martín Ferreyra Biron, Ing. Alberto Miguens, Ing. Fernando Szklanny
Departamento de Ingeniería e Investigaciones Tecnológicas
Universidad Nacional de La Matanza
mferreyra@unlam.edu.ar , amiguens@unlam.edu.ar , fszklanny@unlam.edu.ar

Resumen

El presente trabajo hace referencia al desarrollo de un sistema de procesamiento digital de imágenes, basado en cámaras de alta velocidad, para incorporar en un autómata programable, capaz de moverse articuladamente en seis ejes, y destinado a aplicaciones industriales con exigencia de eficiencia.

El autómata, que actualmente se encuentra funcionando prácticamente en un 100%, está formado por tres elementos: un manipulador (o "brazo"), un controlador y un conjunto de sensores. El controlador contiene el sistema procesador y las unidades de control de energía. El brazo es accionado por un servomecanismo y es capaz de moverse articuladamente en seis ejes. Los sensores utilizados, con la finalidad de mejorar el control del autómata y la seguridad en el entorno de trabajo, son cámaras de alta velocidad y acelerómetros.

El fundamento del presente trabajo consiste en el desarrollo del sistema de procesamiento digital de imágenes que da soporte al autómata en cuestión.

Introducción

La aplicación de los sistemas digitales del procesamiento de señales y su aplicación al control de motores de inducción y autómatas programables tiene su origen en la década de 1970, con los primeros desarrollos de procesadores digitales de señales DSP por Intel, AMI y Bell Labs. Las dificultades en cuanto a la capacidad de procesamiento y velocidad para realizar cálculos, hacen que su aplicación se limite a controles de baja complejidad. No obstante, el gran avance tecnológico producido desde ese momento, tanto en la miniaturización de los sistemas digitales, la capacidad de cálculo en los procesadores de la época y su velocidad de procesamiento hacen que se puedan aplicar a controles con algoritmos de alta complejidad.

El autómata propuesto es un brazo mecánico capaz de moverse articuladamente en seis ejes, del tipo de "articulación coordinada", llamado así por la semejanza de sus movimientos con los del cuerpo humano. La programación de los movimientos del autómata se realiza desde una computadora personal mediante un software a desarrollar.

Los autómatas programables, tal como se mencionó en el apartado anterior, se utilizan ampliamente en la industria automotriz. Pero existen numerosos sistemas de producción que no cuentan con autómatas que se adapten a sus necesidades, por lo que el trabajo que aquí se describe permite proponer soluciones innovadoras que cubran dichas necesidades.

En particular se hace indispensable el desarrollo de algoritmos para el guiado de autómatas en base a la visión, lo que permitirá utilizarlos para detectar y manipular objetos de diferentes tamaños, color y contraste, mejorar la precisión al realizar una trayectoria, y evitar objetos no previstos en una tarea, haciendo al autómata adaptable a diferentes condiciones de trabajo.

En conclusión, con el fin de mejorar la productividad de las industrias, la calidad y el costo de fabricación para ganar competitividad a nivel global se hace interesante contar con autómatas programables en los procesos de fabricación con sistemas de visión.

Marco teórico

El procesamiento digital de imágenes ha tomado un gran impulso en la última década, derivado de la alta capacidad de los procesadores que posibilitan el desarrollo de sistemas más versátiles y de costo reducido, ejemplificados en el control de procesos realizados por robots industriales, la navegación en vehículos autónomos, la detección de eventos e inspección de objetos en procesos de manufacturas y en un sin fin de aplicaciones más.

En la actualidad existe una gran cantidad de bibliografía sobre estudios y algoritmos desarrollados para el control de motores de inducción, sistemas de control electromagnéticos, vehículos eléctricos, procesamiento digital de imágenes y autómatas programables con una vasta aplicación en la industria. Sin embargo, existen numerosos sistemas de producción que no cuentan con

autómatas que se adapten a sus necesidades de proceso automatizados para un aumento del nivel de producción y calidad, menos aún que utilicen el procesamiento digital de imágenes ya sea por su elevado costo o por la inexistencia en el mercado. A partir de esta situación se decidió desarrollar un brazo robot que pueda ser automatizado y tomar decisiones en su movimiento, nutriéndose de la entrada que proporcionan cámaras de video.

Para que el brazo pueda tomar decisiones necesitará reconocer objetos, como se mencionó anteriormente, y esto implica la adecuada elección de un algoritmo que cumpla con este fin. Al momento de iniciar la investigación sobre estos algoritmos se comenzaron a realizar pruebas con SURF[1] y ORB[2] pero no se lograba una correcta detección en objetos que no tuvieran textura. Tampoco se deseaba utilizar redes neuronales convolucionales [3] debido al costo de procesamiento y de entrenamiento más allá de su buen desempeño en reconocer patrones y la atención [4], junto con las redes neuronales recurrentes no se encontraban en ese momento tan difundidas en el campo del procesamiento digital de imágenes.

De todas las opciones estudiadas la que más se acercaba a nuestros requerimientos fue el algoritmo DOT [5] que fue el implementado y el cual se detalla en los siguientes párrafos.

Implementación de DOT

Como parte del proyecto desarrollado se planteó el desarrollo e implementación del trabajo de Stefan Hinterstoisser y otros, en el que se presenta un método (DOT) para detectar objetos tridimensionales en tiempo real, bajo una cierta cantidad de ruido, que no requiere una etapa de entrenamiento demasiado costosa y que puede detectar objetos sin textura (problema que, sí se observó, de manera empírica, al utilizar SURF como se mencionó).

DOT

DOT es un algoritmo de detección de objetos presentado en la conferencia de visión por computadora y reconocimientos de patrones del año 2010. Este algoritmo tiene la capacidad de poder detectar los objetos de una manera muy rápida y repetitiva para lo cual se utilizan distintas imágenes del mismo objeto, a las que se denomina patrones (a partir de ahora se utilizará la terminología original “templates”) del objeto a buscar. Debido a la forma en que se generan estos “templates” y al método con el que se busca el objeto en la escena, este algoritmo es robusto en la detección de objetos sin texturas. DOT se puede resumir en tres grandes pasos:

- Creación de los “templates”.
- Preparación de la escena.
- Búsqueda del objeto.

A continuación, y rápidamente se realizará una breve descripción de cada uno de los pasos anteriormente mencionados

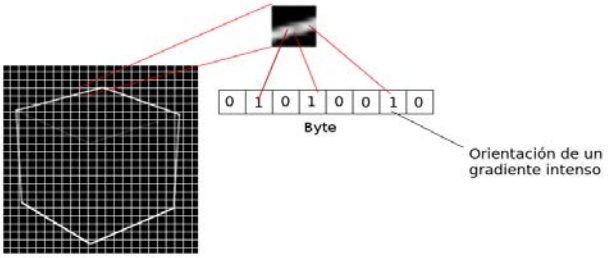

Figura 1: Ejemplo de como se realiza un template

Figura 2 : Templates del envase de un adhesivo vinílico

Creación de templates

La primera etapa es la creación de los “templates” a partir de los objetos a buscar. La metodología propuesta por DOT es simple: obtener imágenes del objeto desde distintas perspectivas. Mientras más imágenes distintas se obtengan para generar los “templates”, más eficiente será la búsqueda, pero también más lenta. Estas imágenes serán la materia prima de los “templates” y se buscarán posteriormente en la escena. Un ejemplo de cómo se realiza un template se puede observar en la figura 1.

Posteriormente, a cada “template” se le aplica el filtro Sobel [6] para así poder hallar los gradientes de la imagen. La imagen resultante se divide en cuadrados “pequeños” de N x N pixel y se recogerá la orientación de los 7 primeros gradientes que tengan más intensidad en cada cuadrícula. Estos datos se guardarán en un byte siendo el MSB el bit que indique si en esa celda de la cuadrícula hubo o no gradientes. Un ejemplo de templates se puede observar en la figura 2.

Preparación de la escena

Se denomina “escena” a la imagen en la que se buscará el objeto a detectar. El proceso es similar a como se prepara un “template”: los bordes de la escena se detectan con un filtro Sobel para así obtener los gradientes de esta, luego se divide la imagen en cuadrados de la misma medida utilizada en los “templates” con la diferencia de que no se buscan los 7 gradientes más intensos por cada uno de los cuadrados, sino que se guarda la orientación del gradiente más intenso. En el caso de trabajar con videos esto se tiene que hacer fotograma a fotograma y la manera en que se debe programar el algoritmo tiene que ser muy rápida para ofrecer una detección acorde a los tiempos del brazo robot.

Búsqueda

La etapa de búsqueda es conceptualmente muy sencilla. Se tienen en cuenta las orientaciones de los gradientes más fuertes de cada cuadrado en el "template" y la orientación del gradiente más fuerte de cada cuadrícula de la escena. Según la investigación DOT, se debe deslizar el "template" sobre la escena, calculando cuántas orientaciones del mismo coinciden con las orientaciones de la escena. En el lugar donde hubo mayores coincidencias es donde se supondría que se encuentra el objeto a detectar.

Primera implementación

En esta investigación el algoritmo de DOT fue implementado en más de una ocasión y en diversas formas buscando siempre una optimización en velocidad. La primera implementación de DOT fue realizada en una computadora con arquitectura x86 sin tener en cuenta ningún tipo de optimización. Si bien los resultados fueron alentadores, desde el punto de vista del comportamiento

Figura 3: Detección de un envase de adhesivo vinílico con una pequeña oclusión y una piedra de desgaste.

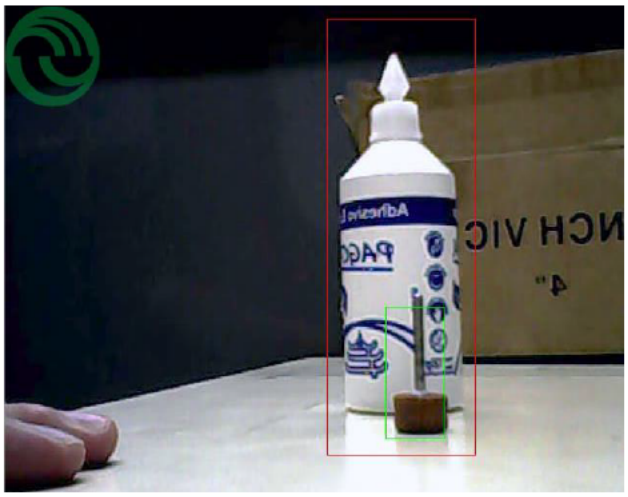

Figura 4: Dos elementos detectados mientras uno ocluye al otro

Figura 5: Detección en movimiento

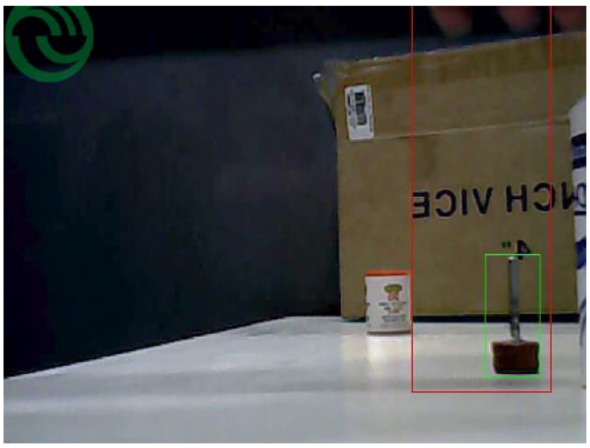

Figura 6: Error en la detección

del algoritmo, fue necesario programar el mismo nuevamente con mucho más cuidado.

En la segunda implementación, también programada para una arquitectura x86, se tuvo en cuenta la optimización del código y se trató de implementar, como recomiendan los autores de DOT [7], las instrucciones MMX. Con todos estos cambios, se obtuvo un aumento en la performance del algoritmo, pero la misma no fue la suficiente para poder utilizarlo en un brazo robot. Sin embargo, se pudo programar el algoritmo de una forma mucho más eficiente como se detallará a continuación.

En la figura 3 se puede observar la detección de un envase de adhesivo vinílico y una piedra de desgaste. En el caso del envase se puede detectar con cierto nivel de oclusión sin mayores inconvenientes. En el caso de la figura 4 se puede observar como el algoritmo puede detectar un objeto ocluyendo a otro, siendo ambos detectados correctamente.

En la figura 5 se puede observar cómo se detecta el envase de adhesivo vinílico en movimiento. En la figura 6 el algoritmo programado falla al no encontrar el objeto. Este es uno de los problemas a resolver.

Implementación de DOT en GPU

Con el objetivo de avanzar en el desarrollo de algoritmos para la detección y procesamiento de imágenes, se procedió a la incorporación al sistema de una placa de desarrollo Nvidia Jetson TK1. La placa en cuestión está construida sobre la base de un procesador ARM® Cortex™-A15 y una unidad destinada al procesamiento gráfico de NVIDIA Kepler GPU con 192 núcleos CUDA. Cabe mencionar que CUDA es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento del rendimiento del sistema.

En un primer intento se compiló el código realizado anteriormente con mínimas modificaciones para ejecutarlo en la placa de desarrollo. Las consideraciones iniciales, basadas en que gran parte del código hace uso de la biblioteca OpenCV, [8] y en que varios métodos utilizados están optimizados para la ejecución en una GPU, prometían o sugerían un buen rendimiento. Pero esto no fue así. La detección en un video de prueba superaba por mucho el tiempo de procesamiento al de su ejecución mediante el análisis en una CPU.

Visto y considerando este inconveniente se comenzó a analizar el código con detenimiento. Se observó que la manera en la que se estaba intentando utilizar la GPU no era la correcta, por lo que se examinó cada uno de los métodos utilizados para identificar aquellos que producían la mayor demora y optimizarlos. A partir de este análisis se llegó a la conclusión de que el cálculo de gradientes era el primero en que más demoraba en ejecutarse. Este método se debe ejecutar fotograma a fotograma.

En el cálculo de gradientes las tareas a realizar son las siguientes y cada una de ellas depende de la anterior:

1. Se convierte a escala de grises la imagen del fotograma a analizar.
2. Se calcula el filtro Sobel por X e Y.
3. Se calcula el gradiente a partir de los filtros anteriores.
4. Se halla el máximo gradiente en cuadrados de NxN pixeles en toda la imagen.
5. Se calcula el ángulo de dicho gradiente guardando su orientación en un byte.
6. Se almacena el resultado.

De toda la lista anteriormente mencionada, el primer lugar en donde se atacó la demora fue en el cálculo del filtro Sobel. Se utilizó la función optimizada para CUDA de OpenCV de manera correcta y debido a que se tienen que hallar los componentes en X y en Y, estas tareas se ejecutaron en paralelo. Sin embargo, si bien se obtuvo una mejora sustancial, la comprensión y el análisis de los tiempos globales de demora llevó a los responsables de la investigación a la conclusión de que la mejor manera de implementar el algoritmo DOT no era utilizando los métodos optimizados ofrecidos por OpenCV, sino a través del desarrollo de un kernel. Esto fue producto de dos circunstancias, en un principio, desconocidas al momento de iniciar las pruebas:

- El tiempo de transmisión de datos desde la memoria principal a la memoria de la GPU (y viceversa) no es despreciable y de manera inherente genera una sobrecarga en la ejecución del algoritmo (“*overhead”)*. Por lo tanto, utilizar una GPU sin tener en cuenta lo anterior produce demoras y las mismas deben ser minimizadas.
- Se debe buscar en la imagen máximos locales en regiones de interés de NxN pixeles. En primer lugar, OpenCV no ofrece una función optimizada que utilice CUDA para realizar esta tarea y en el caso de ofrecerla se debería tener en cuenta el tiempo de transferencia entre la memoria principal y la memoria de la GPU.

Por todo esto se procedió al diseño de un kernel que pudiera acelerar y hacer uso correcto de la GPU y así hallar los gradientes de manera rápida y correcta. El enfoque utilizado para resolver el problema fue dividir y solucionar el mismo en etapas parciales.

Lo primero que se realizó fue el desarrollo de un kernel que permitiera aplicar el filtro Sobel a una imagen utilizando la GPU. Este paso fue fundamental ya que en un kernel se solucionaban los puntos 2 y 3 de la lista de tareas mencionada anteriormente.

Seguidamente se ensayó el comportamiento del kernel sobre un video y los resultados fueron muy alentadores, en un tiempo mucho menor de lo esperado la aplicación del filtro era calculado. Además, se comprobó también que la placa aceptara el uso de una webcam la cual se utilizó con el algoritmo y funcionó de manera correcta. Por lo tanto, la lista de tareas que hasta ahora se resume es la siguiente:

1. Se convierte a escala de grises la imagen del fotograma a analizar.
2. Se calculan el filtro Sobel por X e Y y el gradiente (GPU).
3. Se halla el máximo gradiente en cuadrados de NxN pixeles en toda la imagen.
4. Se calcula el ángulo de dicho gradiente guardando su orientación en un byte
5. Se almacena el resultado.

El paso siguiente consistió en la búsqueda del máximo gradiente en cuadrados de NxN pixeles en la GPU.

La primera aproximación para solucionar este problema fue programar en un kernel distinto la búsqueda de máximos locales utilizando la GPU en bloques de NxN pixeles. De esta manera se trabajó la imagen resultante del filtro anterior en cuadrados NxN, pudiendose hallar paralelamente los máximos de cada bloque. Esto fue correcto pero el hecho de utilizar un único hilo para buscar el máximo gradiente desperdiciaba el poder de procesamiento de la GPU. Es por esto que después de investigar y analizar el problema se reprogramó la

solución utilizando el concepto de reducciones paralelas, en este caso, en matrices.

La idea de las reducciones paralelas es sencilla: Para hallar un máximo se deben utilizar comparaciones. Si se utiliza un solo núcleo o un solo hilo, ese único hilo o núcleo debe realizar todas las comparaciones. Pero al tener una GPU y poder decidir cuantos hilos se ejecutan por cada bloque, esta situación cambia. Como la operación de comparación es binaria, la máxima cantidad de hilos que pueden utilizarse para solucionar el problema de manera óptima, es la mitad de la cantidad de elementos (si ésta es par).

Por ejemplo, si se trata de hallar el máximo en una matriz de 8x8 elementos existirán 64 elementos a comparar y si se utilizan 32 hilos se podrán comparar 32 pares de elementos de una sola vez, guardando el resultado en una de las mitades. Si esta operación se continúa realizando (con los elementos que sobran comparar) la eficiencia que se obtiene es realmente considerable.

La figura 7 ejemplifica de una manera clara el concepto de reducción que se programó.

A posteriori de haberse hallado el máximo, en el mismo kernel se programó el cálculo para transformar y expresar su orientación medida en un byte, tal como lo proponen los autores del algoritmo DOT. Por lo tanto, la lista de tareas que se resume hasta ahora es las siguiente:

1. Se convierte a escala de grises la imagen del fotograma a analizar.
2. Se calculan el filtro Sobel por X e Y, y el gradiente (GPU)
3. Se halla el máximo gradiente en cuadrados de NxN pixeles en toda la imagen, se calcula el ángulo de dicho gradiente guardando su orientación en un byte y se almacena el resultado (GPU).

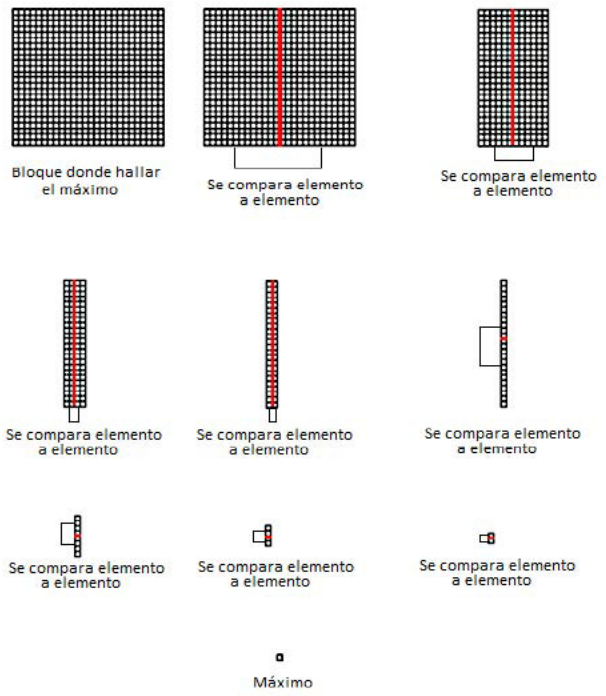

Figura 7: Ejemplificación de la reducción de una matriz utilizando una GPU

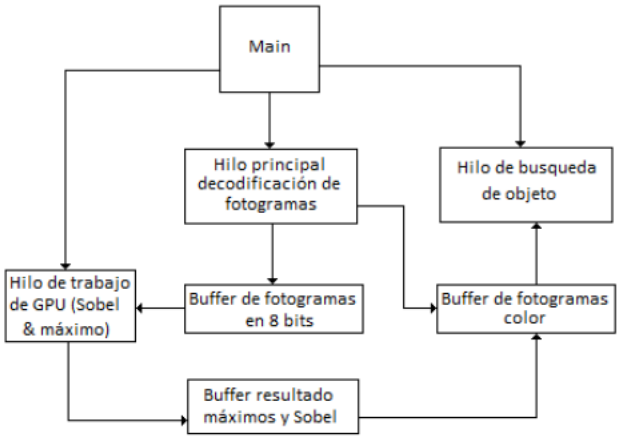

Figura 8: Esquema de ejecución del algoritmo

Por último, debido a que existen demoras entre la ejecución de ambos kernels, estos se combinaron en uno solo. Todas estas mejoras dejan así un aumento de velocidad considerable respecto a las mismas tareas ejecutadas en una CPU o utilizando en parte la GPU, a través de OpenCV.

Ya solucionado el problema generado por la demora del método que calcula los gradientes, se siguió analizando el programa en búsqueda de otras demoras. Un punto clave que producía las mismas, fue la decodificación del método de compresión utilizado en los videos o en la misma cámara de la que se obtienen los fotogramas. Debido a que la única manera de aumentar la velocidad de esta decodificación es utilizar la GPU, y la transferencia entre la memoria principal y la de la GPU produce demoras, se decidió decodificar los fotogramas en un hilo.

Se debe tener en cuenta que la placa Nvidia Jetson TK1 posee un microcontrolador ARM de cuatro núcleos y utilizar uno en particular para decodificar los fotogramas no produce ningún tipo de problemas.

Valiéndose de una arquitectura de productor consumidor, en el que en un hilo se decodifican los fotogramas (productor) y en otro hilo se ejecuta el kernel donde se buscan los gradientes y los máximos (consumidor), se aumentó aún más la eficiencia de ejecución del algoritmo DOT. El esquema de la figura 8 muestra cómo se comunican y ejecutan los diversos algoritmos.

Solamente resta optimizar la búsqueda del objeto con la GPU denominado “Match Template”.

Optimización de la búsqueda

En primer lugar, se entiende que no se puede sobrecargar más a la GPU en este punto, por lo tanto, la búsqueda del objeto se debe realizar en la CPU. El algoritmo DOT plantea que lo que se debe realizar para encontrar un objeto es buscar la coincidencia más probable entre la escena y los objetos que se aprendieron. Para poder hacer esto se necesita utilizar una ventana deslizante (que recorre toda la escena) y buscar coincidencias de ángulos de los objetos aprendidos. El ganador o la región candidata a ser el objeto en cuestión es aquella en la que existan más coincidencias de orientaciones entre la

plantilla del objeto aprendido y la escena. En una versión simplificada del código se vería así:

```
Por cada plantilla de objeto aprendido
    Por cada pixel en Y de la escena
      Por cada pixel en X de la escena
            Por cada pixel en Y de la plantilla
             Por cada pixel en X de la plantilla
Match+=Escena(x,y)&plantilla(x,y)
```

Es evidente que la complejidad computacional es alta y de alguna manera habría que minimizarla. Después de analizar la forma en que funcionaba el algoritmo, se llegó a la siguiente conclusión:

Se podría aumentar la eficiencia del algoritmo si solamente se comprobara la coincidencia de ángulos en los lugares donde en la plantilla estén definidos, de esa manera se evitaría una iteración "*for*" y, por ende, varias operaciones en la búsqueda. Además se notó, experimentalmente, que es mucho más rápido hacer la búsqueda cada dos pixeles en la escena que uno por uno (en este caso se estaría bajando la resolución de búsqueda).

Para lograr esto se modificó el método que utiliza el programa de aprendizaje para analizar y guardar los objetos aprendidos, ahora en vez de guardar toda la plantilla se guarda solamente el valor del/los ángulos y la posición en la misma. En resumen, una versión simplificada del código se vería así

```
Por cada plantilla del objeto aprendido
    Por cada pixel par en Y de la escena
        Por cada pixel par en X de la escena
            Por cada ángulo valido en la plantilla
                If(Escena(x,y) es un ángulo definido)
                  Match+=Escena(x,y)&plantilla (x,y)
```

Si bien el cambio no es drástico, es evidente que existen mejoras.

Cambios drásticos en los objetos.

El algoritmo DOT no es perfecto y en las pruebas se observó más de una vez que quizás por falta de plantillas de objetos la detección no se realizaba en el lugar correcto. Para morigerar este comportamiento se puede suavizar de alguna forma la detección brusca de objetos, teniendo en cuenta que los mismos no pueden aparecen en otro lugar en la escena de una forma inmediata, sino que necesitan trasladarse por la misma. Para solucionar este problema se implementó la siguiente solución: Si se detectó un objeto en una posición que supera umbral de detección, ya habiéndose detectado el mismo, la ventana de detección se trasladará de manera progresiva al lugar donde se supone que se encuentra el objeto.

Pruebas en ambiente real

A continuación, se presentan algunas imágenes de momentos en los que el algoritmo produjo resultados apropiados, en videos que se tomaron en ambiente real con ruido de fondo. Es importante recalcar que fueron algunos momentos debido a que no se utilizaron demasiadas plantillas para realizar la detección. Las imágenes de las figuras 9 a 16 son grises debido a que se consume demasiada memoria si se utilizan los tres canales de colores. El exceso de consumo de memoria genera, como consecuencia, que el sistema operativo termine de manera abrupta el proceso de detección.

Fig. 9. Algoritmo reconociendo el envase de pegamento

Fig. 10. Algoritmo presentando una falla en la detección

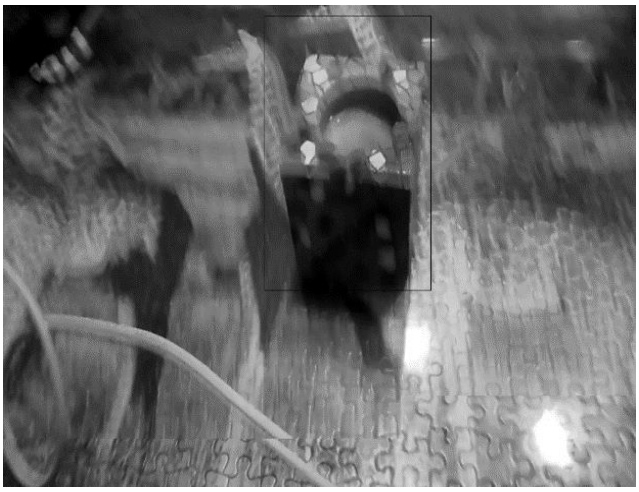

Fig. 11. Algoritmo detectando un cubo correctamente

Fig. 12. Algoritmo fallando por falta de plantillas. En este video en particular la tasa de fallas fue muy alta. El algoritmo malinterpretaba el ruido de fondo (rompecabezas)

Fig. 13. Tomando un video donde se reconocía el cubo en la implementación en CPU se trató de reconocerlo. En rojo la detección de esta implementación y en azul la de la anterior. Cabe destacar que se utilizaron diferentes plantillas

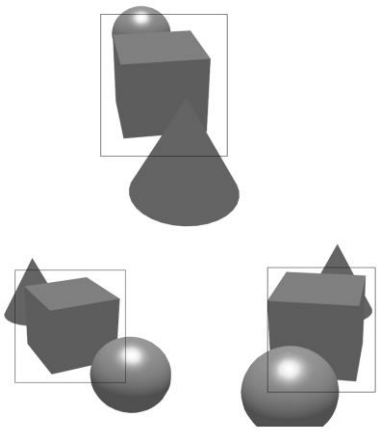

Fig. 14. Detecciones correctas en el video de prueba que se utilizó. Observar la detección con oclusiones

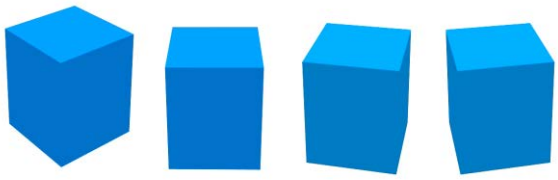

Fig. 15. Plantillas que se utilizaron para la detección de la imagen anterior. El video de prueba no tuvo falsos positivos.

Detección de profundidad

La detección de un objeto a partir de la implementación de DOT permite encontrar el mismo en el plano, pero no en el espacio, siendo parte fundamental la detección de esta dimensión debido a la naturaleza posicional del brazo robot. Es por esto que se necesita realizar la medición de la profundidad por medio de un sistema estereoscópico. La posibilidad que se evaluó es la de ubicar dos cámaras en un mismo plano M (con coordenadas x e y) separadas por una distancia b en el eje x, y con sus ejes focales paralelos y perpendiculares al plano M. De esta forma se lograrían dos imágenes dispares a nivel horizontal, y el producto de esa disparidad se utilizaría para medir la profundidad.

A nivel matemático [9], este mecanismo puede ser resuelto a través del método de triangulación. En la figura 16 se puede observar un esquema básico, en el cual se muestran dos sistemas de ejes cartesianos cada uno correspondiente a cada cámara del par estereoscópico.

A una profundidad dada por la distancia focal de las cámaras f (que debe ser la misma en ambas) se encuentra el plano denominado epipolar que contiene los planos I_I e I_D correspondientes a las imágenes tomadas por cada cámara. En este plano es donde se miden las disparidades entre dichas imágenes.

Al encontrarse las cámaras a la misma altura (eje y), las únicas diferencias se dan a nivel horizontal ya que, como se comentó, las cámaras se ubican a una distancia de separación b sobre el eje x.

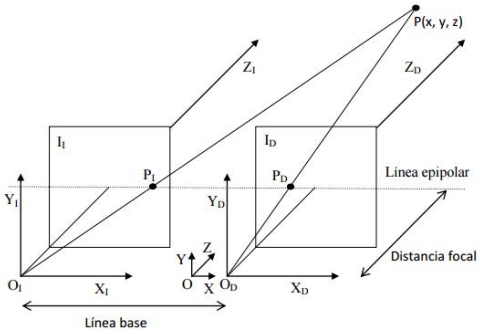

Figura 16 : Diagrama tridimensional para el cálculo de la distancia del punto P

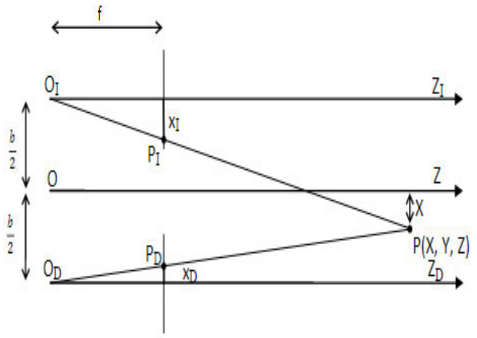

Figura 17: Diagrama bidimensional para el cálculo de la distancia del punto P

En la figura 17, se pueden observar los puntos PI y PD que son las proyecciones del punto P sobre la línea epipolar. Se puede intuir que cuanto más lejano se encuentre el punto P, los vectores de proyección tenderán a ser paralelos al eje Z. Lo que hará PI y PD tiendan a ubicarse sobre el eje ZI y ZD.

La disparidad del punto se calcula como XI - XD = d y en el caso de que un objeto se encuentre ubicado en el infinito del eje Z hará que d valga 0. Así entonces el cálculo de la distancia del objeto sobre eje Z será

$$ImagenIzquierda: \frac{\frac{b}{2}+x}{z} = \frac{x_I}{f} \Rightarrow x_I = \frac{\left(x+\frac{b}{2}\right)f}{z}$$

$$ImagenDerecha: \frac{\frac{b}{2}-x}{z} = \frac{x_D}{f} \Rightarrow x_D = \frac{\left(x-\frac{b}{2}\right)f}{z}$$

En la ecuación *Imagen Derecha* se asigna un signo negativo a XD para compatibilizar los sistemas de coordenadas de los ejes OI y OD. Siendo la disparidad del punto P: x_I-x_D= d reemplazando y despejando se llega a la ecuación:

$$z = \frac{fb}{d}$$

Con la cual se puede calcular la distancia z del punto P (objeto detectado)

Por otra parte, el desarrollo de los algoritmos de control del autómata se apoyó en las bibliotecas desarrolladas por Peter Corke, realizándose simulación mediante el software MATLAB.

Queda pendiente la integración con las bibliotecas que se desarrollaron para la detección de objetos, debido a que se priorizó el desarrollo de las bibliotecas de procesamiento gráfico.

Calibración de las cámaras

Lo ideal es que, en un principio, el ajuste fuese realizado a grandes rasgos lo más precisamente posible. para que luego el algoritmo de calibración tuviese mejores resultados.

Una vez hecho este ajuste, se comenzó con el cálculo de las matrices de calibración, para lo que se utilizó la biblioteca de OpenCV que contiene clases para calibración y reconstrucción de imágenes 3D: “Camera calibration and 3D reconstruction”. Para la calibración se usó la aplicación *calibrate_cameras*, que forma parte de la biblioteca StereoVision (Bajo licencia pública GNU)

Esta aplicación toma como argumentos imágenes estéreo que contienen en su escena un patrón cuadriculado símil a un tablero de ajedrez, con capturas en distintas perspectivas de este (figura 18). El tablero se rota y además se lo traslada a lo largo de las múltiples tomas de imágenes. Matemáticamente se puede demostrar que la cantidad mínima de tomas depende de la cantidad de esquinas que contiene el tablero patrón. El término “esquinas” hace referencia a la cantidad de vértices de cuadrados negros que hacen contacto entre sí (léase negros o blancos). La máxima cantidad utilizada es 50, bajo el criterio de cuantas más muestras haya los cálculos de corrección de errores serán más precisos.

La aplicación *Calibrate_cameras* da como resultado una carpeta con un conjunto de matrices que contiene los parámetros intrínsecos (focos, centrados), extrínsecos (traslación y rotación) y las matrices de rectificación de distorsiones. Con estas matrices se realizan las correcciones sobre las imágenes tomadas en cada cámara. Una vez rectificadas las imágenes se puede realizar el cálculo del eje Z con la fórmula citada más arriba.

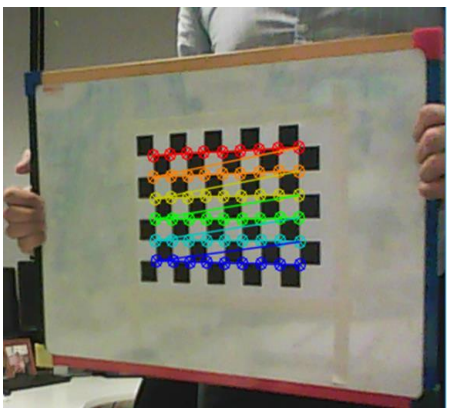

Figura 18: Vértices detectados por el algoritmo de calibración “findChessboardCorners” de la biblioteca OpenCV

Para lograr este cálculo hay que ubicar en ambas imágenes del par estéreo el mismo objeto, para esto es necesario un algoritmo de búsqueda e identificación de objetos. Esto se puede resolver con el algoritmo DOT que fue detallado anteriormente.

Verificación con mapa de disparidad

La verificación con mapa de disparidad se realizó utilizando las bibliotecas de *StereoVision*, una prueba empírica realizando un mapa de disparidad y una nube de puntos. La nube de puntos consiste en conseguir las coordenadas espaciales de cada pixel de la escena tomadas por las cámaras. De esta forma se puede visualizar una escena desde diferentes perspectivas rotando los ejes de coordenadas. Esto tiene un límite y es que esos "puntos" de la imagen se obtienen desde una posición fija del espacio, si se deseara observar perspectivas laterales se necesitaría más información de la escena. Esto se puede salvar hasta cierto ángulo, replicando los píxeles de la frontera, de forma que cubran los huecos de información faltante.

Para lograr un buen mapa de disparidad se necesita que los objetos de la escena sean detectados inequívocamente en las imágenes del par estéreo. Se utilizó la herramienta *tune_blockmatcher* de la biblioteca *StereoVision* con el fin de ajustar los parámetros del detector de objetos y se verificó en paralelo la "calidad" del mapa de disparidad que se muestra en la figura 19.

Los ajustes mencionados se deben realizar en forma iterativa y empírica ya que resulta ser el método más rápido y eficiente. Luego de realizar el ajuste del algoritmo de búsqueda y detección de objetos, se utilizó la función *images_to_pointcloud* (también de la citada biblioteca StereoVision), la que toma como argumentos las matrices de rectificación, un par de imágenes estéreo, los parámetros de ajuste del algoritmo de "*blockmatching*" y esto da como resultado una matriz de tres dimensiones que contiene los vectores (también de tres dimensiones) correspondientes a la información del color de cada pixel. Como resultado se puede observar la siguiente nube de puntos Figuras 20 y 21)

Conclusión

Quizás las modificaciones y los agregados realizados sobre el algoritmo DOT no sean suficientes para obtener velocidades considerables en la detección, aunque permiten mejorar adecuadamente la misma. Ahora es aceptable realizar una búsqueda en una imagen HD, pero aun así sigue siendo relativamente lento. Como se plantea al final del apartado "Implementación DOT en GPU" quizás la respuesta involucre la implementación del proceso de "match template" en la GPU, aprovechando el tiempo ocioso de la misma.

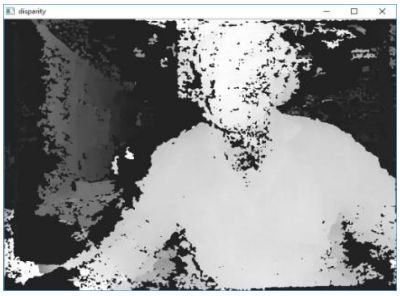

Figura 19: Mapa de disparidad

Figura 20: Un par de imágenes (imagen estéreo)

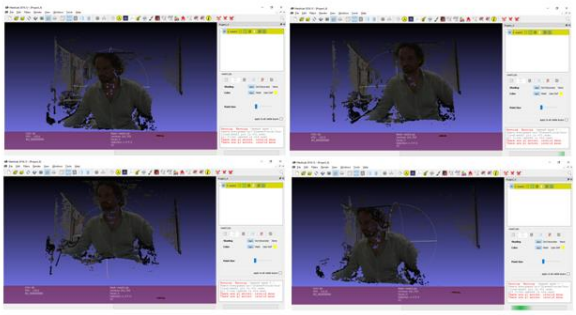

Figura 21: resultado de la nube de puntos, a partir de la imagen estéreo.

Respecto al algoritmo, presenta fallas que quizás sean mejorables. DOT no considera colores, solo considera formas, y queda claro que resolver esa falencia sería una mejora a esta implementación. Otro punto importante a destacar es que el tiempo de reconocimiento de un objeto depende de la cantidad de plantillas y de la dimensión del video. Esto no debería ser así, aunque es una característica intrínseca del algoritmo.

Una solución que se podría proponer para la mejora del algoritmo es la de no utilizar la ventana deslizante, sino que por algún medio se detectaran los posibles candidatos a la imagen de la plantilla buscada y luego realizar el "match template" entre las plantillas y esa región de la imagen.

Respecto al cálculo y análisis de profundidad, la calibración de las cámaras y la comprensión del algoritmo están en un nivel muy avanzado, al punto de que solamente sería necesario programar la detección y agregarla a la de objetos. El único inconveniente será el tiempo de procesamiento que se sumaría al que posee la detección DOT y, como se expresó, se debería buscar una solución de compromiso o analizar la unión de ambos algoritmos mucho más detenidamente.

En etapas siguientes de este proyecto de investigación, que sigue vigente en el ámbito de la Universidad, se procederá a continuar el trabajo sobe aquellas cuestiones que hoy se mencionan como dificultades o asuntos pendientes.

Referencias

[1] Bay, Herbert,Tuytelaars, Tinne, Van Gool, Luc.: SURF:Speeded Up Robust Features, European Conference on Computer Vision 2006

[2] Rublee,Rabaud,Konolige,Bradski: ORB: an efficient alternative to SIFT or SURF, IEEE International Conference on Computer Vision 2011

[3] LeCun,Haffner,Bottou,Bengio, Object Recognition with Gradient-Based Learning, CPVR 2004

[4] Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez , Kaiser, Polosukhin, Attention is all you need, Cornell University 2017

[5] Hinterstoisser Stefan, Lepetit Vincent, Ilic Slobodan, Fua Pascal, Navab Nassir.: Dominant Orientation Templates for Real-Time Detection of Texture-Less Objects, Computer Vision Pattern Recognition 2010.

[6] Gonzales, Woods: Digital Image Processing , Tercera Edición, Prentice Hall 2007

[7] Hinterstoisser Stefan, Cagniart Cedric, Ilic Slobodan, Sturm Peter, Navab Nassir, Fua Pascal, Lepetit Vincent.: Gradient Response Maps for Real-Time Detection of Texture-Less Objects, IEEE Transactions on pattern analysis and machine intelligence 2011

[8] OpenCV http://opencv.org/

[9] Martin Montalvo, Técnicas de visión estereoscópica para determinar la estructura tridimensional de la escena, Autor: Curso académico 2009/2010, Máster en Investigación en Informática, Facultad de Informática, Universidad Complutense de Madrid 2009/2010.