Universidad Zaragoza

1542

Trabajo Fin de Grado

Localización de múltiples robots móviles mediante una cámara cenital

Multiple mobile robot location based on cenital camera

Autor

Jorge Barrio Arbex

Director

José María Martínez Montiel

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2016

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

D./Dª. Jorge Barrio Arbex,

con nº de DNI 73003396Q en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Localización de múltiples robots móviles mediante una cámara cenital,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 14 de Septiembre de 2016

Fdo: Jorge Barrio Arbex

Localización de múltiples robots móviles mediante una cámara cenital

RESUMEN

La navegación de robots móviles es el principal problema que se plantea a la hora de manejar un robot. La navegación es la capacidad que tiene el robot de llegar a su punto de destino evitando cualquier tipo de obstáculo.

Para que esta operación se lleve a cabo de forma satisfactoria es necesario saber la localización del robot en todo momento y reconocer el terreno por el que se desplaza a tiempo real.

En este trabajo, utilizando el programa de visión artificial OpenCV y las librerías de realidad aumentada ArUco, mediante una cámara cenital, se reconoce la ubicación del los robots, de los límites de la pista por la que se mueven los robots y de los obstáculos que hay en el terreno, los dos primeros mediante unos marcadores fiduciales obtenidos de las librerías ArUco y los obstáculos mediante colores.

Para que esto sea posible, hay que realizar unos trabajos previos de calibración de la cámara que se va a utilizar y creación de los marcadores fiduciales que serán empleados, estos pasos solo habrá que realizarlos una vez.

Para conocer el color de los obstáculos, se colocan unos marcadores del color deseado en los extremos de la pista, por lo que, aun que cambie la luminosidad de la pista (dentro de unos valores aceptables), los obstáculos se seguirán reconociendo.

Las coordenadas obtenidas en la imagen de los obstáculos, los robots y la pista serán transformadas, cada una con su matriz de homografía, a frecuencia de video, en coordenadas de la pista y viceversa, lo que permite colocar la cámara en diferentes posiciones ya que el programa es capaz de auto calibrarse.

El programa mostrará por pantalla la pista con los obstáculos detectados, y la posición de los robots (x,y,theta) y almacenará como variables las coordenadas de las esquinas de los obstáculos detectados y las coordenadas de la ubicación de los robots y su ángulo de orientación.

La finalidad de este trabajo es conocer en todo momento los límites de la pista, y la localización de los robots y los obstáculos para implementar los datos posteriormente en una Toolbox que generará las trayectorias de movimiento de los robots.

Tabla de contenidos

1. INTRODUCCIÓN

1.1 LA NAVEGACIÓN EN LOS ROBOTS MÓVILES

Nada más aparecer los primeros robots móviles (Figura 1), la robótica tuvo que enfrentarse a uno de los grandes problemas que, a día de hoy, sigue siendo la principal preocupación a la hora de diseñar un robot, la navegación.

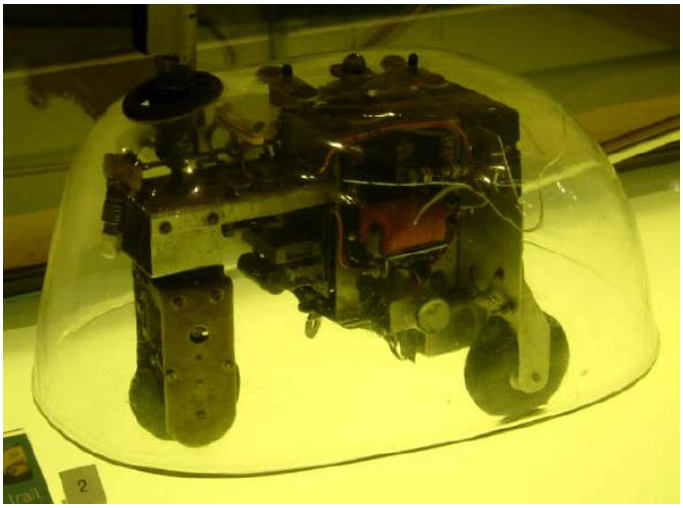


Figura 1: Tortuga de Bristol (http://wiki.robotica.webs.upv.es/)

La navegación es la capacidad que tiene un robot para llegar, desde el punto de origen, al punto de destino, evitando cualquier tipo de obstáculo. Para realizar esta tarea, existen diversos métodos pero todos ellos tienen en común una serie de subtareas. El robot tiene que ser capaz de percibir el entorno que le rodea e interpretar la información obtenida de los sensores, conocer en todo momento su posición en el entorno, planificar una trayectoria para llegar al punto de destino evitando cualquier obstáculo y ser capaz de realizar dicha trayectoria (Ortiz).

Este trabajo se centrará en el análisis de las dos primeras subtareas: interpretar la información del entorno que rodea al robot, obtenida por los sensores, y conocer la posición del robot en todo momento.

1.2 OBJETIVOS Y ALCANCE

El objetivo del proyecto es desarrollar un sistema de localización y detección de obstáculos para un equipo de robots móviles operando sobre una pista plana. La localización va a ser empleada para la navegación autónoma de los robots.

El sistema de localización se basa en una cámara cenital que observa la pista completa. La cámara identifica y localiza los diferentes robots y los obstáculos. Se desarrolla el software para la localización de múltiples robots identificados con un marcador fiducial y de los obstáculos de la pista definidos por formas planas de un color predefinido.

La principal característica de este sistema es la "recalibración a frecuencia de video". La pista se modifica para incluir un patrón de calibración tanto para la geometría de la escena como el color que define los obstáculos. De esta manera el único requisito para el funcionamiento del sistema es que la cámara vea la pista. Sin necesidad de un costoso proceso de calibración previo, ni de una iluminación controlada, ni de un sistema de posicionamiento de la cámara preciso.

1.3 METODOLOGIA

La implementación del software se hará con el programa de realizad virtual en OpenCV /C++, mientras que la calibración, creación de marcadores fiduaciales y la localización de estos se hará con la ayuda de la librería de realidad aumentada ArUco. La biblioteca resultante se hará pública como GPL.

En primer lugar se calibrará una única vez la cámara que vaya a ser utilizada. También se crearán los marcadores fiduciales que se vayan a utilizar, se diseñara la plataforma multi-robot con estos marcadores, y unos patrones de color y se colocará la cámara en posición cenital, enfocando en todo momento a la pista.

Posteriormente se diseñará un software que realizará instantáneas a frecuencia de video y procesará las imágenes obtenidas, creando dos matrices de homografía, una para los obstáculos y otra para los robots. Estas matrices se utilizarán para pasar de coordenadas mundo a coordenadas pixel y viceversa. Los robots serán localizados mediante unos marcadores que se les acoplarán, y los obstáculos serán reconocidos mediante colores.

Por último se verificará experimentalmente el correcto funcionamiento del software.

1.4 ESTRUCTURA

La estructura empleada en la realización de este trabajo es la siguiente:

Elección de los programas utilizados: se detalla una breve descripción de la utilidad de los programas empleados y sus características y se da el motivo de porque se ha elegido utilizar esta serie de programas en la realización de este TFG.

Calibración de la cámara: se describen las diferentes opciones a la hora de calibrar la cámara que se utilizará, los posibles patrones que se pueden utilizar para la calibración y el método

elegido. También se detalla la forma correcta de calibrar una cámara y los programas utilizados para ello.

Diseño de la plataforma multi-robot: en esta sección se explica cómo crear la pista por la que se desplazarán los robots y las características que tiene que contener para su correcto funcionamiento.

Localización de múltiples robots móviles y detección de obstáculos: este es el apartado principal del trabajo, en él se describe el funcionamiento del software creado y los pasos que realiza desde que se ejecuta el programa hasta que se detectan los robots y los obstáculos. También se analizan los resultados obtenidos.

Anexos: los anexos contendrán un manual de instalación de los programas y las librerías utilizadas y una descripción tanto del modo de empleo como del funcionamiento de los programas utilizados.

2. ELECCIÓN DE LOS PROGRAMAS UTILIZADOS

Existen numerosos trabajos centrados en la localización de múltiples robots móviles mediante una cámara, desarrollados en diferentes entornos de desarrollo, siendo MATLAB el más común por su facilidad a posteriori en el cálculo técnico para la generación de trayectorias.

Para la realización de este TFG, la primera opción era usar MATLAB, pero se tuvo que descartar debido a que se requerían una serie de Toolbox de las cuales la universidad no disponía, y el desembolso económico por su adquisición era demasiado elevado, por lo que se optó por **OpenCV**, "una biblioteca de visión artificial gratuita y con licencia BSD, que permite que sea usada libremente para propósitos comerciales y de investigación con las condiciones en ella expresadas" ("Opencv" 2016).

Como entorno de desarrollo se ha elegido **Eclipse**, "una plataforma de software compuesto por un conjunto de herramientas de programación de código abierto multiplataforma. Esta plataforma, típicamente ha sido usada para desarrollar entornos de desarrollo integrados" ("Eclipse (Software)" 2016). Se eligió Eclipse porque ya se estaba usando en los ordenadores de la universidad.

A la hora de elegir la librería de realidad aumentada que se utilizará en la creación y detección de marcadores, se tuvieron en cuenta las dos opciones más utilizadas en el mercado, ARToolKit y ArUco. La primera es una biblioteca desarrollada por Hirokazu Kato en 1999, mientras que la segunda ha sido desarrollada por 2 profesores de la Universidad de Córdoba, Rafael Muñoz Salinas y Sergio Garrido-Jurado en 2014. En la tabla 1 se comparan las principales características de ambas librerías que resultarán útiles para la realización de este trabajo:

	Licencia	Detención de marcadores	Descargas último mes	Descargas último año	Descargas últimos dos años
ArUco	BSD	SI	2.582	19.379	31.219
ARToolKit	GNU GPL	SI	1.907	23.948	57.598

Tabla 1: Comparación ente la librería Aruco y la librería ARToolKit (Fuente: sourceforge.net)

Se observa como ARToolKit está consolidada en el mercado, fue la primera librería de realidad aumentada y por lo tanto es la más utilizada, pero ArUco, aún siendo relativamente nueva, ha tenido gran aceptación entre los usuarios y el número de descargas ha ido aumentando exponencialmente hasta asentarse en el mercado al mismo nivel que ARToolKit.

Otra característica importante, es el tipo de licencia que tiene cada biblioteca, en el caso de este proyecto, interesa una licencia BSD para poder copiar, modificar, crear y distribuir código libremente.

La librería ArUco se encuentra en constate desarrollo y evolución, cada cierto tiempo van saliendo versiones mejoradas y los creadores están a disposición para compartir y solucionar los problemas encontrados al usar sus librerías. En este trabajo hubo problemas con la instalación de las librerías ArUco y los creadores siempre se mostraron receptivos para

intentar solucionar todos los problemas existentes y cuando estos estuviesen subsanados, sacar una nueva versión de la librería con los problemas corregidos.

Por todos estos motivos, se decidió utilizar la librería ArUco debido a que todas las características importantes para este trabajo de la librería ArUco son notablemente mejores que las de la librería ARToolKit, a excepción de que, al ser una librería relativamente nueva, la información de la que se dispone y que se puede encontrar es inferior a la otra librería.

3. CALIBRACIÓN DE LA CÁMARA

Antes de empezar a utilizar el programa, siempre y cuando no se utilice una cámara proyectiva, hay que calibrar la cámara. Esta calibración solo se tiene que realizar una vez para cada cámara, puesto que, al realizarla, se crean unas matrices, que se utilizarán para corregir la distorsión y que solo dependen de las características de cada cámara, por lo que sus valores no cambiarán.

La calibración de la cámara se puede hacer empleado cualquiera de las dos librerías utilizadas para este trabajo, ya sea utilizando el código proporcionado en la documentación de la página de OpenCV (opencv.org) o empleando unos programas ubicados en la carpeta utils de la librería ArUco.

La calibración se puede realizar de dos formas distintas, con un video (en vivo o grabado con anterioridad) en el cual se muestra un tablero con una serie de patrones, que se moverá como se explica posteriormente, y el programa va captando instantáneas cuando localiza el patrón, o mediante la lectura del programa de una serie de imágenes tomadas con anterioridad en las que aparece el tablero en diferentes posiciones. Cuando el programa localiza el tablero, lee y almacena una serie de valores con los que posteriormente calculará los parámetros intrínsecos y extrínsecos de la cámara.

La calibración con OpenCV se realiza utilizando como patrón de calibración un tablero de ajedrez o un tablero con patrones circulares.

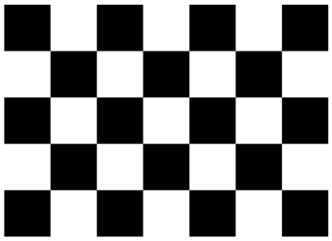


Figura 2: Tablero de ajedrez (mecatronicauaslp.wordpress.com)

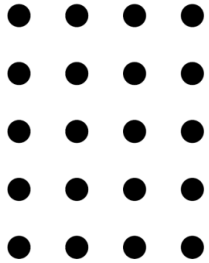


Figura 3: Tablero circular simétrico (mecatronicauaslp.wordpress.com)

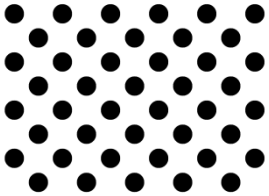


Figura 4: Tablero circular asimétrico (answers.opencv.org)

Para la calibración de la cámara mediante la librería ArUco se necesita crear tableros como estos:

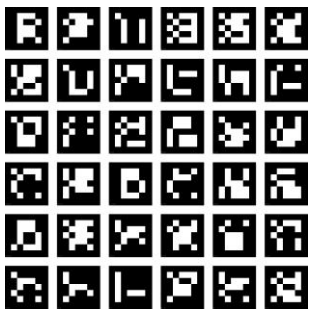


Figura 5: Panel

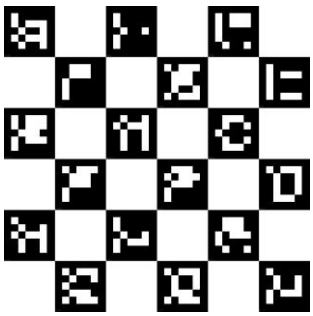


Figura 6: Tablero de ajedrez

Figura 7: Marco

La librería ya viene provista del programa necesario para la creación de este tipo de tableros (ANEXO II).

Para la calibración basta con realizar 8 fotografías del tablero, situado a 45º respecto de la cámara. Se toman 4 fotografías rotando el tablero 90º cada vez, luego se rota la cámara 90º y, girando el tablero 180º , se toman otras 2 fotos, y por último, se gira la cámara 180º y se realizan otras 2 capturas rotando el tablero 180º. Si el tablero utilizado es un tablero de ajedrez de ArUco, el resultado sería el siguiente:

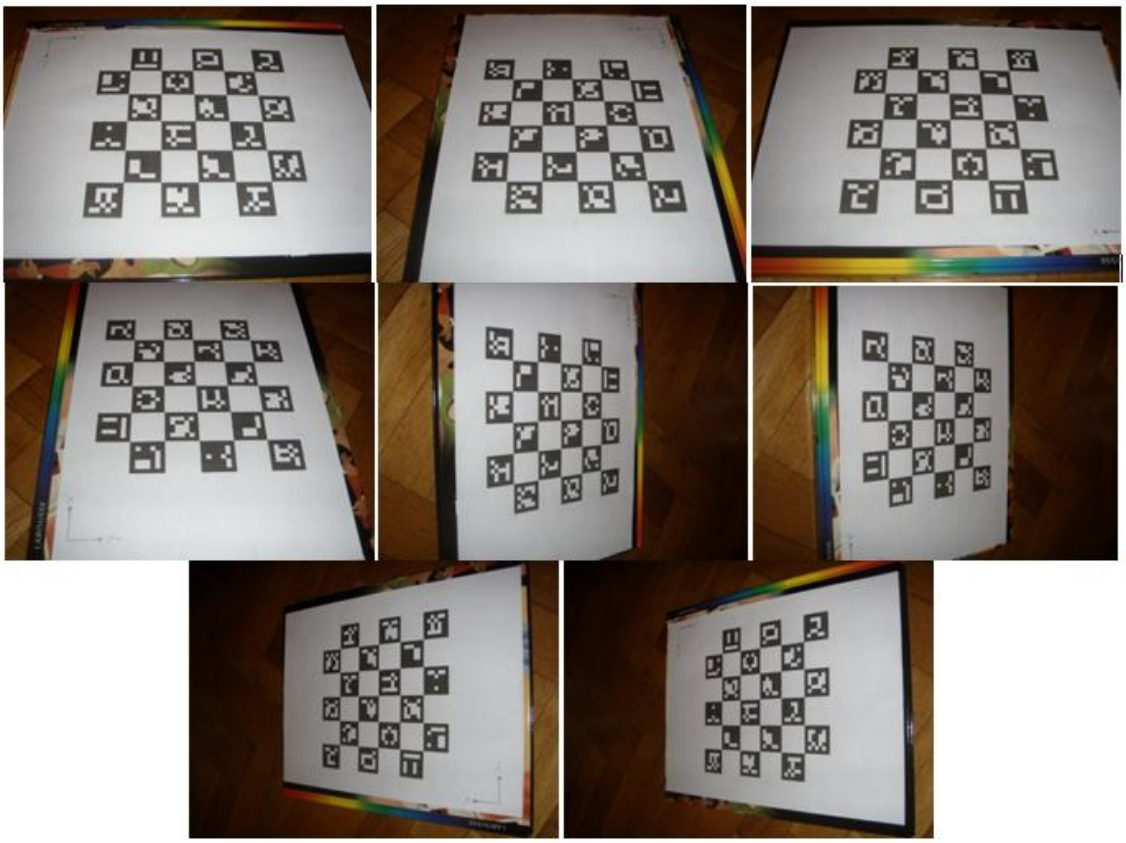


Figura 8: Calibración de la cámara

En este proyecto se ha elegido la segunda opción, debido a que ambas tienen el mismo resultado y así se va familiarizando con el uso de marcadores fiduciales. También se ha optado por calibrar la cámara mediante un video en vivo que vaya tomando imágenes del patrón, al ser el método más rápido de calibración.

Al calibrar la cámara usando la librería ArUco se ha creado un patrón como el de la Figura 8. En primer lugar hay que ejecutar el programa crear_tablero.cpp, en donde, introduciendo los valores necesarios, se crean dos archivos. El primero es una imagen (tablero.jpg) en donde está el tablero deseado y el segundo es un archivo (tablero.yml) que contiene la información de las coordenadas de la esquinas en píxeles de cada marcador fiducial en el tablero, en el ANEXO II se explica detalladamente cómo usar este programa.

Una vez creado el tablero, es necesario pasar la información obtenida en el archivo (tablero.yml) en pixeles a metros. Se ejecuta el programa tablero_metros.cpp e introduciendo

el tamaño de los marcadores fiduciales, se obtiene el archivo tablero_metros.yml con la información del patrón de calibración en metros necesaria para realizar la calibración. La utilización de este programa está explicada en el ANEXO III.

Una vez realizados estos pasos, ya se puede proceder a calibrar la cámara. Se ejecuta el programa calibrar_camara.cpp y moviendo el tablero y la cámara de la forma indicada en el Figura 8, se obtiene el archivo parámetros_camara.yml que contiene los parámetros intrínsecos de la cámara, compuestos por la matriz de la cámara y los coeficientes de distorsión. En el ANEXO IV se expone como usar este programa.

Cuando los parámetros de calibración han sido obtenidos, se procede al diseño de la plataforma multi-robot, para la que será necesario crear los marcadores fiduciales. ArUco cuenta con un programa para realizar esta tarea. Ejecutando crear_marcador.cpp e introduciendo el número de marcador y el tamaño del mismo en pixeles se obtiene el marcador. El uso de este programa se explica en el ANEXO V

Los programas .cpp usados hasta ahora se han modificado ligeramente para adaptarlos a este trabajo, pero conservan casi en su totalidad el contenido inicial desarrollado por los creadores de la librería ArUco.

4. DISEÑO DE LA PLATAFORMA MUTI-ROBOT

La pista por la que se moverán los robots, estará compuesta por 4 marcadores situados a ras de suelo y 4 marcadores situados a la altura de los robots, todos ellos serán los encargados de ubicar los límites de la pista. También habrá 4 zonas con un color determinado, que será el color de los obstáculos (Figura 9).

Figura 9: Plataforma multi-robot

La cámara encargada de grabar la pista se colocará en posición cenital, para así poder aprovechar al máximo las dimensiones de la plataforma y reconocer todos los puntos de interés perdiendo la mínima información posible. Aunque la posición cenital es la posición ideal, la cámara se puede mover siempre y cuando los marcadores no salgan de su campo de visión, los motivos de esta posibilidad se explicarán más adelante en el apartado que describe la matriz homogénea.

Para conocer la distancia máxima a la que se puede poner la cámara respecto de la pista, hay que conocer el tamaño mínimo en píxeles que puede tener el marcador en la imagen. Para obtener este tamaño mínimo, se han creado una serie de marcadores los cuales se han colocado en el suelo y mediante una cámara en posición cenital, se ha ido variando la distancia entre la cámara y los marcadores, obteniéndose los siguientes resultados.

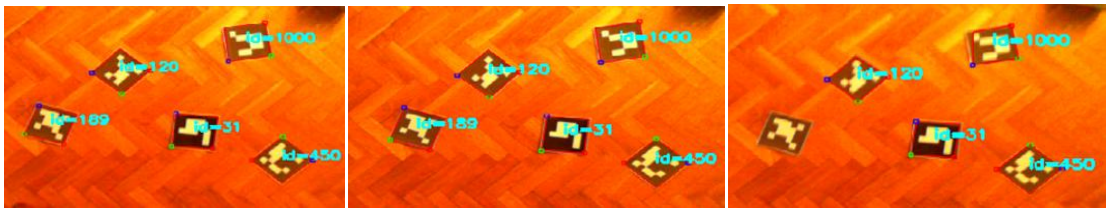


Figura 10: Detección de marcadores a 2.2m, 2.3m y 2.4m respectivamente

En las dos primeras imágenes, los marcadores fiduciales son detectados sin problemas, pero en la tercera imagen ya empieza a haber problemas de detección. Para realizar estas pruebas se ha utilizado una resolución de 640x480 píxeles y el lado de los marcadores creados mide 132mm. Con los resultados obtenidos, no se recomienda poner la cámara a una distancia mayor a 2.3m de los marcadores para este caso concreto.

Colocando la cámara a una distancia de 2.14m respecto del suelo, con una resolución de 640x480 píxeles, las dimensiones de la imagen obtenida son de 2046.6x1534.95 mm. Por lo que, aplicando la ecuación 1, se obtienen las dimensiones de un lado de la pista para una cámara en posición cenital a 2.3 m del suelo, y aplicando la ecuación 2, se obtiene el tamaño mínimo óptimo en píxeles para que el programa reconozca los marcadores fiduciales, que es de 38 píxeles.

$$\text{Ancho } Imagen_2(mm) = \frac{Distancia_{2\ cámara-suelo}(mm) * Ancho\ Imagen\ _1(mm)}{Distancia_{1\ cámara-suelo}(mm)} = \frac{2300*2046.6}{2140} = 2199.6\text{mm}$$

Ecuación 1

$$Tamaño\ marcador\ (píxeles) = \frac{Tamaño\ marcador\ (mm) * Ancho\ imagen\ (píxeles)}{Ancho\ Imagen\ _2(mm)} = \frac{132*640}{2199.617} \simeq 38 \text{ píxeles}$$

Ecuación 2

Otro aspecto muy importante a la hora de diseñar la pista, es conseguir que esta se encuentre lo más uniformemente iluminada, evitando cualquier tipo de brillo, o zonas con distinta luminosidad que dificulten la lectura de los marcadores o modifiquen el aspecto de los colores utilizados.

La primera opción era que la plataforma multi-robot se implementará en un laboratorio del Departamento de Informática e Ingeniería de Sistemas pero se tuvo que desechar. La cámara se encontraba situada en el techo del laboratorio pero la altura del mismo requería unos marcadores excesivamente grandes, algo imposible ya que el tamaño máximo del marcador es el tamaño del robot, debido a que cada robot tiene que llevar un marcador y si este excede el tamaño del robot, habría marcadores que se podrían solapar, impidiendo su lectura correcta. Ante este problema se planteo la alternativa de descolgar la cámara del techo, pero esto requería instalar un sistema capaz de variar la altura de la cámara y, a parte, había otro problema, los focos del laboratorio se reflejaban en el suelo por lo que había zonas en donde ni los marcadores ni los colores podían ser leídos correctamente.

Con todas estas premisas se decidió trasladar la pista y la cámara a mi domicilio, donde se podía conseguir una iluminación más uniforme y ajustar la cámara a la altura deseada más fácilmente.

Como se ha explicado en el apartado anterior, habrá que crear los marcadores que se van a utilizar. Se ha decidido que los 4 marcadores que se situarán en las esquinas de la pista serán los marcadores 21, 22, 23 y 24 mientas que los que se situaran a la altura de los robots serán los números 11, 12, 13 y 14, situados a una distancia D de los marcadores anteriores, como se muestra en la Figura 11.Es recomendable que el valor de D sea lo más pequeño posible, lo que permite obtener una localización más precisa de los robots. También se recomienda dejar un borde de unos 2mm de espesor en blanco a lo largo del marcador para facilitar su detección. Los marcadores habrá que situarlos con la misma orientación con la que aparecen en la Figura 9. Estos marcadores, aparte de poder ser creados con el programa correspondiente, serán

proporcionados en el repositorio. Para situar los marcadores a la altura del robot, se ha optado por utilizar canutos de papel (Figura 12) pero una opción más adecuada para situar el marcador con mayor precisión en la pista sería crear unas cajas de cartón con la altura de los robots y el perímetro de los marcadores.

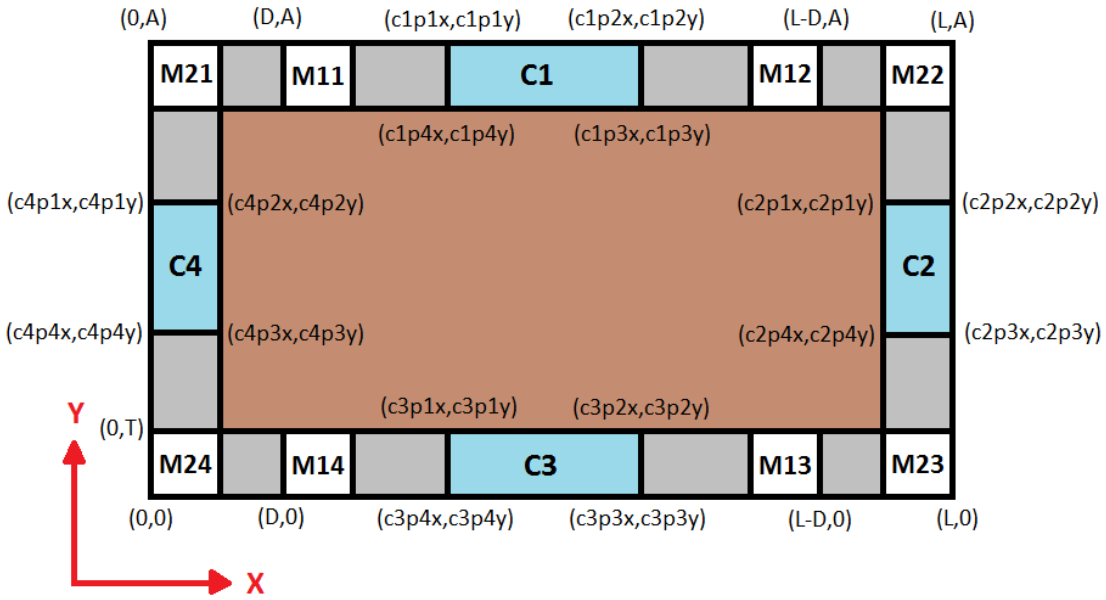


Figura 11: Diseño de la plataforma multi-robot

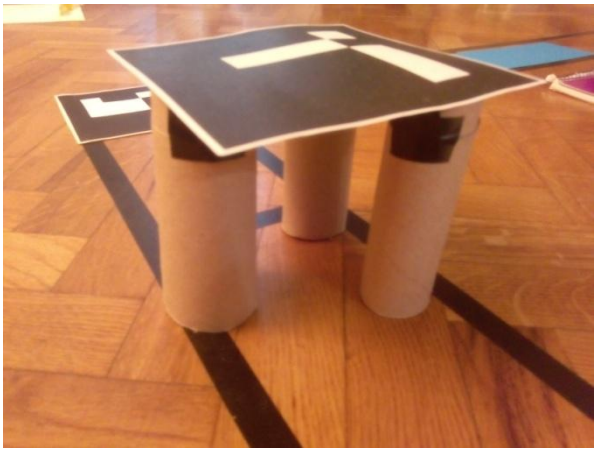


Figura 12: Marcador a la altura de robot

Los 4 patrones del color de referencia (C1, C2, C3 Y C4), del que serán los obstáculos, se situarán en las 4 zonas ubicadas entre los marcadores para no interferir en la pista y las coordenadas de la ubicación de las esquinas de cada color de referencia (c1p1, c1p2, c1p3,…) serán introducidas en el programa.

Así, el tamaño de la pista real por la que se desplazarán los robots será el rectángulo formado por las esquinas interiores de los marcadores 21, 22, 23 y 24, correspondiente al rectángulo marrón de la Figura 11.

Una vez que ya está la plataforma muti-robot montada, se procede a la creación del software para la localización de robots.

5. LOCALIZACIÓN DE MÚLTIPLES ROBOTS MÓVILES Y DETECCIÓN DE OBSTÁCULOS

En el ANEXO VII se encuentran las funciones utilizadas en el programa, así como una breve descripción de su funcionamiento y utilidad. En este capítulo se explicará de forma general el funcionamiento del programa y se analizarán los resultados obtenidos.

5.1 VARIABLES A INTRODUCIR Y FUNCIONAMIENTO DEL PROGRAMA

Antes de empezar a utilizar el programa, hay que introducir todas las coordenadas que aparecen en la Figura 11, la resolución de la cámara, la ubicación del archivo donde se encuentran los parámetros obtenidos al calibrar la cámara, el número de robots que habrá en la pista y el radio de la superficie de los robots.

Con todos estos datos introducidos correctamente, se ejecuta el programa y, partir de aquí, el software realiza las siguientes operaciones en modo bucle.

En primer lugar se corrige la distorsión de la cámara y se procede a reconocer los marcadores fiduciales que limitan la pista. Con los 8 marcadores fiduciales reconocidos, se conoce la posición del centro de cada uno de ellos en la imagen y, al haber introducido anteriormente la información de la pista, también se conoce la ubicación del centro de estos en la pista, por lo que con estos datos, se crean dos matrices de homografía, una para cambiar las coordenadas de los puntos que estén a la altura del suelo (*HO*), y otra para los puntos q estén a la altura de los robots (*HR*), o lo que es lo mismo, una matriz para localizar los obstáculos (*HO*) y otra matriz para localizar los robots (*HR*).

Con la matriz de homografía HR, cuando el programa reconoce un marcador fiducial asignado a un robot, calcula las coordenadas del robot en la pista, así como su orientación.

Para localizar los obstáculos el proceso es más extenso. Las coordenadas de los colores de referencia introducidas se pasan a coordenadas de la imagen con la matriz *HO* y de ahí se obtienen los valores HSV (matiz, saturación, brillo) del color asignado. Como los valores HSV dependen mucho de la iluminación, se calcula la media y desviación típica de los valores HSV de los colores de referencia y se establece un rango en donde todos los puntos dentro de la pista cuyo HSV este dentro de los valores establecidos, es reconocido como obstáculo. Con el rango de colores establecido, se extraen todos los puntos de la imagen que cumplan los requisitos en una imagen binaria y, a partir de esta imagen, se extraen las coordenadas de las esquinas de los obstáculos detectados, que mediante la matriz *HO* se convierten en coordenadas de la pista.

Con todos los datos obtenidos, se crea una imagen en donde se muestra la plataforma multi-robot, los obstáculos detectados y los robots que se encuentran en la pista.

5.2 FUNCIONAMIENTO DEL PROGRAMA DETALLADO

A partir de aquí se describe con mayor detalle el funcionamiento del programa.

Para este trabajo se ha utilizado una resolución 640x480. Se intentó emplear una calidad mayor pero el video a tiempo real iba retrasado por lo que no era una opción válida. La resolución empleada, aunque no sea de una calidad excesivamente buena, es suficiente para detectar los marcadores y permite capturar imágenes a más velocidad. Algo realmente útil si se quiere conocer la localización de un robot que se encuentra en movimiento.

Una vez introducida la resolución, el programa tiene que realizar capturas a frecuencia de video, para esto se realiza un bucle que, hasta que no se pulse la tecla ESC, la cámara grabará lo que ve y realizará instantáneas cada "*tiempocap*" milésimas de segundo. El valor de la variable *tiempocap* habrá sido introducido con anterioridad. Cada captura realizada será guardada y se le aplicarán los parámetros de la matriz de la cámara y los coeficientes de distorsión obtenidos en la calibración, para obtener la imagen corregida.

Esta imagen corregida se guardará en la variable *Imagen*, donde estará la plataforma con los marcadores, los robots y los obstáculos (Figura 13) y se pasará por la función **MDetector.detect()** que se encarga de detectar los marcadores que se encuentran en la pista.

Figura 13: Imagen de la pista

Una vez llegados a este punto, el programa se divide en tres bloques, el primero es la localización de los robots, el segundo es la detección de los obstáculos y el tercero es la creación de una imagen que muestre la plataforma multi-robot, los obstáculos y los robots.

5.2.1 Localización de robots móviles

Para localizar los robots, primero hay que localizar los marcadores que se encuentran en la pista a la misma altura que los robots. Cuando se detecta un marcador, se colorean sus aristas en rojo y se muestra su identificador (Id) (Figura 14). También se guarda la posición del centro en píxeles del marcador detectado.

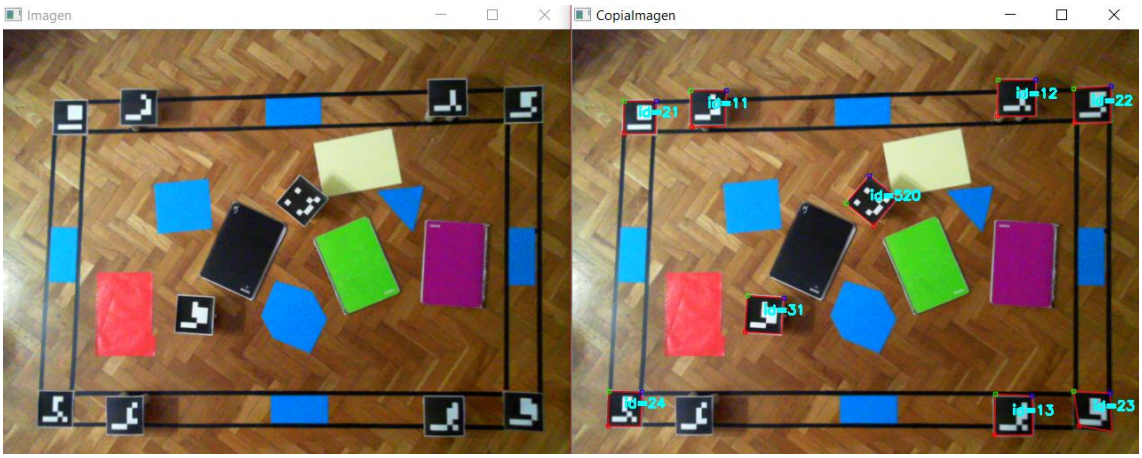


Figura 14: A la izquierda, la imagen de la pista y a la derecha, la misma imagen con los marcadores detectados.

En este trabajo se ha decidido que los marcadores encargados de situar el robot serán los que sus Ids vayan del 11 a 14 (Figura 11), y los robots tendrán marcadores con Ids mayores que 30.

La posición en pixeles del centro de los marcadores se guarda en un vector de 4 filas y la posición en pixeles del robot se guarda en un vector, cuyo número de filas dependerá del número de robots que haya en la pista. Con la función *atan2* se obtendrá la orientación del robot respecto al sistema de coordenadas (theta).

Con la localización de los marcadores y de los robots en la imagen, se quiere obtener las coordenadas de los robots en metros en la pista, esto se consigue con la matriz de homografía. Esta matriz se encarga de trasformas las coordenadas de un punto del mundo a un punto de la cámara (Ecuación 3).

$$\begin{pmatrix} i \\ j \\ 1 \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} * \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Ecuación 3: De coordenadas mundo a coordenadas píxel

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix}^{-1} * \begin{pmatrix} i \\ j \\ 1 \end{pmatrix}$$

Ecuación 4: De coordenadas píxel a coordenadas mundo

Las coordenadas (x,y) correspondes a las coordenadas en el mundo y las coordenadas (i,j) a sus homólogas en pixeles.

OpenCV cuenta con la función **findHomography** para obtener dicha matriz, pero hay que introducirle una serie de valores. OpenCV calcula la matriz de homografía a partir de 2 vectores (2D) de 4 filas. Esto quiere decir que hay que introducir las coordenadas (2D) en píxeles de la imagen y sus coordenadas (2D) en metros de la posición que ocupa ese punto en el mundo. Este proceso hay que hacerlo con 4 puntos, por lo que se acaban introduciendo un

total de 8 puntos (2D), 4 en pixeles y sus correspondientes en metros. Las coordenadas introducidas serán la ubicación del centro de los marcadores en píxeles y en metros.

Usar este método para obtener las coordenadas permite que la matriz de homografía se recalcule a frecuencia de video. Las coordenadas en metros del centro de los marcadores no cambiarán, y las coordenadas en píxeles serán calculadas por el programa cada vez que se realiza una instantánea. Con la matriz de homografía obtenida solo habrá que introducir las coordenadas de un punto en metros y se obtendrán sus homólogas en píxeles y viceversa. Con todas estas premisas, la cámara se pude mover y, siempre y cuando este enfocando la pista y el programa reconozca los marcadores, dará igual el ángulo que tenga respecto de la pista, debido a que el programa recalculará las matrices de homografía y las coordenadas obtenidas con estas matrices.

A pesar de la ventaja de este método y de la posibilidad de colocar la cámara en diferentes posiciones, se recomienda poner la cámara en una posición cenital, para una mejor lectura de la pista.

Como se ha mencionado anteriormente, con la función **findHomography** se obtiene la matriz de homografía, a esta función se le introducen las coordenadas del centro de los marcadores a la altura de los robots tanto en metros como en píxeles y se obtiene la matriz de homografía `HR`. Una vez obtenida dicha matriz, solo habrá que pasarle la localización de los robots en pixeles detectados en la imagen y se obtendrá la localización de dichos robots en la pista (x,y,theta).

Como se ha mencionado antes, se creará una imagen donde aparecen los robots, por lo que será necesario conocer el tamaño de estos. Los robots Arduino (Figura 15) que se emplearán tienen una superficie circular, por lo que el usuario tendrá que introducir el valor del radio del robot en metros en la variable *radioRobot*, y mediante la ecuación 5, se obtendrá su valor en píxeles.

$$\text{Radio (píxeles)} = \frac{radioRobot(metros) * tamaño\ de\ la\ pista\ (píxeles)}{tamaño\ de\ la\ pista\ (metros)}$$

Ecuación 5: Cálculo del radio del robot en píxeles

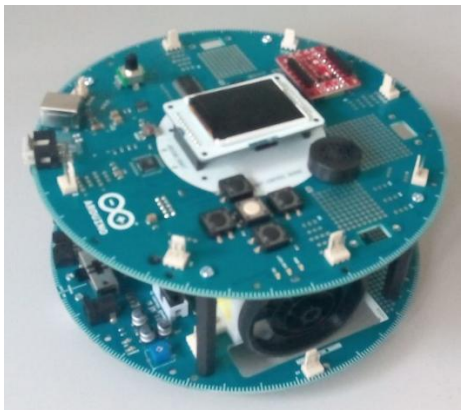


Figura 15: Robot-móvil Arduino

Por último, se mostrar por pantalla la posición de los robots (x,y,theta) en la pista (Figura 16).

```
Robot[1](x,y,theta):[0.473127, 0.294792, -0.302896]
Robot[2](x,y,theta):[0.850856, 0.692568, 2.44641]
```

Figura 16: Posición de los robots

5.2.2 Detección de obstáculos

La primera parte de este proceso es parecida a la utilizada para localizar los robots.

Se ha decidido que los marcadores que tienen Ids de 21 a 24 (Figura 11) serán los que se coloquen a ras de suelo por lo que, introduciendo la posición de estos metros y obteniendo su localización en píxeles, mediante la función **findHomography**, se creará la matriz de homografía HO que se utilizará para obtener las coordenadas de los obstáculos.

Como se he mencionado anteriormente, la detección de obstáculos se va a realizar mediante colores por lo que, para mejorar la detección de estos, la imagen en BGR (Azul, verde, rojo) se va a pasar a HSV (matiz, saturación, brillo) (Figura 17), lo que facilita la elección y detección de colores.

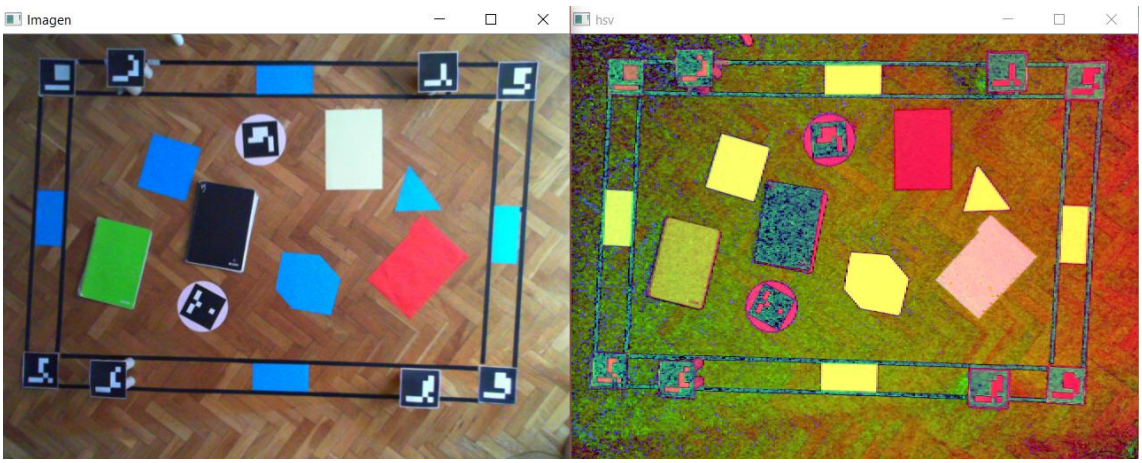


Figura 17: A la izquierda la imagen de la pista en BGR y a la derecha, la misma imagen en HSV

Cuando la imagen está en HSV, hay que elegir el color de los obstáculos. Para este proyecto se ha optado por el color azul y, para evitar problemas de brillo y luminosidad, se ha decidido que 4 cartulinas con forma rectangular del color elegido se colocarán en los extremos e la pista, entre los marcadores fiduciales (Figuras 9 y 11). Al programa se le pasarán las coordenadas mundo de la ubicación de las esquinas de cada cartulina y estas coordenadas se pasarán a píxeles, para que el programa pueda obtener los valores HSV del color deseado. El proceso de transformación de coordenadas mundo a pixeles, es parecido a los casos anteriores, salvo que para este caso, se utilizará la nueva matriz homogénea (*HO*) creada con los marcadores a ras de suelo.

Para extraer los valores HSV de la imagen, OpenCV recorre de izquierda a derecha y de abajo a arriba cada punto de la imagen, pero si la cámara no se encuentra en la posición cenital, la imagen de la pista puede estar deformada, por lo que se ha creado un código para obtener las

coordenadas del rectángulo interior formado por las esquinas del color de referencia (Figura 18), para asegurarse que todos los puntos leídos al detectar el color de referencia, son del color elegido.

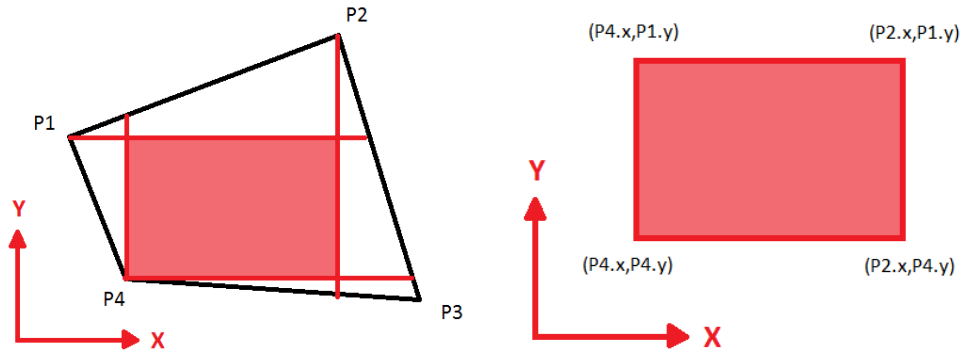


Figura 18: Rectángulo interior formado por las esquinas del trapezoide

Una vez aplicado el código mencionado a las coordenadas del color de referencia en píxeles y obtenido las coordenadas del los rectángulos interiores, hay que extraer el color que tiene cada rectángulo. OpenCV tiene una función (**meanStdDev**) con la que, introduciendo dichas coordenadas, extrae el color de cada punto y guarda, en forma de variables, los valores medios de matiz, saturación y brillo, así como sus desviaciones típicas (Figura 19).

```
color 1
H - media: 102.417  desviación: 0.724992
S - media: 254.976  desviación: 0.312506
V - media: 255  desviación: 0

color 2
H - media: 90.144  desviación: 0.351135
S - media: 250.572  desviación: 4.49099
V - media: 255  desviación: 0

color 3
H - media: 102.863  desviación: 0.73955
S - media: 254.992  desviación: 0.130061
V - media: 255  desviación: 0

color 4
H - media: 106.087  desviación: 0.829524
S - media: 254.995  desviación: 0.0817048
V - media: 224.8  desviación: 7.78621
```

Figura 19: Media y la desviación típica de los valores HSV de cada color de referencia

Como se aprecia en la Figura 19, cada cartulina de referencia tiene unos valores medios de HSV y unas desviaciones típicas, ambos limitados con un rango de 0 a 255. El motivo por el cual los valores son distintos es debido esencialmente a la variación de luminosidad en las distintas zonas de la pista.

Como lo que se quiere es que todos los obstáculos que sean del mismo color que el color de referencia sean detectados, se establece un rango de valores para el cual, todos los puntos cuyo HSV que se encuentren dentro de ese rango, serán reconocidos como obstáculos.

El valor de referencia se establece como el valor medio entre el valor máximo y mínimo de las medias de HSV obtenidas anteriormente y este valor medio habrá que aplicarle el rango mencionado anteriormente. El rango será la suma de la diferencia entre el valor medio mínimo y el valor medio máximo de los colores de referencia, más la desviación típica máxima multiplicada por un factor. Este valor se sumará y se restara al valor medio obtenido anteriormente para lograr el umbral de valores aceptables para que un punto sea reconocido como obstáculo.

Este umbral de valores se pasa a la función **inRange** que crea una imagen binaria que muestra en blanco los puntos de la imagen que se encuentran dentro del rango admitido y son detectados como obstáculos (Figura 20).

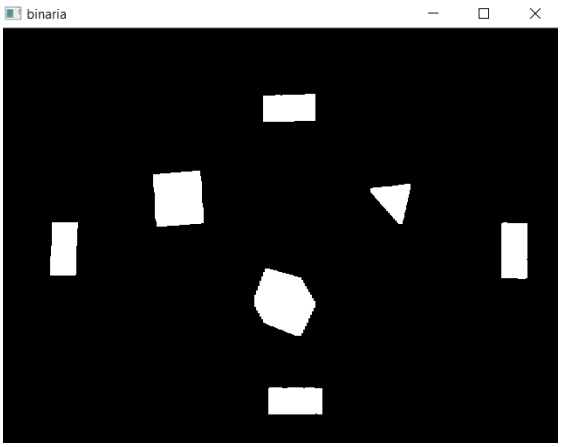


Figura 20: Imagen binaria de los obstáculos

Para mejorar detección, se va a modificar la imagen binaria. A esta imagen se le aplican una serie de operaciones de erosión y dilatación, la primera para eliminar posibles puntos erróneos detectados como objetos, pues reduce el tamaño de los obstáculos detectados y la segunda para devolver a los obstáculos a su tamaño original (Figura 21).

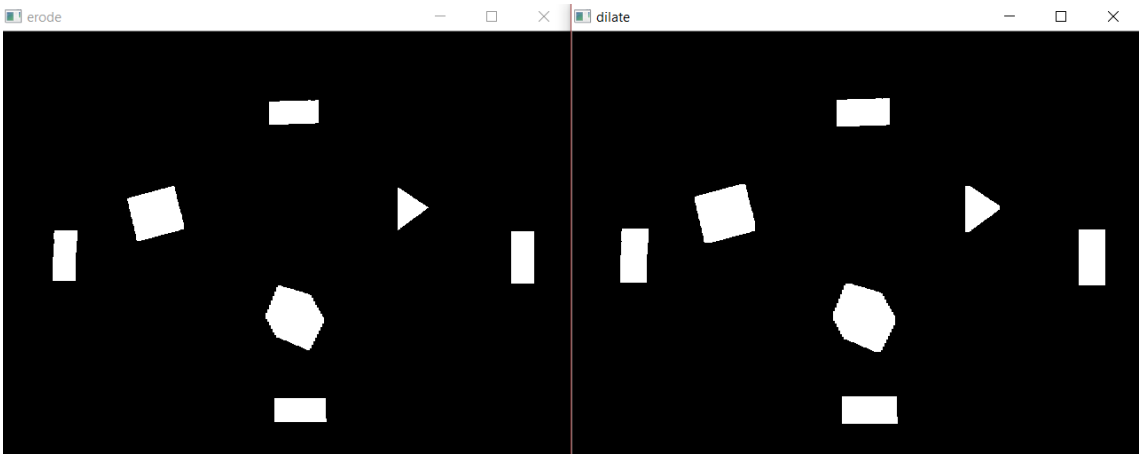


Figura 21: A la izquierda, la imagen binaria erosionada y a la derecha, la imagen dilatada.

Debido a que la calidad de la imagen no es excesivamente buena, se ha decidido disminuir el tamaño de la imagen, para que a la hora de detectar las esquinas de los obstáculos, no detecte falsas esquinas. De esta imagen, se extraen los contornos de los obstáculos y, posteriormente, en una imagen en negro, se dibujan los contornos extraídos y se rellenan. Esta operación se

realiza porque da igual si los obstáculos tienen agujeros interiores, debido a que el robot no podría acceder a ellos. De esta forma también se evita que puntos dentro del obstáculo, que por alguna razón no hayan sido reconocidos como tales, creen lecturas falsas y, a la hora de detectar las esquinas de los obstáculos, aparezcan esquinas interiores que de nada sirven.

Con los obstáculos ya detectados, hay que proceder a localizar las esquinas. La detección de esquinas se realiza con dos funciones, **goodFeaturesToTrack** y **cornerSubPix**. La primera lee los parámetros introducidos por el usuario y detecta las esquinas en la imagen, con la segunda función, se obtienen unos valores más ajustados de las coordenadas de las esquinas. Como en el ejemplo de la Figura 22, donde se muestran la localización de las esquinas de un triángulo con los dos métodos.

```
Corner [0]   (136,67)       Refined Corner [0]   (136.648,68.0344)
Corner [1]   (138,54)       Refined Corner [1]   (139.368,52.4062)
Corner [2]   (127,55)       Refined Corner [2]   (124.761,54.1801)
```

Figura 22: A la izquierda, las esquinas detectadas en un triángulo después de aplicar el método gooFeaturesToTrack y a la derecha, después de aplicar el método cornerSubPix.

Una vez que las esquinas han sido detectadas, sus coordenadas se guardan en un array, pero como la imagen de donde se han obtenido dichos valores había sido reducida, hay que realizar un cambio de escala para obtener la localización de la esquina en la imagen real, esto se consigue dividiendo cada componente del array por el factor utilizado anteriormente para reducir la imagen, que en este caso es 0.3.

Con las coordenadas de las esquinas guardadas, hay que obtener sus coordenadas en la pista, este proceso, como se he ido realizado en numerosas ocasiones en este programa, se realiza con la matriz de homografía *HO,* obteniéndose el vector con la localización en coordenadas mundo de las esquinas del obstáculo. Como el obstáculo detectado puede que esté dentro o fuera de la pista o ambas a la vez, se ha decidido que si la localización de alguna de las esquinas del obstáculo se encuentra fuera de la pista, se reubicarán con las coordenadas que limitan la pista.

Por último, se mostrarán por pantalla las coordenadas de las esquinas de cada obstáculo (Figura 23).

```
obstáculo[1] esquina[1]: [0.740463, 0]
obstáculo[1] esquina[2]: [0.947948, 0]
obstáculo[1] esquina[3]: [0.7434, 0]
obstáculo[1] esquina[4]: [0.950685, 0]
obstáculo[2] esquina[1]: [0.750042, 0.419486]
obstáculo[2] esquina[2]: [0.874864, 0.381533]
obstáculo[2] esquina[3]: [0.87755, 0.1462]
obstáculo[2] esquina[4]: [0.744112, 0.194767]
obstáculo[2] esquina[5]: [0.692287, 0.274687]
obstáculo[2] esquina[6]: [0.94548, 0.271927]
```

Figura 23: Coordenadas de las esquinas detectadas

5.2.3 Imagen con la plataforma multi-robot

Para mejorar la visualización de los resultados, se ha decidido crear una imagen en donde aparezca la pista por la que se pueden mover los robots, los obstáculos que hay en el terreno y los robots detectados, así como su orientación (Figura 24). Independientemente de la posición que tenga la cámara, la imagen creada mostrará solo la pista por la que se desplazarán los robots, todo lo que se encuentre fuera, será omitido.

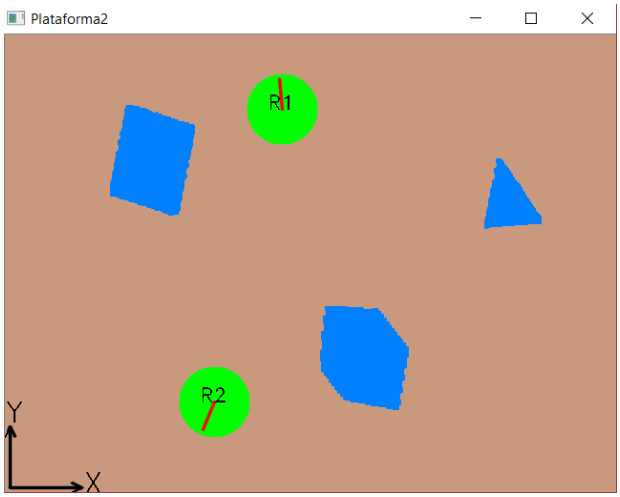


Figura 24: Pista por la que se mueven los robots con los obstáculos y los robots detectados.

La pista, será representada con el color marrón, y los obstáculos con el color azul. Los robots, al ser unos robots Arduino, tienen una superficie circular, por lo que se representan con un círculo de color verde. Para representar el ángulo theta que tiene el robot, se ha dibujado una línea de color rojo que nace en el centro del robot, cuando esta línea este orientada en la dirección del eje x y sea paralela a este, el ángulo del robot será igual a 0. Estos valores se mostrarán en radianes. Cada robot dibujado llevara un identificador (R1, R2,…) para tenerlo localizado en todo momento. También se dibujarán los ejes de coordenadas en la imagen.

Para crear esta imagen, se ha creado una nueva matriz de homografía que permite pasar cada uno de los puntos detectados en la imagen en píxeles a la nueva imagen donde solo aparecerá la plataforma multi-robot.

5.3 RESULTADOS

El resultado de ejecutar el programa creado es el siguiente, en la Figura 25 se muestra, a la izquierda, la pista real y a la derecha, la imagen creada como se ha descrito anteriormente.

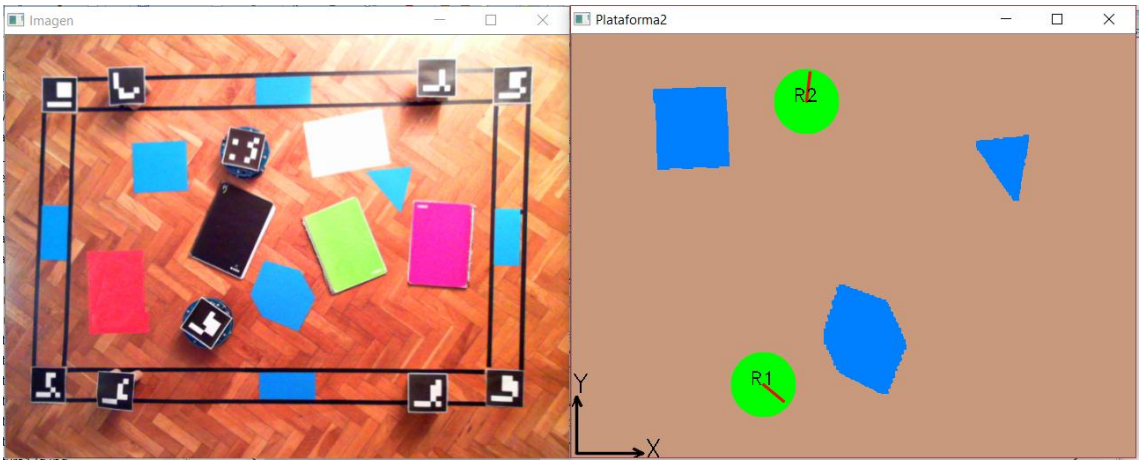


Figura 25: A la izquierda la imagen de la pista real y a la derecha la pista por la que se mueven los robots con los obstáculos y los robots detectados.

A pesar de haber varios colores en la pista, programa solo reconoce como obstáculos los que son del color indicado como color de referencia.

A parte de estas imágenes, el programa muestra por pantalla las coordenadas de los robots y de los obstáculos.

```
Robot[1](x,y,theta):[0.559467, 0.176463, -0.604264]
Robot[2](x,y,theta):[0.67671, 0.815171, 1.34377]
```

Figura 26: Ubicación de los robots en la pista.

```
obstáculo[1] esquina[1]: [0.740175, 0]
obstáculo[1] esquina[2]: [0.944732, 0]
obstáculo[1] esquina[3]: [0.742965, 0]
obstáculo[1] esquina[4]: [0.946831, 0]
obstáculo[2] esquina[1]: [0.665548, 0.309521]
obstáculo[2] esquina[2]: [0.720425, 0.444334]
obstáculo[2] esquina[3]: [0.861858, 0.175407]
obstáculo[2] esquina[4]: [0.719949, 0.222388]
obstáculo[2] esquina[5]: [0.891627, 0.237368]
obstáculo[2] esquina[6]: [0.853465, 0.409748]
obstáculo[3] esquina[1]: [1.586, 0.606365]
obstáculo[3] esquina[2]: [1.586, 0.604729]
obstáculo[3] esquina[3]: [1.586, 0.392205]
obstáculo[3] esquina[4]: [1.586, 0.378858]
obstáculo[4] esquina[1]: [0, 0.419495]
obstáculo[4] esquina[2]: [0, 0.633468]
obstáculo[4] esquina[3]: [0, 0.631655]
obstáculo[4] esquina[4]: [0, 0.41838]
obstáculo[5] esquina[1]: [1.22613, 0.58683]
obstáculo[5] esquina[2]: [1.09928, 0.740681]
obstáculo[5] esquina[3]: [1.27134, 0.753205]
obstáculo[6] esquina[1]: [0.450472, 0.831106]
obstáculo[6] esquina[2]: [0.476737, 0.602628]
obstáculo[6] esquina[3]: [0.289563, 0.589971]
obstáculo[6] esquina[4]: [0.267184, 0.811493]
obstáculo[7] esquina[1]: [0.687968, 0.991]
obstáculo[7] esquina[2]: [0.900145, 0.991]
obstáculo[7] esquina[3]: [0.90247, 0.991]
obstáculo[7] esquina[4]: [0.691346, 0.991]
```

Figura 27: Coordenadas de las esquinas de los obstáculos.

Tanto la localización de los robots como la de las esquinas de los obstáculos son detectadas perfectamente. Se aprecia como los obstáculos 2, 5, y 6 corresponden al hexágono, al triángulo y al cuadrado respectivamente y el resto a los colores de referencia.

Para comprobar el correcto funcionamiento del programa, se ha ido variando la luminosidad de la pista y se colocado la cámara en diferentes posiciones, con los siguientes resultados:

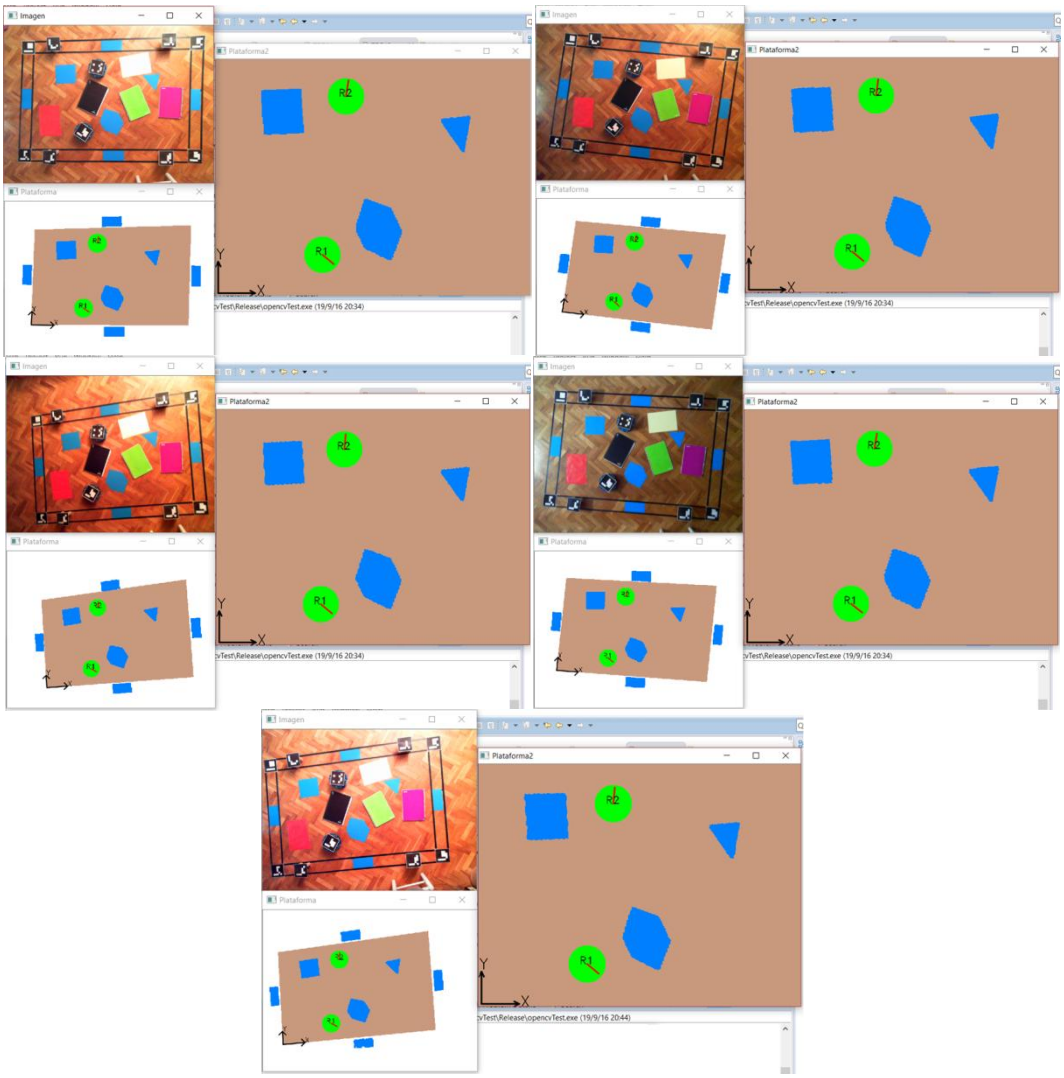
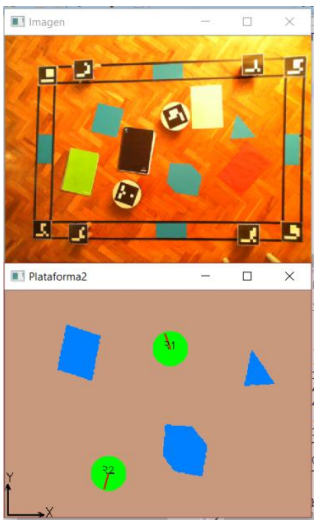


Figura 28: Plataforma multi-robot en diferentes posiciones respecto de la cámara y con variaciones de luminosidad.

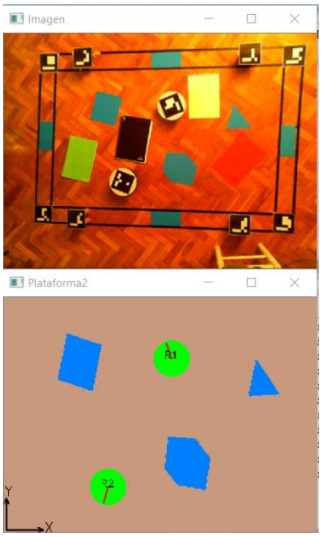
A continuación se muestran las coordenadas de las esquinas de los obstáculos y de los robots, solo se mostrarán los obstáculos que se encuentran dentro de la pista.



Robot[1](x,y,theta):[0.857033, 0.697684, 2.07449]
Robot[2](x,y,theta):[0.540283, 0.202667, -1.86787]

obstáculo[2] esquina[1]: [0.821761, 0.404704]
obstáculo[2] esquina[2]: [0.869779, 0.198858]
obstáculo[2] esquina[3]: [1.01499, 0.175958]
obstáculo[2] esquina[4]: [1.04915, 0.314451]
obstáculo[2] esquina[5]: [0.820769, 0.256646]
obstáculo[2] esquina[6]: [0.969701, 0.405558]
obstáculo[5] esquina[1]: [1.23176, 0.574219]
obstáculo[5] esquina[2]: [1.40345, 0.581589]
obstáculo[5] esquina[3]: [1.2778, 0.741121]
obstáculo[6] esquina[1]: [0.320228, 0.858996]
obstáculo[6] esquina[2]: [0.506727, 0.806507]
obstáculo[6] esquina[3]: [0.263487, 0.649864]
obstáculo[6] esquina[4]: [0.457557, 0.599206]

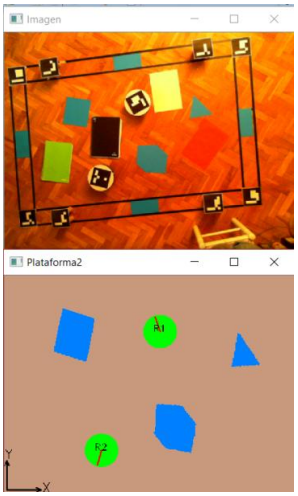
Figura 29: 1º Resultados obtenidos al mover la cámara



Robot[1](x,y,theta):[0.857954, 0.697594, 2.07351]
Robot[2](x,y,theta):[0.540901, 0.2021, -1.87134]

obstáculo[2] esquina[1]: [1.00979, 0.172458]
obstáculo[2] esquina[2]: [0.817771, 0.414177]
obstáculo[2] esquina[3]: [1.03897, 0.331451]
obstáculo[2] esquina[4]: [0.98032, 0.398165]
obstáculo[2] esquina[5]: [0.813374, 0.253857]
obstáculo[2] esquina[6]: [1.03897, 0.331452]
obstáculo[5] esquina[1]: [1.23022, 0.576377]
obstáculo[5] esquina[2]: [1.27946, 0.736033]
obstáculo[5] esquina[3]: [1.39779, 0.59036]
obstáculo[6] esquina[1]: [0.506593, 0.806475]
obstáculo[6] esquina[2]: [0.269363, 0.651354]
obstáculo[6] esquina[3]: [0.458673, 0.602741]
obstáculo[6] esquina[4]: [0.310848, 0.854631]

Figura 30: 2º Resultados obtenidos al mover la cámara



Robot[1](x,y,theta):[0.858397, 0.698242, 2.07418]
Robot[2](x,y,theta):[0.541478, 0.202942, -1.86942]

obstáculo[2] esquina[1]: [1.02711, 0.174684]
obstáculo[2] esquina[2]: [0.830143, 0.40933]
obstáculo[2] esquina[3]: [0.824624, 0.238966]
obstáculo[2] esquina[4]: [0.859337, 0.201359]
obstáculo[2] esquina[5]: [1.06093, 0.301697]
obstáculo[5] esquina[1]: [1.40024, 0.587124]
obstáculo[5] esquina[2]: [1.23447, 0.565363]
obstáculo[5] esquina[3]: [1.28524, 0.734066]
obstáculo[6] esquina[1]: [0.509485, 0.807035]
obstáculo[6] esquina[2]: [0.321888, 0.853909]
obstáculo[6] esquina[3]: [0.274464, 0.6458]
obstáculo[6] esquina[4]: [0.463646, 0.599478]

Figura 31: 3º Resultados obtenidos al mover la cámara

El obstáculo 2 corresponde al polígono de 5 puntas, el obstáculo 5 al de 3 y el 6 al de 4. Se observa que para el polígono de 5 puntas, se han detectado 6 esquinas, una más de las que deberían. Esto se debe a que la calidad de la imagen no es excesivamente buena y el principal problema de este software es la detección de esquinas.

Para solventar este problema, se han utilizado métodos refinados para la localización de esquinas, pero aun así es difícil localizar el punto exacto de la esquina con precisión. Cuanto más pronunciadas sea la esquina, más fácil será esta de detectar, esto se consigue ajustando el valor de la variable *qualityLevel*, para valores bajos, detectará esquinas poco pronunciadas y para valores altos, solo esquinas muy pronunciadas. Este error en la detección se considera aceptable ya que el número de esquinas detectado no es mucho mayor que el número de esquinas reales y es preferible que se detecten más esquinas de las que hay, pues la figura geométrica resultante variará poco, a que se detecten menos esquinas, ya que se obtendría otra figura completamente distinta.

Analizando las Figuras 29, 30 y 31 si se compara la localización de los robots con la de las esquinas, se observa como el error máximo cometido en el primer caso está en torno a un 0.001 mientras que en segundo caso se aproxima al 0.003. Si se analiza el ángulo theta de los robots, se aprecia que el error cometido es mayor que en la coordenadas x e y del robot, esto es debido a que para obtener el ángulo del robot, se han utilizado 2 esquinas de los marcadores, mientras que para obtener las coordenadas, se han utilizado 4, lo que otorga mayor precisión.

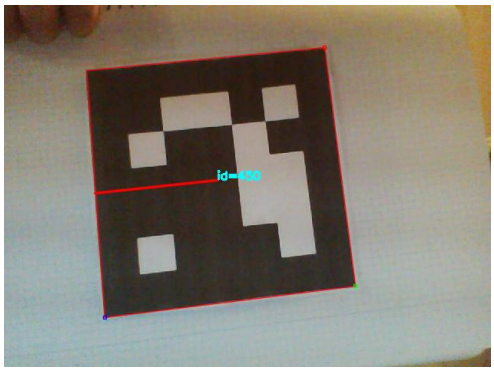
Un porcentaje de este error cometido vendrá de la perdida de información al mover la cámara, Esto se debe a que, al colocar la cámara en distintas posiciones, se va perdiendo información respecto a la posición ideal, que es la posición cenital y, cuanto mayor sea la distancia respecto de dicha posición, mayor será el error cometido. Pero el mayor porcentaje del error cometido, vendrá dado por que para que haya una mejor detección de esquinas, se ha reducido la imagen y se han obtenido las coordenadas de las esquinas, posteriormente, estos valores se han multiplicado por la reducción hecha anteriormente, que en este caso ha sido de 3,33, por lo que el error que se hubiera cometido en la detección, queda multiplicado por ese factor de conversión.

Se ha decidido dar por bueno estos porcentajes, ya que se está hablando de errores máximos, que en la mayoría de detecciones no ocurre.

Tanto el número asignado a cada obstáculo en la detección como el número asignado a cada esquina puede variar, ya que la imagen se analiza de izquierda a derecha y de abajo a arriba por lo que, según la orientación del la pista, encontrará antes unos obstáculos u otros y unas esquinas u otras.

Se puede afirmar que, mientras los marcadores sean reconocidos en la imagen, no importa la ubicación de la cámara y la luminosidad en la pista (exceptuando variaciones muy drásticas de luminosidad). El programa recalculara todos los valores que se hayan modificado y mostrará por pantalla la pista con los robots y los obstáculos a frecuencia de video, además de las coordenadas de los robots y de las esquinas de los obstáculos.

Para colocar de forma correcta el marcado en el robot a modo de ayuda para futuros trabajos, se ha creado un programa (ANEXO VI), en el que mostrando el marcador que se va a colocar al robot a la cámara, se muestra por pantalla el ángulo theta y lo dibuja con una línea roja en el marcador (Figura 32).



Marcador 450 ángulo (radianes): -3.01837

Foto 32: Representación de la orientación del marcador

A parte de los resultados obtenidos, se han querido conocer el tiempo que le cuesta al software realizar cada operación, para conocer la frecuencia a la que es capaz de localizar los objetos y los robots. En la tabla 2 se muestran los procesos más lentos y el tiempo total medio obtenido al realizar 18 iteraciones.

Proceso realizado	Tiempo (segundos)	Tiempo (%)
Compensación de la distorsión	0,0207	24,12
Detección de marcadores	0,0271	31,57
Detección de obstáculos	0,0247	28,75
Total	0,0858	100,00

Tabla 2: Tiempos de operación de los procesos más lentos

Se observa como la suma de los tres procesos más lentos es casi la totalidad del tiempo que emplea el software para procesar las imágenes. En los dos primeros pasos, se emplea únicamente una función para realizar cada proceso, y el tiempo empleado por estas dos funciones ocupa en torno al 56% del tiempo total, por lo que, por mucho que se optimice el código, también habrá que buscar la manera de reducir el tiempo empleado por estas dos funciones para conseguir el programa funcione a frecuencia de video. El objetivo era que el programa se ejecutase a frecuencia de video (24 fps), pero esto no ha sido posible, aún así, el programa es capaz de realizar y procesar unas 11 imágenes por segundo.

Una alternativa para reducir el tiempo de operación, es utilizar una cámara proyectiva, en donde el error cometido por la distorsión sea despreciable, por lo que no sería necesario emplear una función que corrija este error y el programa sería capaz de procesar unas 15 imágenes por segundo, un resultado notablemente superior al anterior, que otorgaría mayor precisión en la localización de robots y obstáculos. Otra opción seria, en vez de compensar la distorsión en toda la imagen, hacerlo solo en los puntos de interés, lo que aumentaría notablemente el tiempo de operación de esta función.

6. CONCLUSIONES

El objetivo principal de este trabajo era la creación de un software que fuera capaz de localizar los robots móviles y los obstáculos que se encuentran en la plataforma multi-robot.

Después de analizar los resultados obtenidos, se ha conseguido crear un software capaz de detectar los robots, los obstáculos y la pista, independientemente de los cambios de luminosidad que se produzcan y de la posición de la cámara, debido a que el programa es capaz de autocalibrarse, a una velocidad de 11 fps (imágenes por segundo) y con un error del 0.3%. El software estará sujeto a las condiciones de una licencia GPL.

La localización de la plataforma se ha conseguido mediante una serie de marcadores fiduciales obtenidos de la librería ArUco. El principal problema de estos marcadores es su lectura, debido a que se requiere un tamaño mínimo del marcador en pixeles para que su lectura sea correcta. Como el tamaño de los marcadores está limitado por el robot, la cámara en posición cenital tiene una altura máxima a la que se puede colocar para que identifique todos los marcadores, lo que limita las dimensiones de la plataforma multi-robot.

La detección de obstáculos se ha realizado mediante colores, se establece un color de referencia y todos los puntos en la pista cuyo color esté dentro de un rango establecido, son considerados obstáculos. Para localizar el obstáculo, se obtienen sus esquinas, pero el problema que esto conlleva es que, si la imagen procesada no tiene una buena resolución, y los pixeles son demasiado grandes, el programa puede equivocarse y lo que a priori sería una línea recta, es dibujada en la imagen como una escalera y se detectan numerosas esquinas que no tendrían que aparecer. Este problema se ha reducido limitando el número máximo de esquinas que pueden aparecer, la distancia mínima que tiene que haber entre ellas y la robustez que deben de tener los puntos analizados para ser consideras como esquinas. Aún con todo, en algunas ocasiones, pueden aparecer algunas esquinas más de la de deberían.

Otro problema detectado a la hora de realizar este trabajo es la luminosidad, por este motivo se tuvo que trasladar la plataforma multi-robot, que en un principio se iba a implantar en el laboratorio del Departamento de Informática e Ingeniería de Sistemas. Las luces del laboratorio impedían la correcta lectura tanto de los marcadores como de los colores, por eso es recomendable instalar la plataforma en una zona con una luminosidad lo más uniformemente posible. Aunque el programa es capaz de detectar los cambios de luminosidad y ajustarse con buenos resultados, siempre habrá menos errores si la zona esta uniformemente iluminada.

En definitiva, el punto fuerte de este software es que es capaz de autoajustares y seguir funcionando correctamente si la cámara se mueve o si hay alguna variación de luminosidad en la plataforma multi-robot.

Como líneas de trabajo futuras se propone, a parte de la implementación del código en la Toolbox de MATLAB para generar las trayectorias que recorrerán los robots, la ejecución del programa con una cámara que permita trabajar a una calidad mayor y comprobar si el problema en la detección de esquinas es solventado y si no es el caso, buscar un método de

detección de esquinas para reducir al mínimo posible el error cometido. También se propone la optimización de código para poder aumentar el número de imágenes procesadas por segundo.

7. BIBLIOGRAFÍA

"Aruco: A Minimal Library For Augmented Reality Applications Based On Opencv | Aplicaciones De La Visión Artificial". 2016. Uco.Es. http://www.uco.es/investiga/grupos/ava/node/26.

"Calibrar Una Cámara Con Opencv (Parte II)". 2016. *Tecnicasdevision.Blogspot.Com*. http://tecnicasdevision.blogspot.com/2014/11/calibrar-una-camara-con-opencv-parte-ii.html.

“CAPITULO 2. Navegación en Robots Móviles”. Planificación de Trayectorias para Robots Móviles. http://webpersonal.uma.es/~VFMM/PDF/cap2.pdf.

"Cplusplus.Com - The C++ Resources Network". 2016. Cplusplus.Com. http://www.cplusplus.com/.

D. Lélis Baggio et al. 2012. “Mastering Opencv With Practical Computer Vision Projects”. Birmingham: Packt Pub.

"Dibujar Formas Y Texto". 2013. *Acodigo.Blogspot.Com*. http://acodigo.blogspot.com/2013/05/dibujar-formas-y-texto.html.

"Eclipse (Software)". 2016. *Es.Wikipedia.Org*. https://es.wikipedia.org/wiki/Eclipse_(software).

G. Bradski and A. Kaehler. 2008. “Learning OpenCV”. http://www.bogotobogo.com/cplusplus/files/OReilly%20Learning%20OpenCV.pdf.

Garrido Pérez, José Carlos. 2014. "Desarrollo De Una Librería Para Acceder A Aruco Desde Java". Licenciatura, Universidad de Córdoba.

"Historia De Los Robots | Wiki De Robótica". 2016. *Wiki.Robotica.Webs.Upv.Es*. http://wiki.robotica.webs.upv.es/wiki-de-robotica/introduccion/historia/.

Examples, Basic. 2016. "Basic Drawing Examples". *Opencvexamples.Blogspot.Com*. http://opencvexamples.blogspot.com/2013/10/basic-drawing-examples.html.

"Newest Questions". 2016. *Stackoverflow.Com*. http://stackoverflow.com/questions/.

"Opencv". 2016. *Es.Wikipedia.Org*. https://es.wikipedia.org/wiki/OpenCV.

"Opencv API Reference — Opencv 2.4.9.0 Documentation". 2016. Docs.Opencv.Org. http://docs.opencv.org/2.4.9/modules/refman.html.

"Opencv Tutorials — Opencv 2.4.9.0 Documentation". 2016. Docs.Opencv.Org. http://docs.opencv.org/2.4.9/doc/tutorials/tutorials.html.

"Opencv User Guide — Opencv 2.4.9.0 Documentation". 2016. Docs.Opencv.Org. http://docs.opencv.org/2.4.9/doc/user_guide/user_guide.html.

Ortiz, Alberto. "Navegación Para Robots Móviles". Presentation, http://dmi.uib.es/aortiz/mobots_navegacion.pdf.

R. Igual y C. Medrano. 2007. “Tutorial de OpenCV”. http://josbram.delifrut.cl/files/openCV/tutorial_opencv.pdf.

R. Szeliski. 2011. “Computer Vision. Algorithms and Applications”. Springer. http://szeliski.org/Book/drafts/SzeliskiBook_20100903_draft.pdf.

"Realidad Aumentada (1/7) | Inmensia". 2016. Inmensia.Com. http://inmensia.com/blog/20110523/realidad_aumentada_introduccion.html.

"Realidad Aumentada (Opencv+Aruco)". 2016. *Acodigo.Blogspot.Com*. http://acodigo.blogspot.com/2016/02/realidad-aumentada-opencvaruco.html.

"Tutorprogramacion/Tutoriales-Opencv". 2016. *Github*. https://github.com/TutorProgramacion/tutoriales-opencv.

ANEXOS

ANEXO I

INSTALACIÓN DE OPENCV Y ARUCO EN LINUX

La instalación de OpenCV y las librerías Aruco en Linux, es un proceso bastante más sencillo que en Windows, por lo que no se detallara en este trabajo.

La utilización de Opencv y las librerías con Eclipse, se realiza de la misma forma en Linux que en Windows por lo que su instalación de describirá con detalle en el siguiente apartado, instalación de OpenCV y ArUco en Windows.

INSTALACIÓN DE OPENCV Y ARUCO EN WINDOWS

El proceso de instalación de OpenCV con Eclipse en Windows es bastante complejo, a continuación se detallan los pasos a realizar para instalarlo en un ordenador con Windows 10 y un sistema operativo de 64 bits, con un procesador x64.

Lo primero que hay que hacer es descargar los siguientes archivos

Nombre	Fecha de modifica...	Tipo	Tamaño
cmake-3.4.3-win32-x86.exe	01/05/2016 14:25	Aplicación	12.812 KB
codelite-5.1.0-mingw4.7.1.exe	01/05/2016 14:28	Aplicación	34.623 KB
eclipse-cpp-mars-2-win32.zip	01/05/2016 16:56	Carpeta comprimi...	180.746 KB
opencv-2.4.9.exe	01/05/2016 14:25	Aplicación	357.083 KB

Figura A1: Archivos descargados

Para descargar OpenCV se va a la página oficial http://opencv.org/-->DOWNLOADS y desde allí se selecciona la versión adecuada para el sistema operativo, en este caso la versión 2.4.9 para Windows.

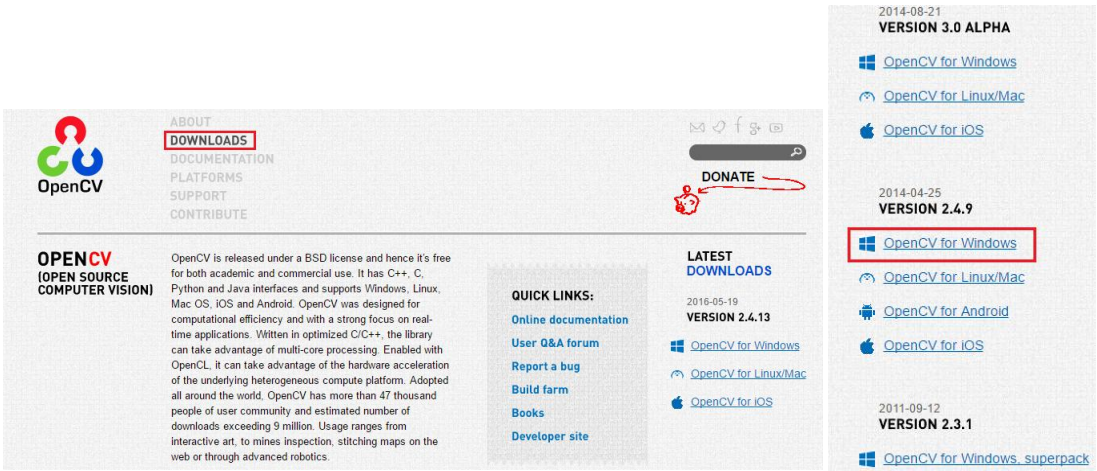


Figura A2: Descarga de OpenCV

Una vez descargado OpenCV, se procede a descargar CMake, para ello, desde la página https://cmake.org/ → Download se selecciona el archivo apto, en este caso cmake-3.4.3-win32-86.exe (Este nombre no corresponde con el de la imagen debido a que es una versión más antigua, pero el modo de instalación es el mismo).

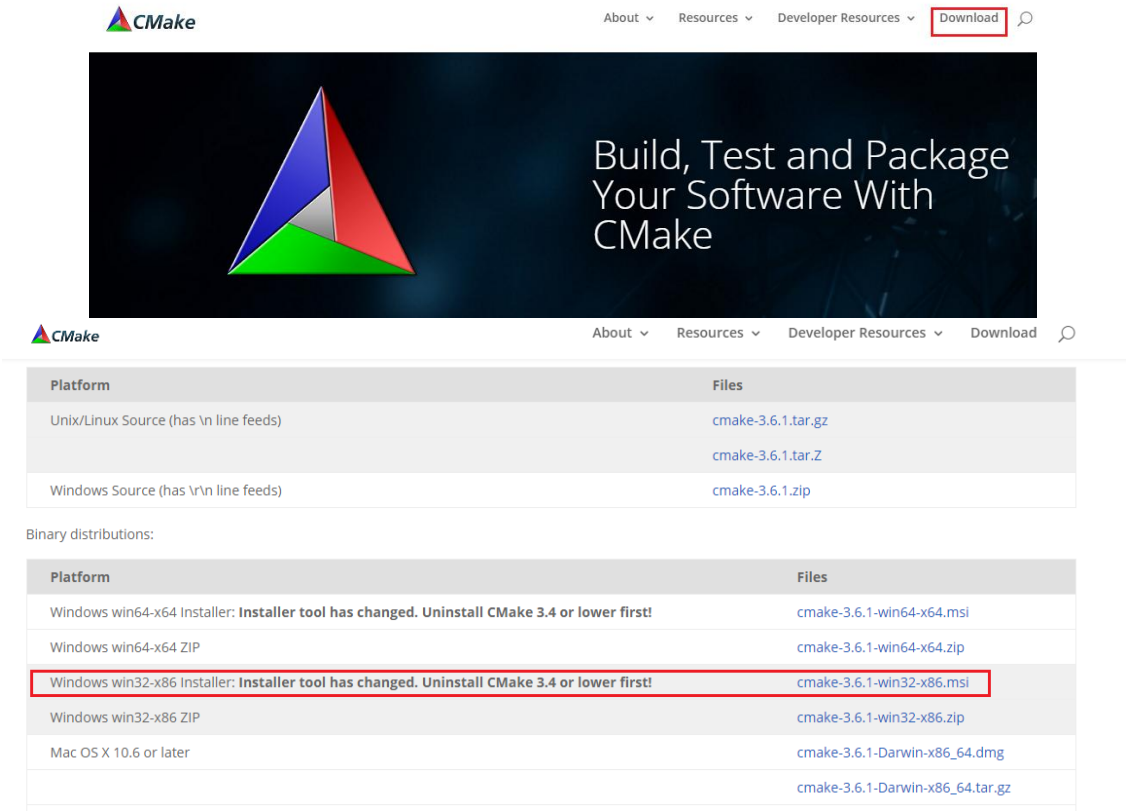


Figura A3: Descarga de CMake

El siguiente paso es descargar CodeLite, a través de la página de descargas SourceForge

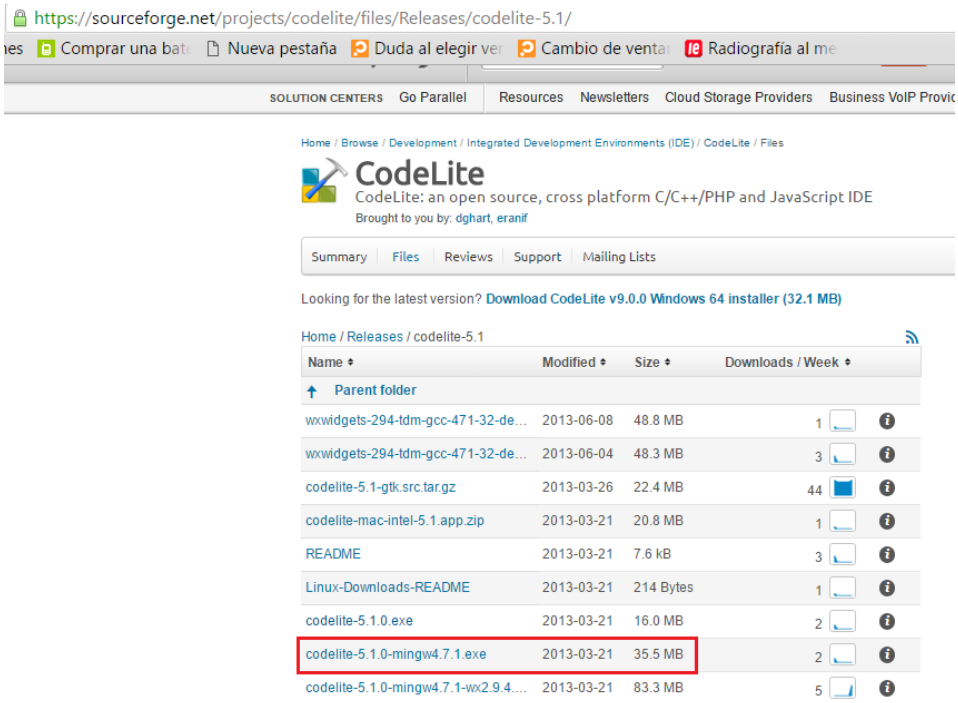


Figura A4: Descarga de CodeLite

Y por último se descarga Eclipse. Se entrar en la página https://eclipse.org/ → DOWNLOAD → Download Packages y descargar el Eclipse IDE for C/C++ Developers.

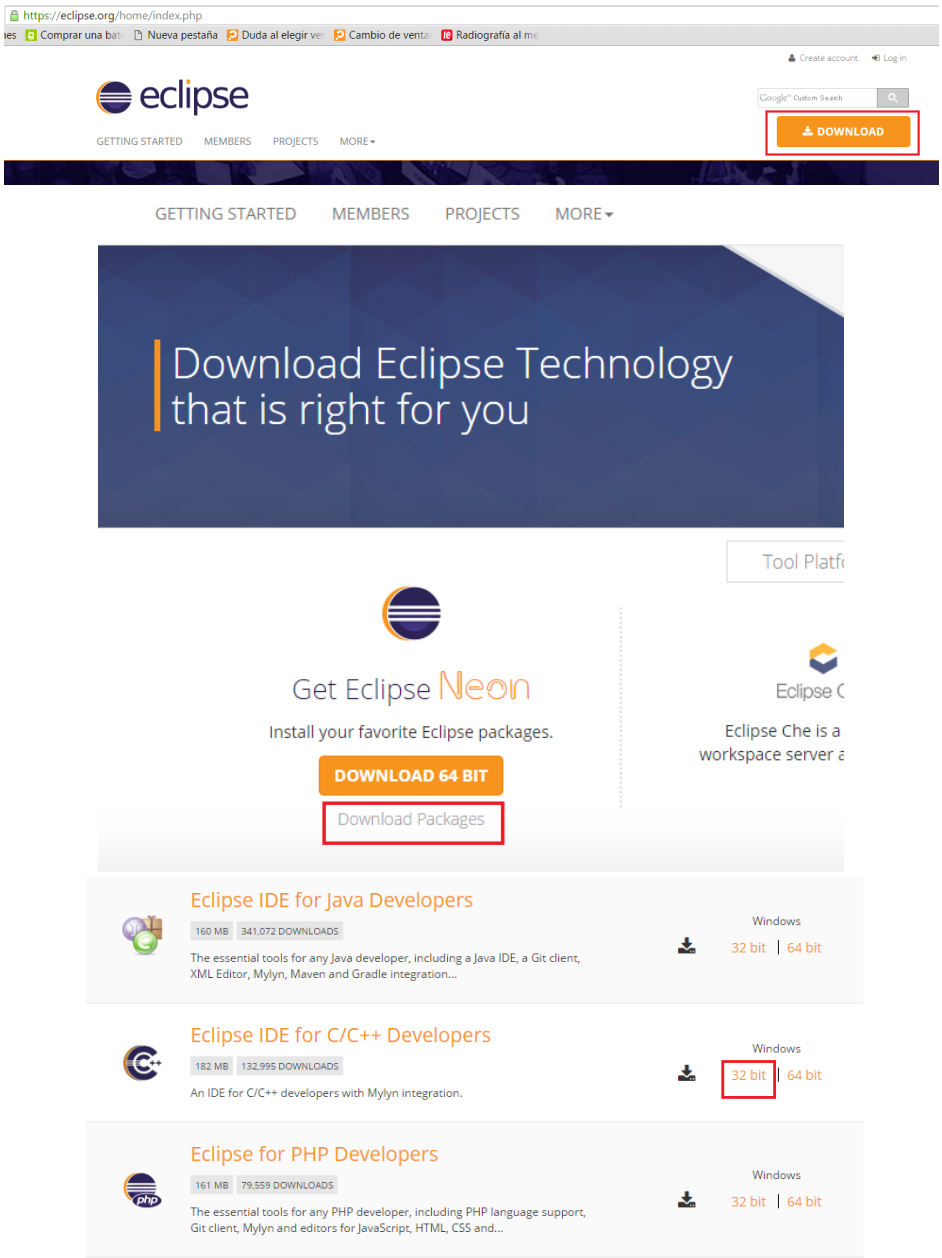


Figura A5: Descarga de Eclipse

Una vez se tenga todo descargado, se procede a la instalación, primero se instalará OpenCV, para ello se ejecuta la aplicación y se extrae en la carpeta por defecto. Cuando haya acabado la extracción, se ejecuta el codelite y se seleccionan las siguientes opciones:

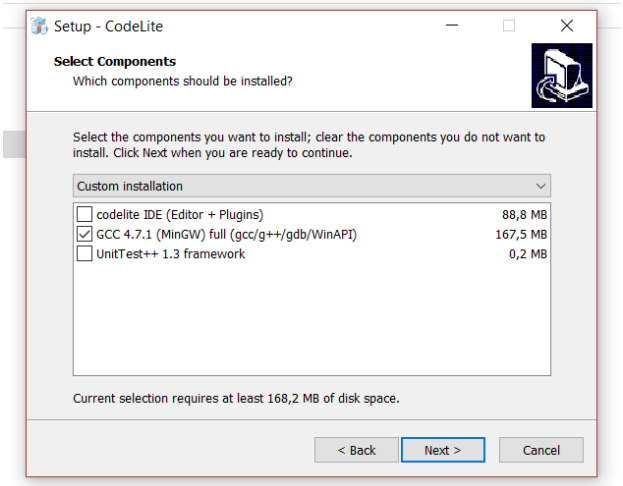


Figura A6: Instalación de CodeLite

Y las demás se dejan por defecto. Una vez acabada la instalación, se va a la carpeta donde se ha instalado el MinGW y de ahí a la carpeta bin

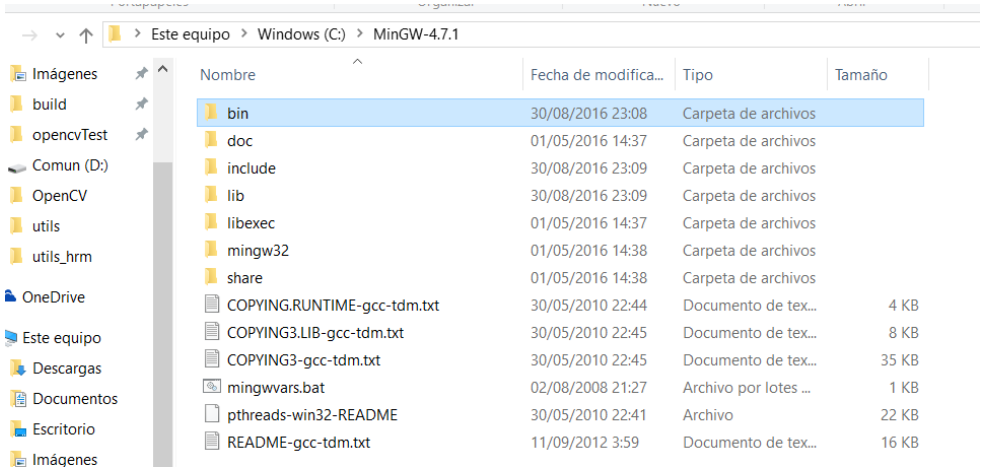


Figura A7: Carpeta bin de MinGW

Una vez estemos en ese carpeta, se copia la ruta, en este caso C:\MinGW-4.7.1\bin, y se va al editor de variables del entorno del sistema

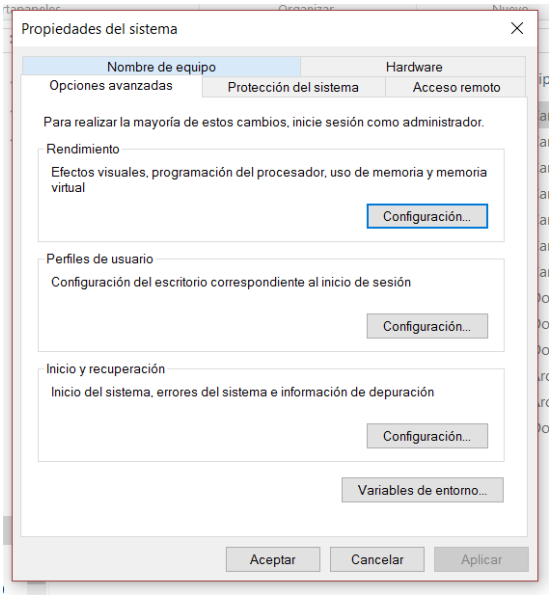


Figura A8: Editor de variables del sistema 1

Variables del entorno → Variables del sistema → Path → Editar → Nuevo → se copia la ruta → Aceptar

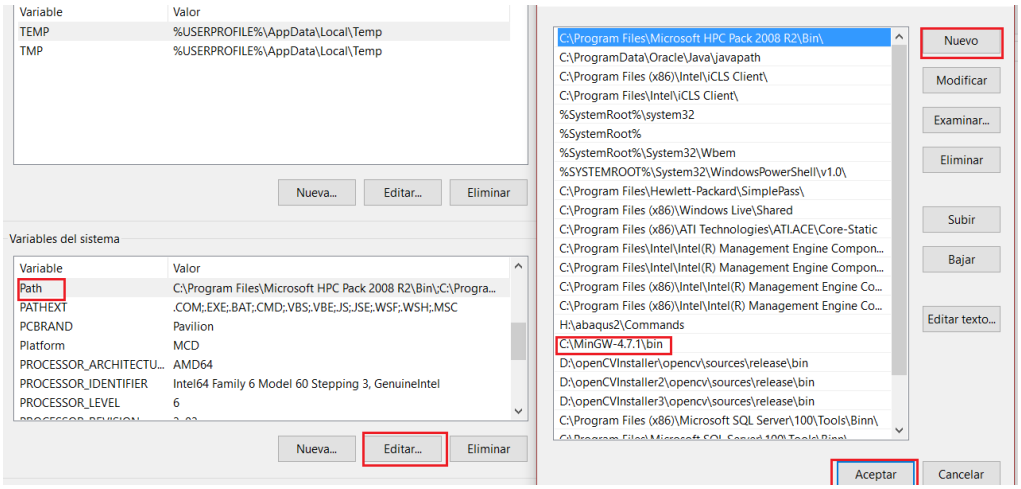


Figura A9: Editor de variables del sistema 2

Posteriormente se ejecuta el CMake se selecciona la siguiente opción

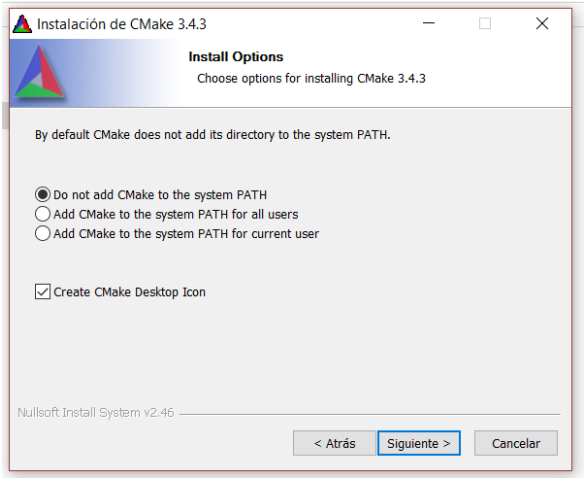


Figura A10: Instalación de CMake

Y el resto se dejan por defecto. Una vez instalado se abre el programa y en el apartado source code, se escribe la ruta donde se encuentra la carpeta sources, ubicada dentro de la carpeta opencv extraída anteriormente. En el apartado build the binaries, se selecciona el directorio donde está la carpeta release, la cual no existe y tiene que ser creada por el usuario dentro de la carpeta sources

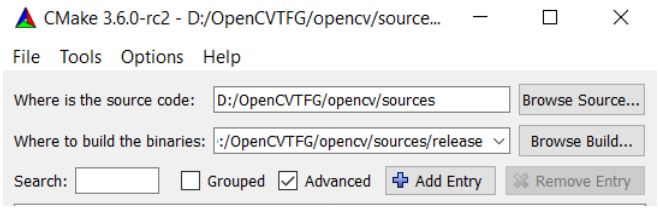


Figura A11: Extracción de los binarios de OpenCV 1

Una vez realizados estos pasos se selecciona configure y los siguientes parámetros

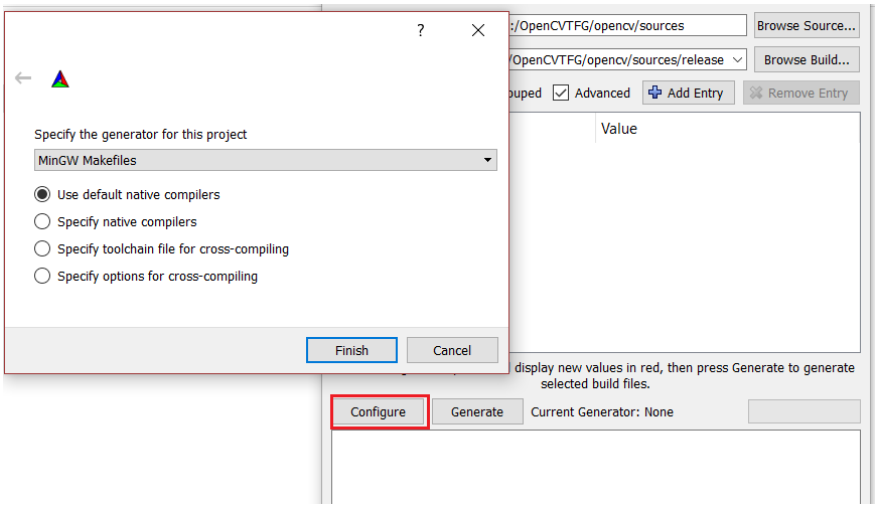


Figura A12: Extracción de los binarios de OpenCV 2

Cuando esta operación ha finalizado, en el apartado Search se escribe exam y se seleccionan las dos primeras opciones y se le vuelva a dar a configure

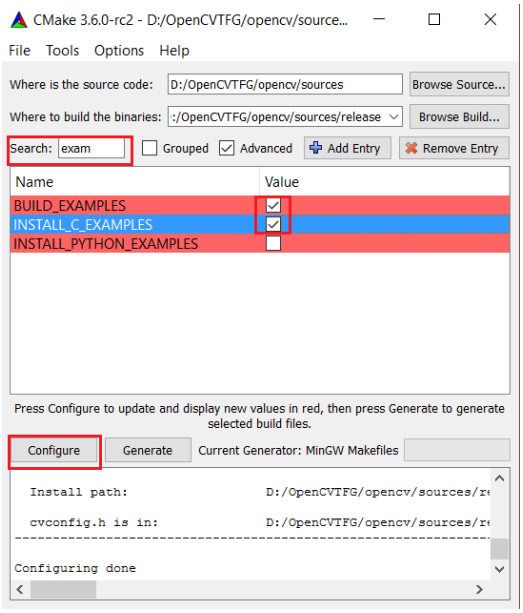


Figura A13: Extracción de los binarios de OpenCV 3

Se borra la palabra exam y se le vuelve a dar a configure

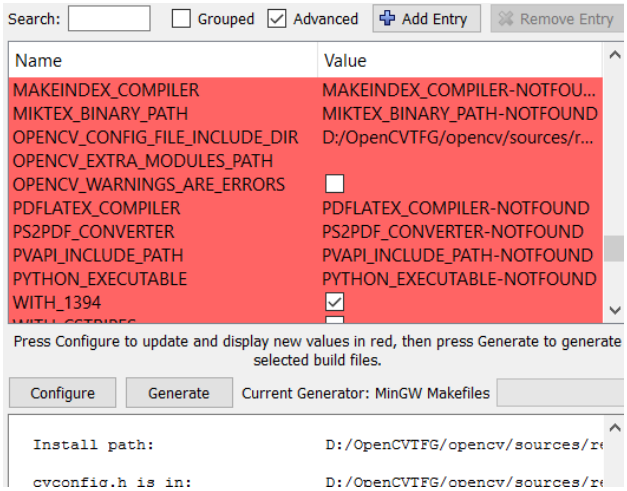


Figura A14: Extracción de los binarios de OpenCV 4

Y por último se selecciona generate.

Cuando todos estos pasos se han realizado satisfactoriamente se va a la carpeta release creada y desde allí se abre el símbolo del sistema y se ejecuta la opción mingw32-make

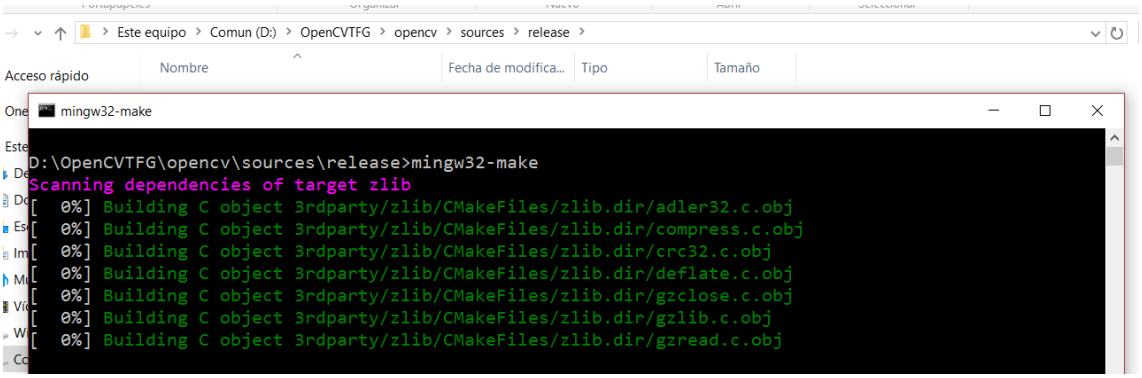


Figura A15: Instalación de OpenCV 1

Cuando este proceso ha acabado, se procede a realizar la instalación ejecutando mingw32-make install

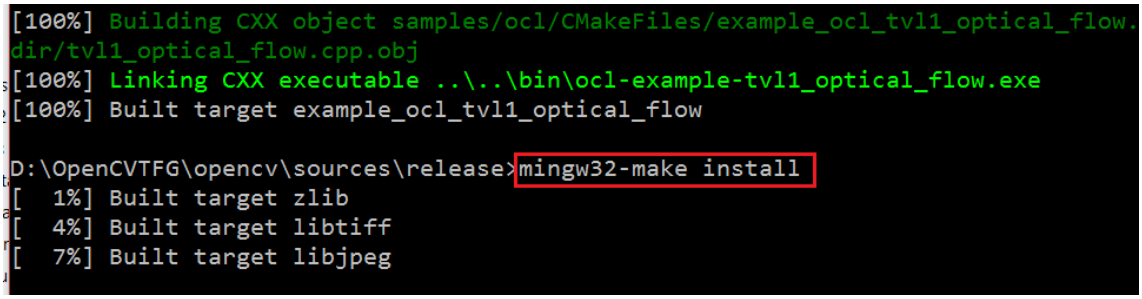


Figura A16: Instalación de OpenCV 2

Una vez acabada la instalación, dentro de la carpeta release, se va a la carpeta bin, y como se ha hecho anteriormente se copia la ruta de la carpeta en el Path de las variables del sistema

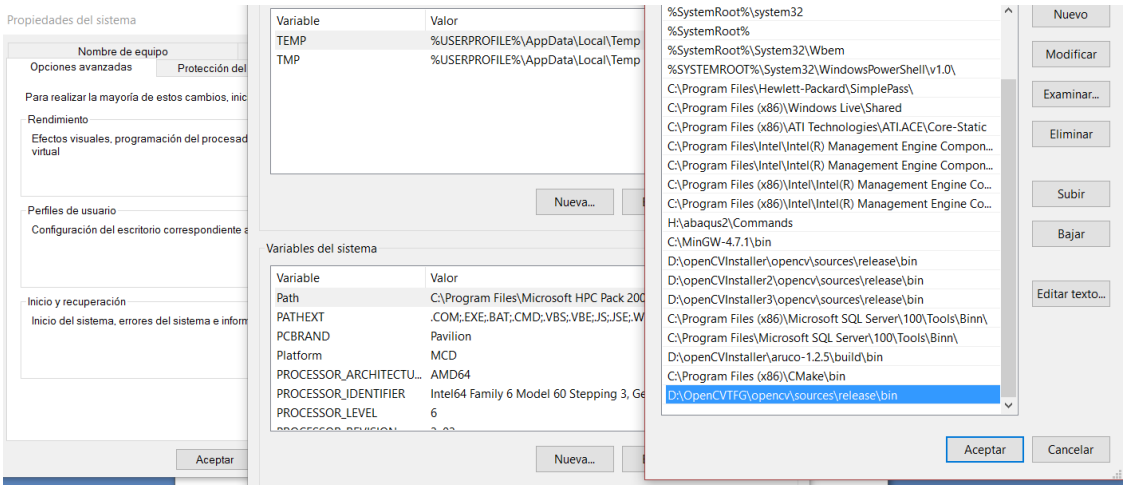


Figura A17: Variables del sistema 3

Una vez realizada esta operación, se extrae la carpeta de Eclipse descargada anteriormente

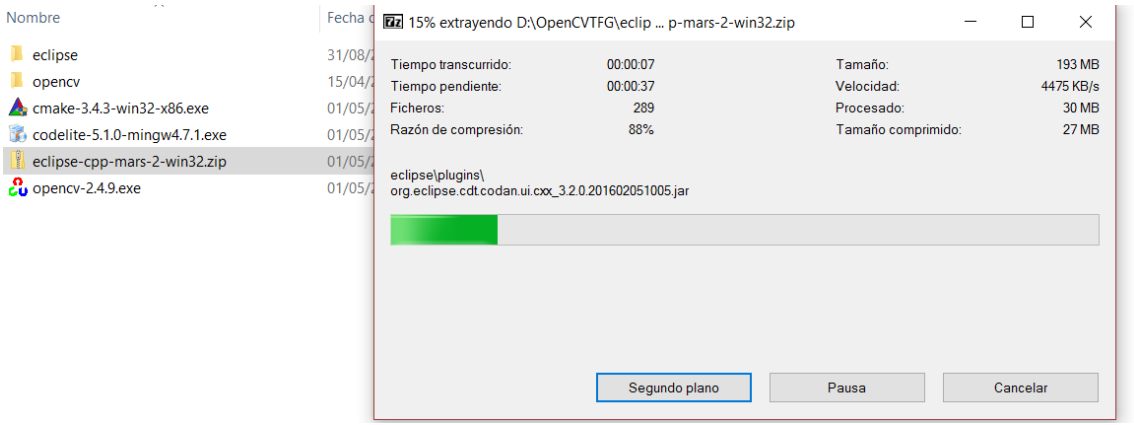


Figura A18: Extracción de Eclipse

A continuación, dentro de la carpeta extraída, se ejecuta la aplicación de eclipse y se establece la ruta donde se guardaran los trabajos realizados

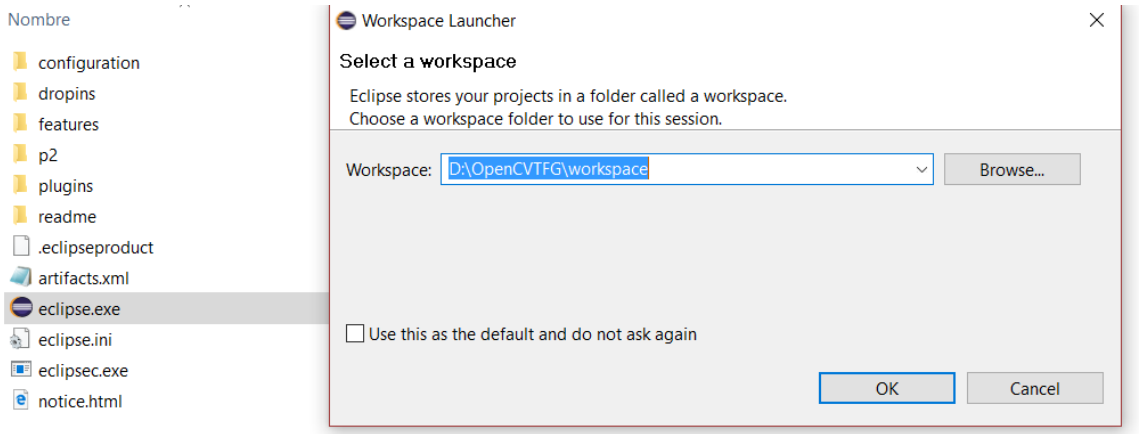


Figura A19: Ejecución de Eclipse

Antes de añadir los Paths y las librerías de OpenCV en Eclipse, se va a descargar la librería ArUco para ahorrar pasos. Para descargarla, se va a la página de descargas SourceForge y se descargar la versión deseada. Si es posible, descargar la última versión, ya que contiene más funciones lo que facilita la posibilidad de realizar programas mejores. En este caso se ha descargado la versión 1.2.5 ya que era la máxima compatible.

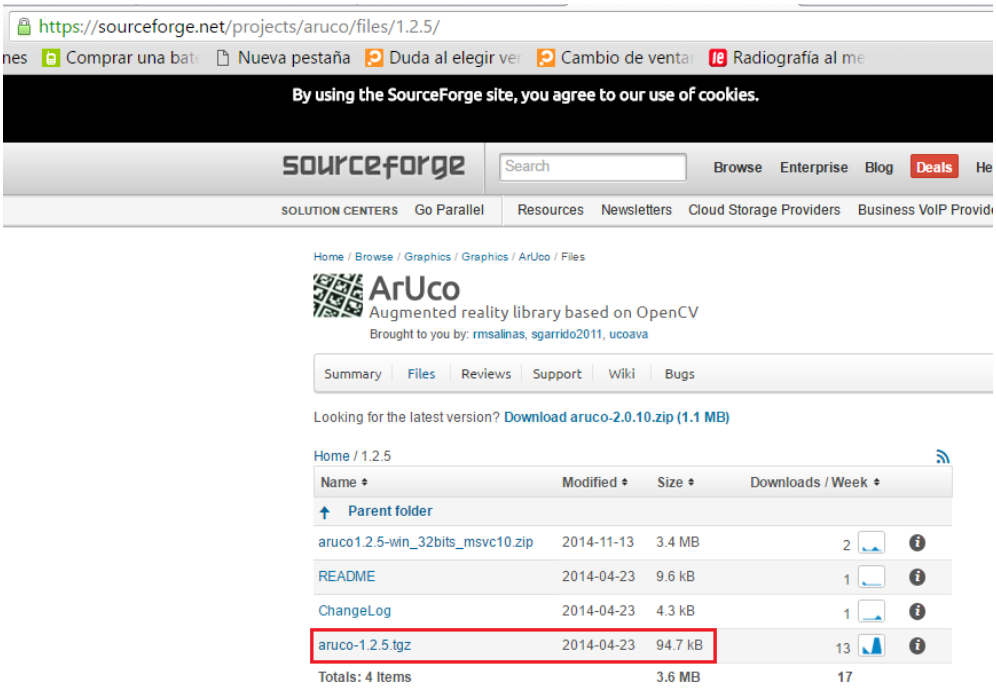


Figura A20: Descarga de ArUco

Cuando este descargada, habrá que descomprimirla dos veces

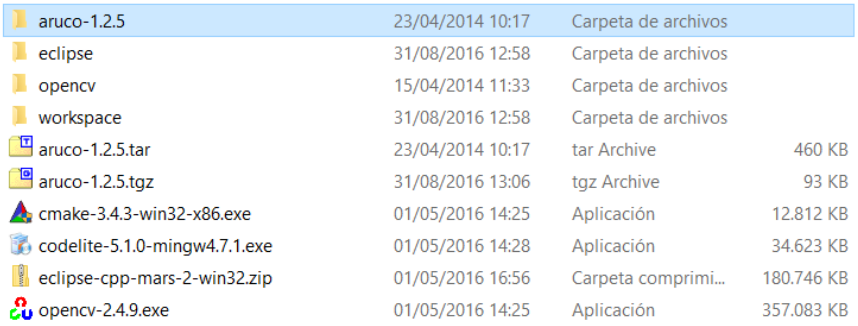


Figura A21: Librería ArUco descomprimida

Una vez descomprimida, se abre la carpeta y se crea una nueva carpeta, a la que se llamara build. Y, como en el caso de OpenCV, habrá que utilizar el codeblocks y MinGW para compilar. Se abre CMake se copian las ubicaciones de las carpetas donde se encuentra el código y donde se generaran los binarios → Configure → se seleccionan las opciones mostradas en la Figura A22→ Generate

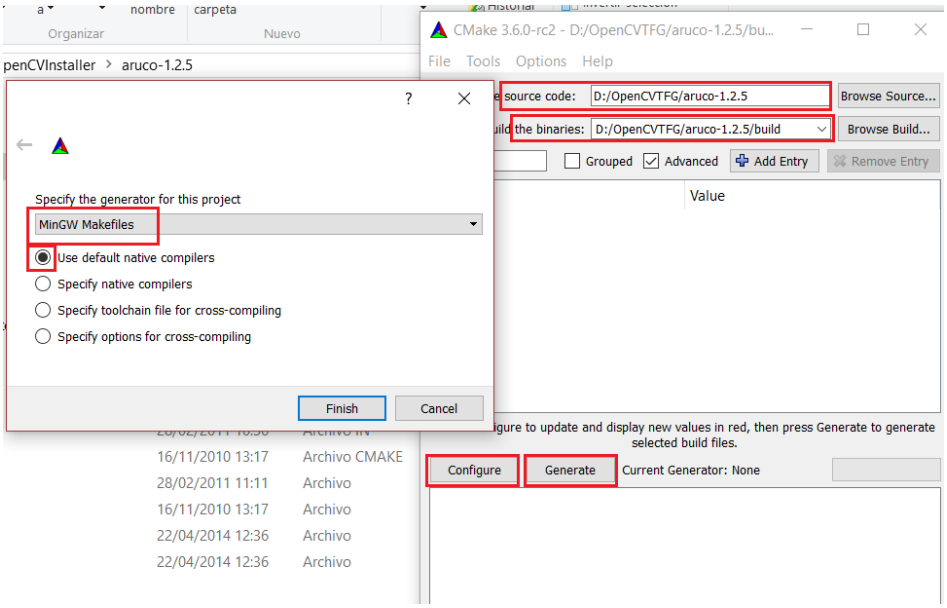


Figura A22: Extracción de los binarios de la librería ArUco

Una vez generados los binarios, se abren los símbolos del sistema, desde la carpeta build, y se ejecutan las órdenes mingw32-make y mingw32-make install, como se ha descrito anteriormente.

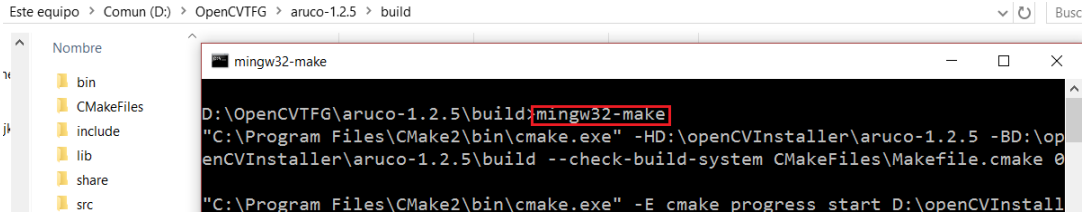


Figura A23: Instalación de la librería ArUco 1

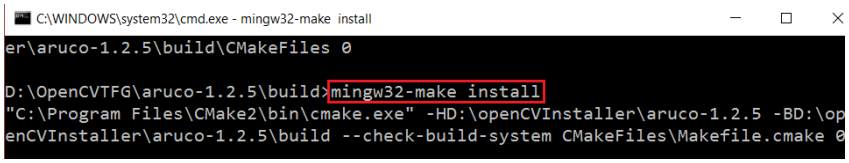


Figura A24: Instalación de la librería ArUco 2

Se va a las variables del sistema, y se copia en el Path la ruta donde se encuentra la carpeta bin

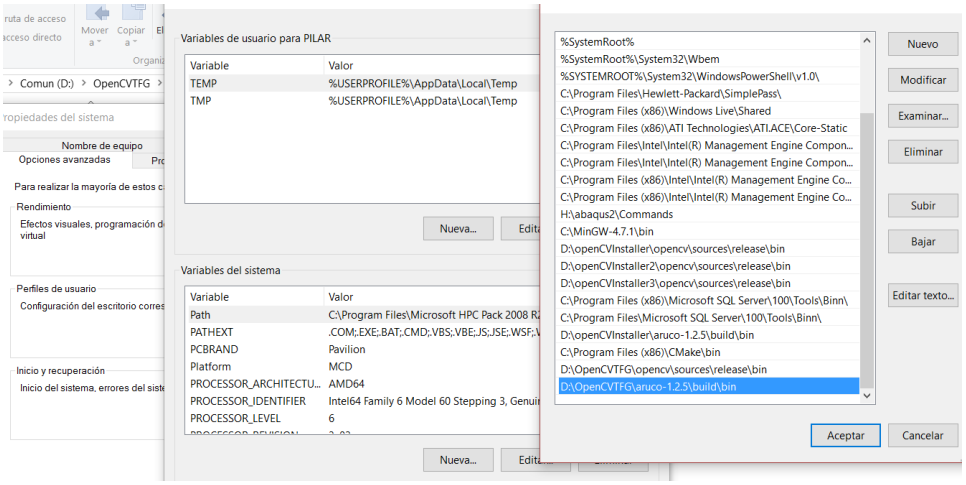


Figura A25: Variables del sistema 4

Después de tener todo instalado, se abre Eclipse → File → New → C ++ Proyect

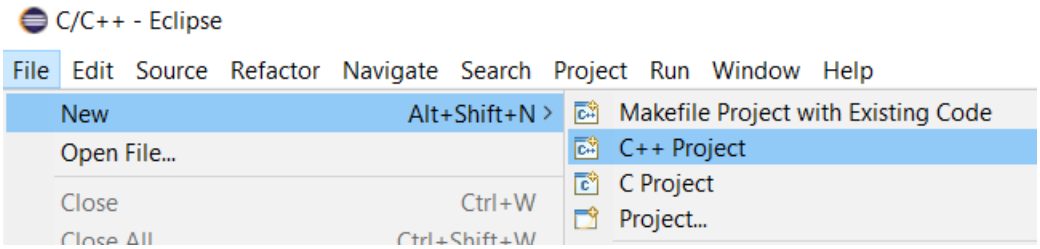


Figura A26: Crear un proyecto nuevo en Eclipse 1

Se da el nombre que se desee al proyecto → se seleccionan las opciones de la Figura A27 → Next

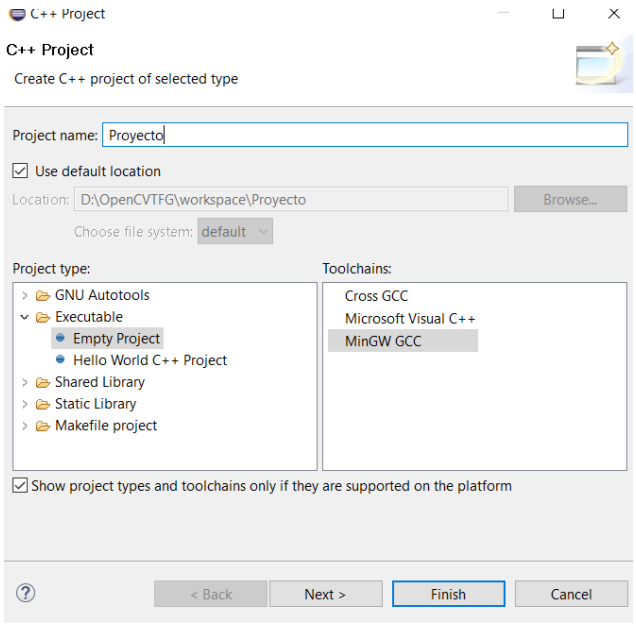


Figura A27: Crear un proyecto nuevo en Eclipse 2

Se seleccionan las opciones mostradas en la Figura A28 → Finish

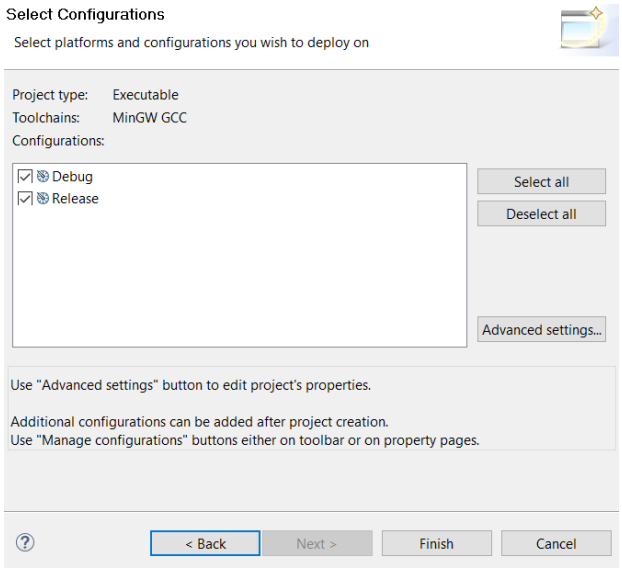


Figura A28: Crear un proyecto nuevo en Eclipse 3

Proyecto (botón derecho) →Build Configurations → Set Active → Release

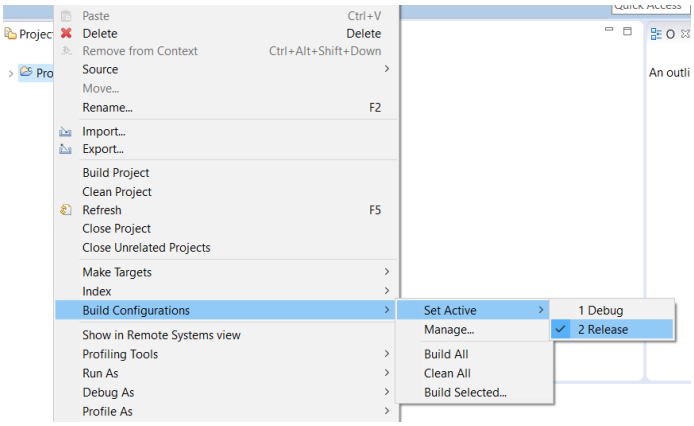


Figura A29: Crear un proyecto nuevo en Eclipse 4

Proyecto (botón derecho) → Properties → C/C++ Build → Settings → Configuration: Release → GCC C++ Compliler → Includes → Add → y se seleccionan los paths, tanto de OpenCV como de ArUco → OK

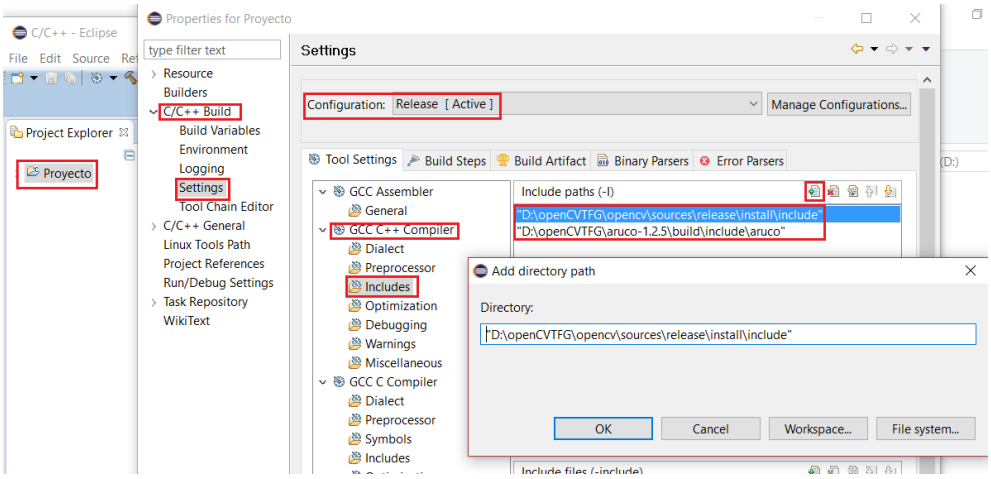


Figura A30: Compilar las librerías de OpenCV y de ArUco en Eclipse 1

GCC C Compiler → Includes → Add → Se copian los mismos paths del apartado anterior → OK

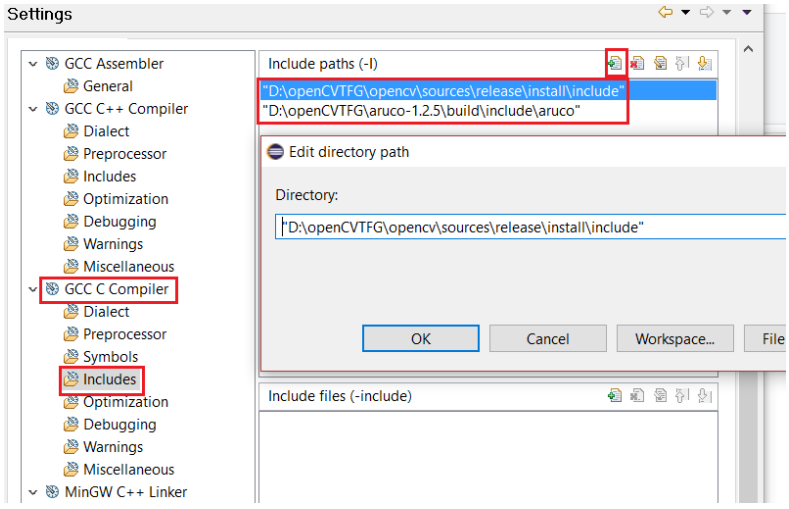


Figura A31: Compilar las librerías de OpenCV y de ArUco en Eclipse 2

MinGW C++ Linker → Libraries → Añadir las librerías y los paths donde se encuentran

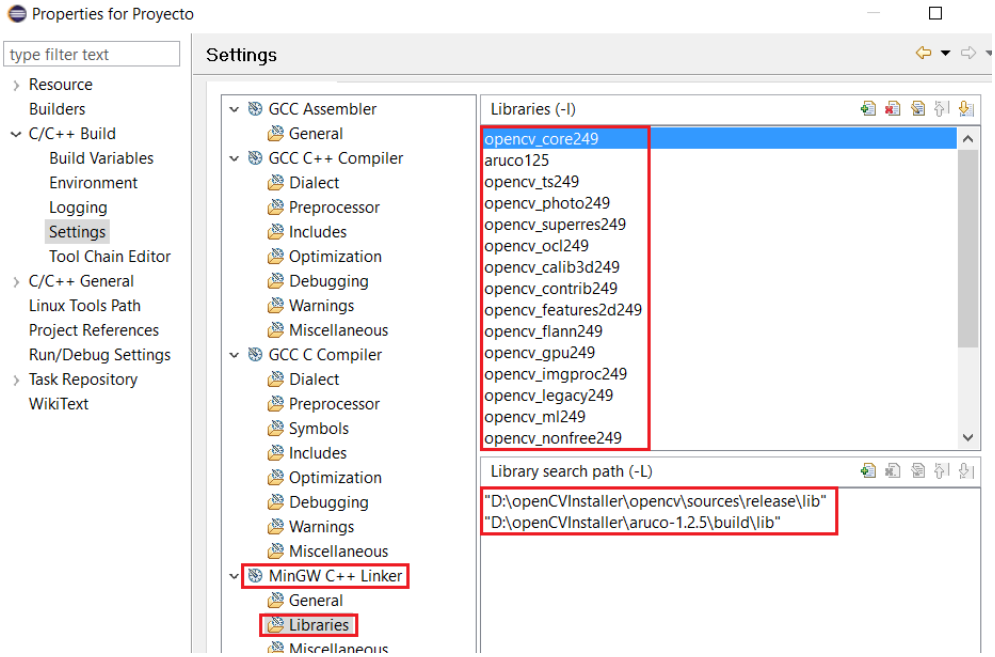


Figura A32: Compilar las librerías de OpenCV y de ArUco en Eclipse 3

Apply → OK

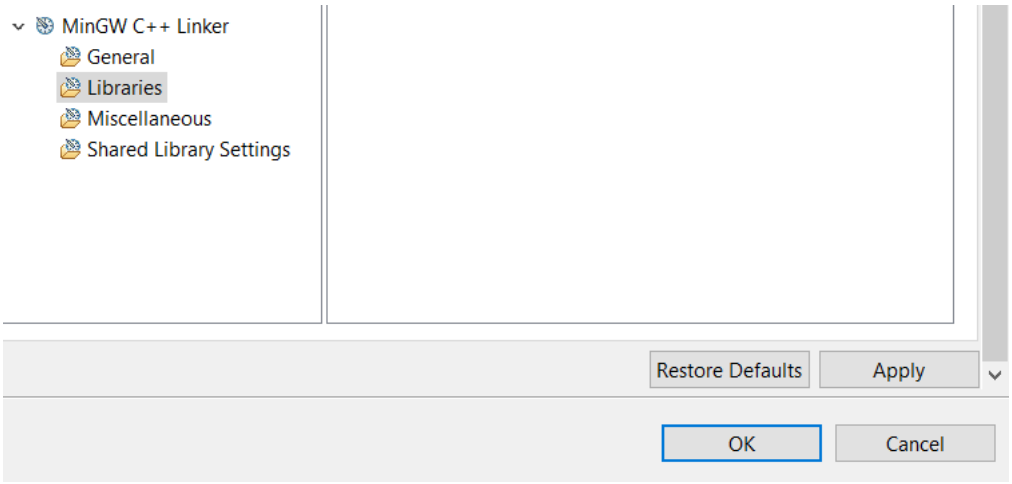


Figura A33: Compilar las librerías de OpenCV y de ArUco en Eclipse 4

Ir a Configuration: Debug y repetir todos los pasos usados para Release

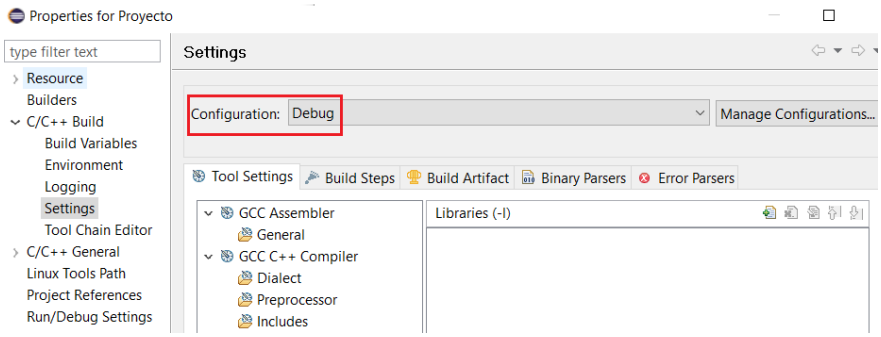


Figura A34: Compilar las librerías de OpenCV y de ArUco en Eclipse 5

Una vez realizado este proceso ya se puede proceder a realizar programas, para ello se va a Proyecto (botón derecho) → New → Source File → nombre_del_programa.cpp → Finish

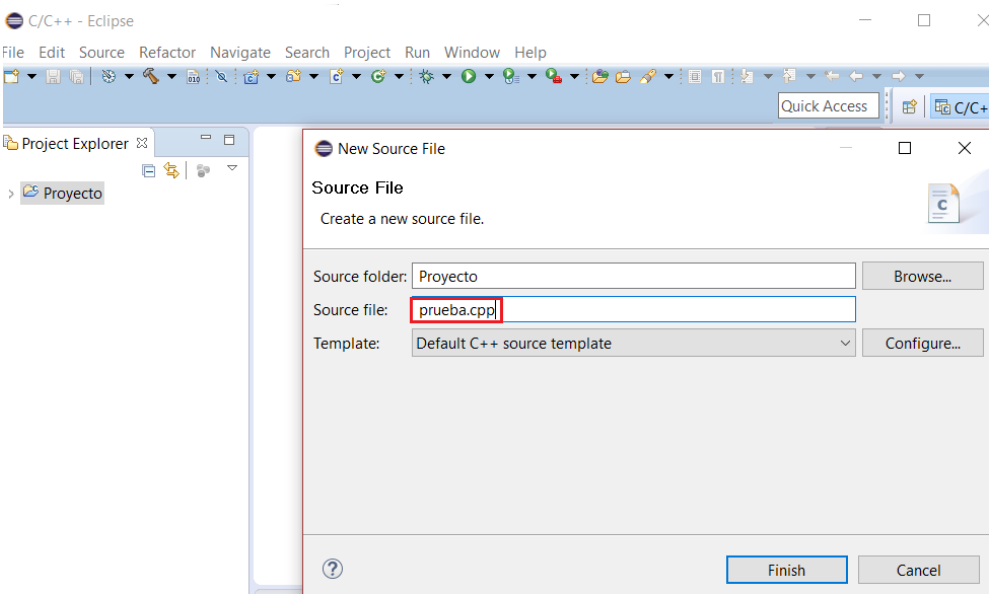


Figura A35: Crear un nuevo programa en Eclipse 1

Conviene crear otra carpeta, a parte de la de Proyecto, para guardar allí los programas que no se vayan a usar.

En los includes relacionados con la librería ArUco, habrá q poner la ruta completa donde se encuentra la carpeta

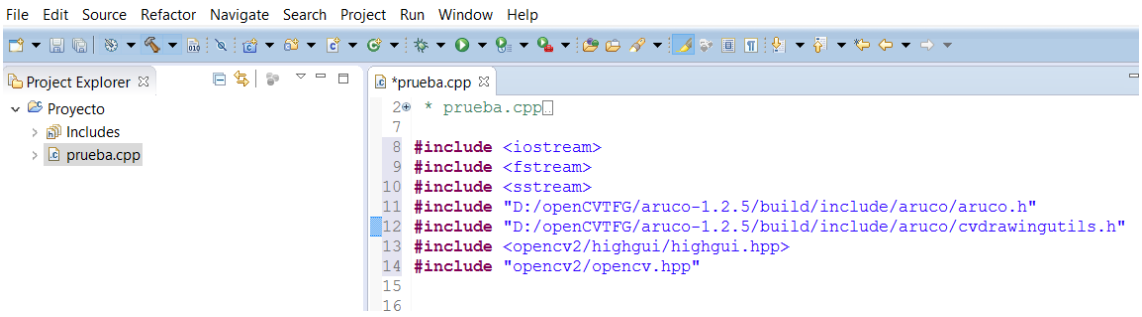


Figura A36: Crear un nuevo programa en Eclipse 2

Y ya se puede proceder a realizar o compilar los programas deseados.

ANEXO II

CREAR TABLERO

El programa crear_talero.cpp es utilizado para crear el tablero que posteriormente se utilizara para calibrar la cámara. Este programa crea dos archivos, un archivo imagen del tablero creado y un archivo de texto con los datos del tablero, tales como número de marcadores, id de los marcadores y localización de las esquinas de los marcadores en píxeles.

Para ejecutar este programa hay que introducir una serie de valores, primero hay que decir el número de marcadores que va a haber en el eje X y en el eje Y, modificando los valores de `XSize` y de `YSize`. Mediante la constante `pixSize` se establece el tamaño de los marcadores en pixeles. Con `interMarkerDistance` se da el valor de la distancia entre los marcadores y con `typeBoard`, el tipo de tablero deseado. Para crear e tablero hay 3 opciones distintas, si `typeBoard` =0, se creara un panel (Figura 7), si vale 1 se hará un tablero de ajedrez (Figura 8), en este caso, los valores introducidos en `XSize` y `YSize` será en número de cuadriculas en el tablero y no el numero de marcadores, y si vale 2, se obtendrá un marco (Figura 9).

Una vez establecidas las constantes para crear el tablero, hay que guardar los resultados obtenidos, para ello hay dos funciones, **`imwrite`** y **`saveToFile`**. Mediante las constantes char `name1` y `name2` se ha establecido la ruta, el nombre y el tipo de archivo a guardar. En este caso se ha seleccionado la ruta por defecto, que es la carpeta donde se está trabajando y se ha guardado un archivo .jpg con el nombre tablero. Lo mismo ocurre con el archivo de texto, se selecciona la ruta por defecto con el nombre tablero.yml.

```
int XSize=6; //Marcadores en el eje X ( Si se selecciona el tablero de
ajedrez, en vez de marcadores son cuadrados)
int YSize=6; //Marcadores en el eje Y ( Si se selecciona el tablero de
ajedrez, en vez de marcadores son cuadrados)
int pixSize=100;    //Tamaño de marcadores en pixeles
float interMarkerDistance=0.2;   //Distancia entre los marcadores [0,1]
int typeBoard=1;    //typeBoard(0: panel,1: tablero de ajedrez, 2: marco)
char name1[]="tablero.jpg";
char name2[]="tablero.yml";
```

ANEXO III

PASAR EL TABLERO CREADO EN PÍXELES A METROS

La función de este programa es pasar le información del tablero obtenida en el ANEXO II en pixeles a metros.

Para realizar este proceso, hay que imprimir el tablero.jpg obtenido en el ANEXO II y medir el tamaño de los marcadores en metros, en el caso de este trabajo, los marcadores miden 0.0262 metros. Esta información se escribe en la variable `markerSize_meters`.

El programa necesitara que se le introduzcan dos variables más, la primera es el archivo que contiene la información del tablero en píxeles y la segunda es el archivo donde se guardara esta misma información en metros. Las rutas de la ubicación de estos archivos es escriben en las variables `name1` y `name2` respectivamente.

```
float markerSize_meters=0.0262;          //Tamaño del marcador en metros
char name1[]="tablero.yml";              //Nombre y ruta del archivo a leer
char name2[]="tablero_metros.yml";       //Nombre y ruta del archivo a guardar
```

ANEXO IV

CALIBRACIÓN DE LA CÁMARA

Este programa se emplea para calibrar la cámara, hay que introducir el número de la cámara que se va a calibrar en `ncamara`, esto es debido a que puede haber varias cámaras disponibles en el momento de calibración.

El programa mostrará por pantalla lo que ve la cámara y, como se explica en el apartado de calibración de la cámara, habrá que ir moviendo el tablero para que el programa obtenga la información necesaria y pueda crear un archivo de texto que contenga la matriz de la cámara y los coeficientes de distorsión.

Para que esto sea posible, habrá que introducir en la variable `name1` la ubicación de archivo que contiene las coordenadas del tablero en metros y en la variable `name2` la ubicación del archivo con los resultados de la calibración.

```
int ncamara=0;
char name1[]="tablero_metros.yml";
char name2[]="parametros_camara.yml";
```

ANEXO V

CREAR MARCADOR

Este programa permite la creación de marcadores fiduciales, los cuales se usaran durante todo el trabajo tanto para localizar la pista como para localizar a los robots.

Para obtener el marcador, hay que introducir el numero de marcador que se desea crear en `maker` (el número de marcadores disponibles va de 0 a 1023), el tamaño del marcador en píxeles en `size`, y la ruta y el nombre donde se guardará el marcador en `name1`, en este caso se ha guardado en la ruta por defecto y con el nombre marcador31.png.

```
int maker=31;//Marcador [0,1023]
int size=500;//Tamaño del marcador en pixeles
char name1[]="marcador31.png"    //Ubicación donde se guardará el marcador
```

ANEXO VI

ORIENTACIÓN DE LOS MARCADORES

La función de este programa es conocer la orientación del marcador fiducial antes de colocárselo al robot para que cuando este se mueva por la pista, el ángulo orientación mostrada por pantalla sea la del robot.

El código para realizar esta operación se ha extraído casi en su totalidad del programa principal, pero se ha creído conveniente realizarlo, ya que facilita notablemente la correcta colocación de los marcadores en los robots.

Para ejecutar este programa hay que introducir la resolución de la cámara (`ancho, alto`), el número de cámara que se va a utilizar (`ncamara`) y la ubicación del archivo con los datos de la calibración (`name`).

No se considera necesario introducir más parámetros ya que la utilidad de este programa es meramente informativa.

```
int ancho = 640, alto = 480;  //Resolución de la cámara
unsigned int ncamara=0;       //Número de la cámara que se va a
                              utilizar
char name[]="parametros_camara.yml
```

ANEXO VII

LOCALIZACIÓN DE ROBOTS Y DETECCIÓN DE OBSTÁCULOS

El código creado se encuentra bajo licencia GPL, las condiciones para su uso, modificación o distribución se encuentran más detalladas en la cabecera del programa.

Para que el programa funcione correctamente, hay que introducir una serie de valores.

```
int ancho = 640, alto = 480;  //Resolución de la cámara en píxeles
char name[]="parametros_camara.yml";//Ubicación del archivo de
texto donde se encuentran los parámetros de calibración de la cámara
unsigned int ncamara=2; //Número de la cámara que se va a utilizar
int tiempocap=40; //Tiempo en milisegundos de la frecuencia con la que
se ejecuta el programa
float anchop=1.255;     // Ancho de la pista en metros
float largop=1.850;     //Largo de la pista en metros
int nrobots=2;          //Número de robots en la pista
float TamanioMarcador=0.132;//Longitud del lado del marcador en metros
float radioRobot=0.093; //Radio del robot móvil en metros
float distmar=0.300;    //Distancia del extremo de la pista al inicio
del marcador elevado en metros
```

En las variables `ancho` y `alto` hay que introducir la resolución de la cámara en píxeles, en la variable `name`, la ubicación y el nombre del archivo donde se encuentran los parámetros obtenidos al calibrar la cámara. En `ncamara` y `tiempocap` definimos el número de la cámara que se va a utilizar y el tiempo mínimo en milisegundos que tardará el programa en procesar una imagen. Las variables `largop` y `anchop` se utilizan para definir las dimensiones de la pista, corresponden a las coordenadas L y A de la Figura A37 y la variable `distmar` corresponde a la coordenada D de la Figura A37, todas ellas en metros. Con `nrobots` se define el número de robots que interactuarán en la pista y con `TamanioMarcador`, la longitud del lado de los marcadores fiduciales utilizados en metros. También habrá que introducir el valor del radio de los robots móviles en metros en la variable `radioRobot`.

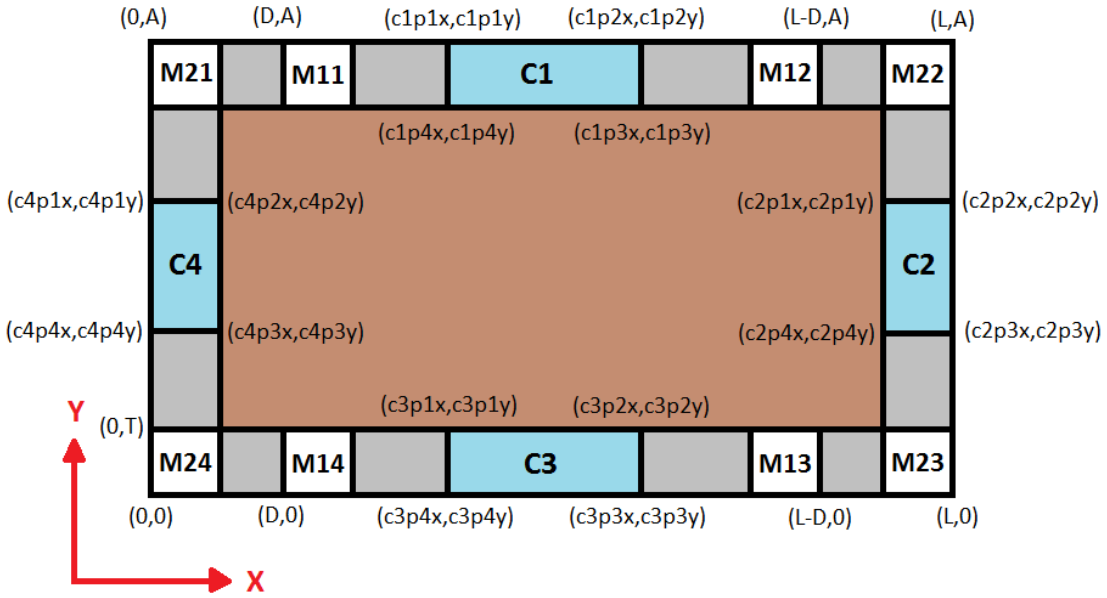


Figura A37: Diseño de la plataforma multi-robot

Aparte de estos valores, hay q introducir las coordenadas de los colores de referencia (Figura A37). Se han utilizado 4 marcadores de color, siendo el numero ubicado después de la palabra color el que indica a que color de referencia se está refiriendo, y el número detrás de p, el indicador de la esquina del marcador de color, por lo que la variable color1p2 indicaría la esquina 2 del color de referencia número 1 (Figura A38).

```
//Introducir las coordenadas (x,y) del color de referencia 1
Point2f color1p1(0.84,1.23);
Point2f color1p2(1.04,1.23);
Point2f color1p3(1.04,1.145);
Point2f color1p4(0.84,1.145);

//Introducir las coordenadas (x,y) del color de referencia 2
Point2f color2p1(1.74,0.725);
Point2f color2p2(1.825,0.725);
Point2f color2p3(1.825,0.525);
Point2f color2p4(1.74,0.525);


//Introducir las coordenadas (x,y) del color de referencia 3
Point2f color3p1(0.885,0.115);
Point2f color3p2(1.08,0.115);
Point2f color3p3(1.08,0.016);
Point2f color3p4(0.885,0.016);

//Introducir las coordenadas (x,y) del color de referencia 4
Point2f color4p1(0.02,0.745);
Point2f color4p2(0.11,0.745);
Point2f color4p3(0.11,0.545);
Point2f color4p4(0.02,0.545);
```

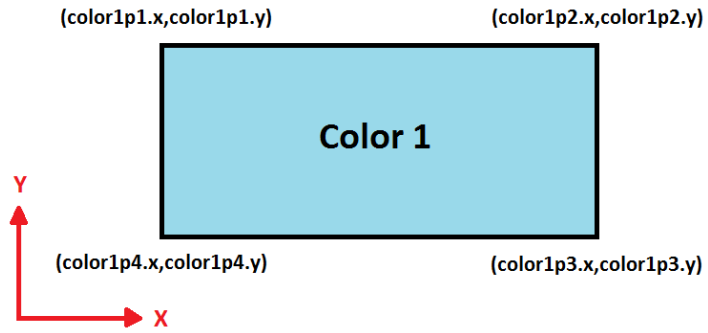


Figura A38: Coordenadas del color de referencia

Y por último, hay serie de valores utilizados para ajustar los parámetros en la detección de esquinas que no se recomienda modificar, salvo que se quiera comprobar diferentes situaciones en las que el obstáculo tiene demasiadas esquinas, están muy juntas,…

```
//Parámetros para la detección de esquinas
double qualityLevel = 0.3;    //Calidad mínima aceptada de las esquinas
                              de la imagen
double minDistance = 3; //Distancia euclídea mínima entre las esquinas
int maxCorners = 10;    //Número máximo de esquinas para encontrar.
                        //Si se encuentran más esquinas, las más
                        fuertes son las que se muestran
```

Con las datos para configurar la cámara y la información para reconocer la pista introducida correctamente, se ejecuta el programa y partir de aquí, el software realiza las siguientes operaciones.

Con las variables `ancho` y `alto`, mostradas anteriormente, definen la resolución que tendrá la cámara. Para este trabajo se ha utilizado una resolución 640x480. Se intentó emplear una calidad mayor pero el video a tiempo real iba retrasado por lo que no era una opción válida. La resolución empleada, aunque no sea de una calidad excesivamente buena, es suficiente para detectar los marcadores y permite capturar imágenes a más velocidad. Algo realmente útil si se quiere conocer la localización de un robot que se encuentra en movimiento.

Insertando el alto y ancho de la imagen se consigue configurar la resolución de la cámara.

```
Capturadevideo.open(ncamara);
Capturadevideo.set(CV_CAP_PROP_FRAME_WIDTH, ancho);
Capturadevideo.set(CV_CAP_PROP_FRAME_HEIGHT, alto);
```

Una vez introducida la resolución, el programa tiene que leer los datos obtenidos en la calibración.

```
ParametrosdelaCamara.readFromXMLFile(name);
```

Esta función se encarga de leer el archivo generado durante la calibración, que contiene la matriz de la cámara y los coeficientes de distorsión obtenidos.

Cuando los parámetros de la calibración han sido leídos el programa tiene que realizar capturas a frecuencia de video.

```
do
{
      Capturadevideo.retrieve(Imagendistorsionada);
      undistort(Imagendistorsionada,Imagen,ParametrosdelaCamara.Camera
      Matrix,ParametrosdelaCamara.Distorsion);
      .
      .
      .
      key=cv::waitKey(tiempocap);
}while(key!=27 && Capturadevideo.grab());
```

Mientras la tecla ESC no sea pulsada, la cámara grabará lo que ve y realizará instantáneas cada "`tiempocap`" milésimas de segundo. El valor de la variable `tiempocap` habrá sido introducido con anterioridad. Las capturas realizadas serán guardadas en la matriz `Imagendistorsionada` y utilizando la función **`undistort`**, se aplican los parámetros obtenidos en la calibración a la imagen distorsionada y se obtiene la imagen corregida.

La variable `Imagen`, donde estará la plataforma con los marcadores, los robots y los obstáculos, se pasa por la función `MDetector.detect()` que se encarga de detectar los marcadores que se encuentran en la pista y posteriormente se crea una copia de la Imagen (`CopiaImagen`) para poder modificarla sin sobrescribir la original.

Para localizar los robots, primero hay que localizar los marcadores que se encuentran en la pista a la misma altura que los robots. El código siguiente colorea las aristas de los marcadores detectados anteriormente en rojo y muestra el identificador (Id) de cada marcador. También guarda en la variable centro, la posición del centro en píxeles del marcador detectado.

```
for (unsigned int i=0;i<Marcadores.size();i++) {
    Marcadores[i].draw(CopiaImagen,Scalar(0,0,255),1);
    centro=Marcadores[i].getCenter();
```

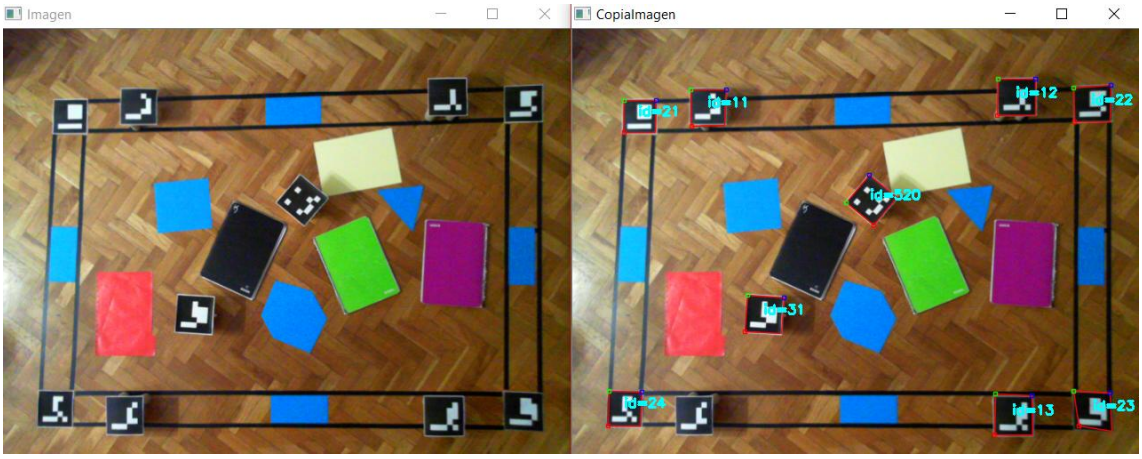


Figura A39: A la izquierda, la imagen de la pista y a la derecha, la misma imagen con los marcadores detectados.

Los marcadores situados a la altura de los robots tendrán unas Ids del 11 a 14, y los robots tendrán marcadores con Ids mayores que 30, por lo que se ha creado el siguiente código.

```
if (Marcadores[i].id==11)
      pixelR[0]=Point2f(centro);
if (Marcadores[i].id==12)
      pixelR[1]=Point2f(centro);
if (Marcadores[i].id==13)
      pixelR[2]=Point2f(centro);
if (Marcadores[i].id==14)
      pixelR[3]=Point2f(centro);
if ((Marcadores[i].id>30) and (Marcadores[i].id<31+nrobots))
      for (int cn=0;cn<(nrobots);cn++)
      {
            //Localización de los robots en píxeles
            Robot[cn]=Point2f(centro);

            //Obtención de la orientación del los robots
            punto1[cn]=Point2f(Marcadores[i][1].x,Marcadores[i][1].y);
            punto2[cn]=Point2f(Marcadores[i][2].x,Marcadores[i][2].y);
            punto[cn]=Point2f(((Marcadores[i][3].x+Marcadores[i][2].x)/
            2),((Marcadores[i][3].y+Marcadores[i][2].y)/2));
            cn++;
      }
```

La posición en pixeles del centro de los marcadores se guarda en un vector de 4 filas denominado `pixelR` y la posición en pixeles del robot se guarda en la variable `Robot`, cuyo número de filas dependerá del número de robots que haya en la pista. En los vectores `punto1` y `punto2` se almacenan la posición de dos de las esquinas de los marcadores, para posteriormente obtener e ángulo theta del robot. La variable `punto` se utilizará más adelante para dibujar la orientación del robot en la imagen.

Con la localización de los marcadores y de los robots en la imagen, se quiere obtener las coordenadas de los robots en metros en la pista, esto se consigue con la matriz de homografía. Para obtener dicha matriz, se ha creado el siguiente código.

```
mundoR[0]=Point2f(distmar-TamanioMarcador/2,
anchoreal+TamanioMarcador/2);
mundoR[1]=Point2f(largoreal-distmar+TamanioMarcador/2,
anchoreal+TamanioMarcador/2);
mundoR[2]=Point2f(largoreal-distmar+TamanioMarcador/2,0-
TamanioMarcador/2);
mundoR[3]=Point2f(distmar-TamanioMarcador/2,0-TamanioMarcador/2);
HR=findHomography(mundoR,pixelR);
invHR = HR.inv(DECOMP_LU);
```

El vector `mundoR` contiene las coordenadas de los marcadores en el mundo, mientras que el vector `pixelR` contiene dichas coordenadas en píxeles. Con estos dos vectores se obtiene la matriz de homografía `HR` y su inversa (`invHR`) con las que se pueden pasar puntos del mundo real a la imagen y viceversa.

Con la matriz `invHR` se va a calcular la localización de los robots y de los dos puntos obtenidos anteriormente en la pista.

```
for( int h=0;h<nrobots;h++)
{
	//Obtener la posición del robot en coordenadas mundo
	convertPointsToHomogeneous(Robot,RobotH); //Se convierte el
	vector 2D a la forma homogénea
	convertPointsToHomogeneous(punto1,punto1H);
	convertPointsToHomogeneous(punto2,punto2H);
	Mat RobotHT=Mat(RobotH[h]);   //Esquina homogénea y traspuesta
	Mat punto1HT=Mat(punto1H[h]);
	Mat punto2HT=Mat(punto2H[h]);
	Mat invHR2 = Mat::eye(3, 3, CV_32FC1);
	invHR.convertTo(invHR2, CV_32FC1);
	Mat rob=invHR2*RobotHT; //Se calculan las coordenadas reales del
	pixel
	Mat pun1=invHR2*punto1HT;
	Mat pun2=invHR2*punto2HT;
	vector<Point3f> robotvecH = Mat_ <Point3f>(rob);//Se convierte
	la matriz en un vector
	vector<Point3f> punto1vecH = Mat_ <Point3f>(pun1);
	vector<Point3f> punto2vecH = Mat_ <Point3f>(pun2);
	vector<Point2f> robotvec;
	vector<Point2f> punto1vec;
	vector<Point2f> punto2vec;
	convertPointsFromHomogeneous(robotvecH,robotvec);//Se convierte
	el vector homogéneo a un vector 2D
	convertPointsFromHomogeneous(punto1vecH,punto1vec);
	convertPointsFromHomogeneous(punto2vecH,punto2vec);
	Robotm[h]=Point2f(robotvec[0]); //A las coordenadas se la
	posición del robot, se añade su orientación
	punto1mundo[h]=Point2f(punto1vec[0]);
	punto2mundo[h]=Point2f(punto2vec[0]);
}
```

Como se muestra en las ecuación 2 la matriz homogénea es una matriz 3x3 y el vector en pixeles es un vector de 3 componentes, por lo que, mediante la función

convertPointsToHomogeneous, se obtiene, a partir de un vector 2D, el vector homogéneo. Con este vector ya transformado y la matriz inversa obtenida, se calcula el punto que ocupan las coordenadas en el mundo. Este punto se pasa a 2D con la función **convertPointsFromHomogeneous** y se obtiene el vector `Robotm`, que contiene la localización del robot en el mundo (x,y). El mismo proceso ocurre para las variables `punto1`y `punto2`.

Una vez obtenidas las coordenadas del robot en la pista, se procede a detectar los obstáculos. La primera parte de este proceso es parecida a la utilizada para localizar los robots. En el vector pixelO se guardará la localización en píxeles de los marcadores que se encuentran a ras de suelo, en este caso los que tienen ids de 21 a 24.

```
if (Marcadores[i].id==21)
      pixelO[0]=Point2f(centro);
if (Marcadores[i].id==22)
      pixelO[1]=Point2f(centro);
if (Marcadores[i].id==23)
      pixelO[2]=Point2f(centro);
if (Marcadores[i].id==24)
      pixelO[3]=Point2f(centro);
if (pixelO[0]!=Point2f(0,0)and pixelO[1]!=Point2f(0,0)and
pixelO[2]!=Point2f(0,0)and pixelO[3]!=Point2f(0,0))
{
      mundoO[0]=Point2f(0-TamanioMarcador/2,anchoreal+ T
      amanioMarcador/2);
      mundoO[1]=Point2f(largoreal+TamanioMarcador/2,anchoreal+TamanioM
      arcador/2);
      mundoO[2]=Point2f(largoreal+TamanioMarcador/2,0-
      TamanioMarcador/2);
      mundoO[3]=Point2f(0-TamanioMarcador/2,0-TamanioMarcador/2);
      HO=findHomography(mundoO,pixelO);
      invHO = HO.inv(DECOMP_LU);
      entrar=1;

}
```

Cuando la posición del marcador en píxeles ha sido obtenida, hay que insertar la posición del marcador en el mundo, dato que se guarda en el vector `mundoO`. Con estos dos vectores, se puede obtener la matriz de homografía `HO` y su inversa `invHO`.

Como se he mencionado anteriormente, la detección de obstáculos se va a realizar mediante colores por lo que, para mejorar la detección de estos, la imagen en BGR (Azul, verde, rojo) se va a pasar a HSV (matiz, saturación, brillo) (Figura A40), lo que facilita la elección y detección de colores. Esto se consigue con la función

```
cvtColor(Imagen, hsv, CV_BGR2HSV);
```

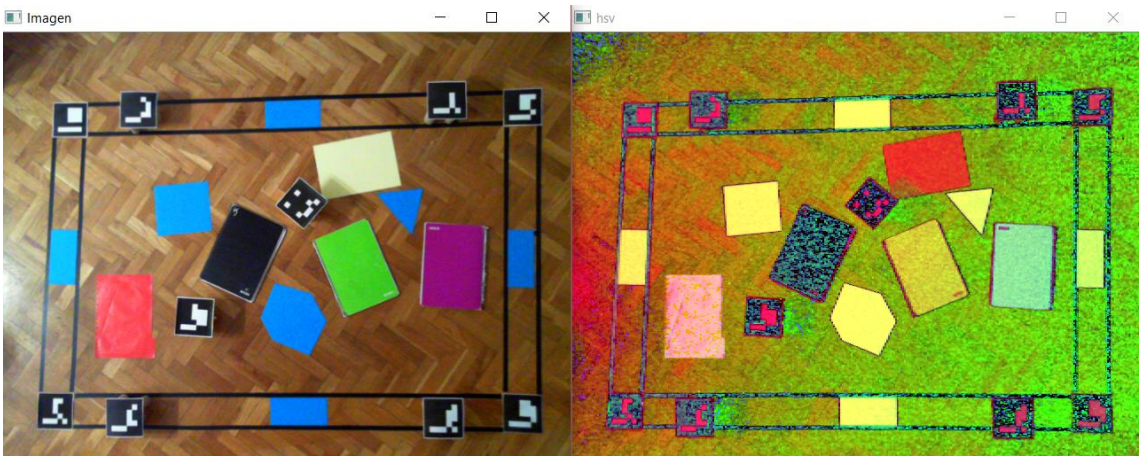


Figura A40: A la izquierda la imagen de la pista en BGR y a la derecha, la misma imagen en HSV

Cuando la imagen está en HSV, hay que obtener el color de referencia. Al programa se le pasarán las coordenadas mundo de la ubicación de las esquinas de cada color de referencia y estas coordenadas se pasarán a píxeles para que el programa pueda obtener los valores HSV del color deseado. El proceso de transformación de coordenadas mundo a pixeles, es parecido a los casos anteriores, salvo que en este hay que utilizar la matriz HO en vez de la matriz inversa.

```
convertPointsToHomogeneous(colorref1,colorrefH1);
colv10=colorrefH1[0];
colv11=colorrefH1[1];
colv12=colorrefH1[2];
colv13=colorrefH1[3];
colorrefHT10=Mat(colv10);
colorrefHT11=Mat(colv11);
colorrefHT12=Mat(colv12);
colorrefHT13=Mat(colv13);
HO.convertTo(HO2, CV_32FC1);
co10=HO2*colorrefHT10;
co11=HO2*colorrefHT11;
co12=HO2*colorrefHT12;
co13=HO2*colorrefHT13;
colvecH10= Mat_ <Point3f>(co10);
colvecH11= Mat_ <Point3f>(co11);
colvecH12= Mat_ <Point3f>(co12);
colvecH13= Mat_ <Point3f>(co13);
convertPointsFromHomogeneous(colvecH10,colvec10);
convertPointsFromHomogeneous(colvecH11,colvec11);
convertPointsFromHomogeneous(colvecH12,colvec12);
convertPointsFromHomogeneous(colvecH13,colvec13);
colorrefpixel11=Point2f(colvec10[0]);
colorrefpixel12=Point2f(colvec11[0]);
colorrefpixel13=Point2f(colvec12[0]);
colorrefpixel14=Point2f(colvec13[0]);
```

De aquí en adelante, como ocurre en este caso, si el código es muy extenso, solo se mostrara la parte correspondiente al color de referencia 1, teniendo que repetir el mismo proceso para los otros 3 colores de referencia. El resultado de aplicar este método es la obtención de 16 vectores, denominados colorrefpixel con el número del color de referencia y de la esquina, por

ejemplo, el vector `colorrefpixel24` correspondería a la esquina 4 del color de referencia número 2.

Una vez obtenidas las coordenadas de de las cartulinas en la imagen, hay que extraer el color que tiene cada una de ellas. Para realizar este proceso, OpenCV crea un rectángulo y va recorriendo cada punto interior del rectángulo, obteniendo los valores HSV. Aunque el color de referencia ubicado en la pista sea un rectángulo, lo más probable es que al mostrar el color por la cámara, este se haya deformado, quedando en su lugar un trapezoide por lo que hay q buscar el rectángulo interior formado por sus esquinas.

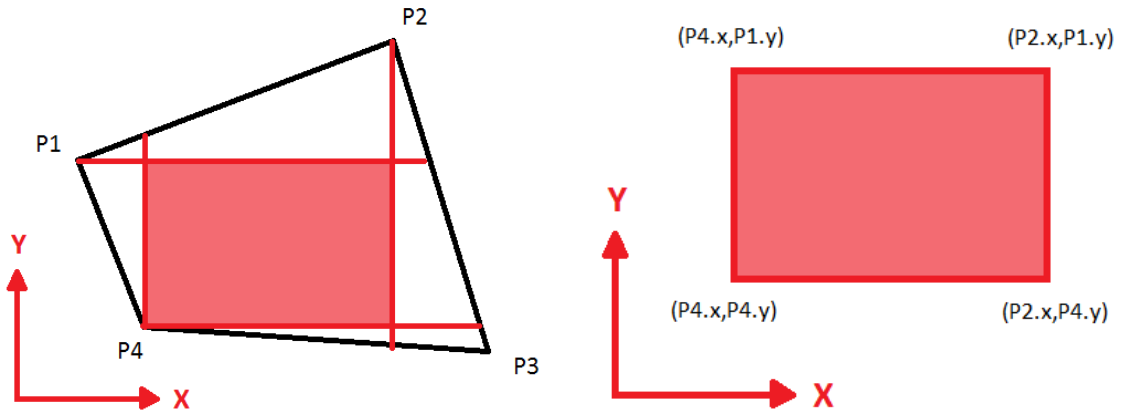


Figura A41: Rectángulo interior formado por las esquinas del trapezoide

Como no se conoce la orientación del trapezoide, ya que la cámara se puede mover, por lo que su orientación puede ir cambiando, hay que comprobar punto a punto la ubicación de las esquinas, esto se consigue con el siguiente código.

```
//color 1 coordenada x
if (colorrefpixel11.x<colorrefpixel12.x){
      col1p1.x=colorrefpixel11.x;
      col1p2.x=colorrefpixel12.x;}
else{
    col1p1.x=colorrefpixel12.x;
      col1p2.x=colorrefpixel11.x;}
if (col1p1.x<colorrefpixel13.x)
      if (col1p2.x<colorrefpixel13.x)
            col1p3.x=colorrefpixel13.x;
      else{
            col1p3.x=col1p2.x;
            col1p2.x=colorrefpixel13.x;}
else{
      col1p3.x=col1p2.x;
      col1p2.x=col1p1.x;
      col1p1.x=colorrefpixel13.x;}
if (col1p1.x<colorrefpixel14.x)
      if (col1p2.x<colorrefpixel14.x)
            if (col1p3.x<colorrefpixel14.x)
                  col1p4.x=colorrefpixel14.x;
            else{
                  col1p4.x=col1p3.x;
                  col1p3.x=colorrefpixel14.x;}
      else{
```

```
            col1p4.x=col1p3.x;
            col1p3.x=col1p2.x;
            col1p2.x=colorrefpixel14.x;}
else{
col1p4.x=col1p3.x;
col1p3.x=col1p2.x;
col1p2.x=col1p1.x;
col1p1.x=colorrefpixel14.x;}

//color 1 coordenada y
if (colorrefpixel11.y<colorrefpixel12.y){
      col1p1.y=colorrefpixel11.y;
      col1p2.y=colorrefpixel12.y;}
else{
      col1p1.y=colorrefpixel12.y;
      col1p2.y=colorrefpixel11.y;}
if (col1p1.y<colorrefpixel13.y)
      if (col1p2.y<colorrefpixel13.y)
            col1p3.y=colorrefpixel13.y;
      else{
            col1p3.y=col1p2.y;
            col1p2.y=colorrefpixel13.y;}
else{
      col1p3.y=col1p2.y;
      col1p2.y=col1p1.y;
      col1p1.y=colorrefpixel13.y;}
if (col1p1.y<colorrefpixel14.y)
      if (col1p2.y<colorrefpixel14.y)
            if (col1p3.y<colorrefpixel14.y)
                  col1p4.y=colorrefpixel14.y;
            else{
                  col1p4.y=col1p3.y;
                  col1p3.y=colorrefpixel14.y;}
      else{
            col1p4.y=col1p3.y;
            col1p3.y=col1p2.y;
            col1p2.y=colorrefpixel14.y;}
else{
      col1p4.y=col1p3.y;
      col1p3.y=col1p2.y;
      col1p2.y=col1p1.y;
      col1p1.y=colorrefpixel14.y;}
```

La función de este código es localizar las coordenadas del rectángulo interior, obteniéndose los siguientes resultados.

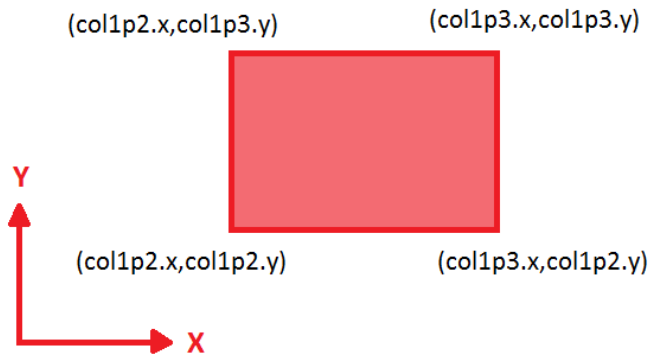


Figura A42: Coordenadas del rectángulo interno del color de referencia 1.

Una vez obtenida la localización del rectángulo interno, hay que extraerlo de la imagen y obtener el valor de sus colores.

```
rectangulo1.width = (col1p3.x - col1p2.x);
rectangulo1.height = (col1p3.y - col1p2.y);
rectangulo1.x = (col1p2.x);
rectangulo1.y = (col1p2.y); color1(hsv, rectangulo1);

Mat color1(hsv, rectangulo1);

meanStdDev ( color1, mean1, stddev1 );
```

En la imagen `hsv` se recorta el rectángulo obtenido anteriormente y, de esta nueva imagen llamada color1 (Figura A43), se extrae el valor medio y la desviación típica del matiz, saturación y brillo, mediante la función **`meanStdDev`**. Este proceso se repite para cada uno de los colores de referencia.

Figura A43: Rectángulos HSV extraídos de los 4 colores de referencia

Con los valores medios de HSV y las desviaciones típicas de cada color de referencia, hay que establecer un rango de valores para el cual, todos los puntos cuyo HSV que se encuentren dentro de ese rango, serán reconocidos como obstáculos. El valor de referencia se establece como el valor medio entre el valor máximo y mínimo de las medias de HSV obtenidas anteriormente. Para ello, hay que calcular el valor máximo de la desviación estándar y el valor máximo y mínimo de la media para cada valor hsv.

```
//Valor máximo de la desviación estándar del matiz del color de
referencia
if (stddev1.val[0]>stddev2.val[0])
      maximodev0.val[0]=stddev1.val[0];
else
      maximodev0.val[0]=stddev2.val[0];

if (maximodev0.val[0]>stddev3.val[0])
      maximodev0.val[1]=maximodev0.val[0];
  else
      maximodev0.val[1]=stddev3.val[0];

if (maximodev0.val[1]>stddev4.val[0])
      maximodev0.val[2]=maximodev0.val[1];
else
      maximodev0.val[2]=stddev4.val[0];


media.val[0]=(maximo0.val[2]+minimo0.val[2])/2;
media.val[1]=(maximo1.val[2]+minimo1.val[2])/2;
media.val[2]=(maximo2.val[2]+minimo2.val[2])/2;
```

A este valor medio habrá que aplicarle un rango mencionado anteriormente.

```
rango0= (maximo0.val[2]-minimo0.val[2])*1.5+maximodev0.val[2]*3;
rango1= (maximo1.val[2]-minimo1.val[2])+maximodev1.val[2]*4;
rango2= (maximo2.val[2]-minimo2.val[2])+maximodev2.val[2]*3;
```

En donde se ha decidido que el rango será la suma de la diferencia entre el valor medio mínimo y el valor medio máximo de los colores de referencia, más la desviación típica multiplicada por un factor. Este valor se sumará y se restara al valor medio obtenido anteriormente para lograr el rango de valores aceptables para que un punto sea reconocido como obstáculo.

```
if (media.val[0]<rango0)
      valormin.val[0]=0;
else
      valormin.val[0]=media.val[0]-rango0;

if (media.val[1]<rango1)
      valormin.val[1]=0;
else
      valormin.val[1]=media.val[1]-rango1;

if (media.val[2]<rango2)
      valormin.val[2]=0;
else
      valormin.val[2]=media.val[2]-rango2;

if (media.val[0]>255-rango0)
      valormax.val[0]=255;
else
      valormax.val[0]=media.val[0]+rango0;

if (media.val[1]>255-rango1)
      valormax.val[1]=255;
else
      valormax.val[1]=media.val[1]+rango1;

if (media.val[2]>255-rango2)
      valormax.val[2]=255;
else
      valormax.val[2]=media.val[2]+rango2;
```

Se crean dos vectores, `valormax` que contendrá los valores máximos de HSV que puede tener el punto para ser detectado como obstáculo y `valormin` que contendrá los valores mínimos. Los valores HSV se encuentran limitados entre 0 y 255.

La información que contiene el rango de valores admisible hay que llevarla a la imagen mediante la función **`inRange`**, que analiza la imagen HSV y devuelve una imagen binaria que muestra en blanco los puntos de la imagen que se encuentran dentro del rango admitido (Figura A44).

```
inRange(hsv, Scalar(valormin.val[0], valormin.val[1],
valormin.val[2]), Scalar(valormax.val[0], valormax.val[1],
valormax.val[2]), binaria);
```

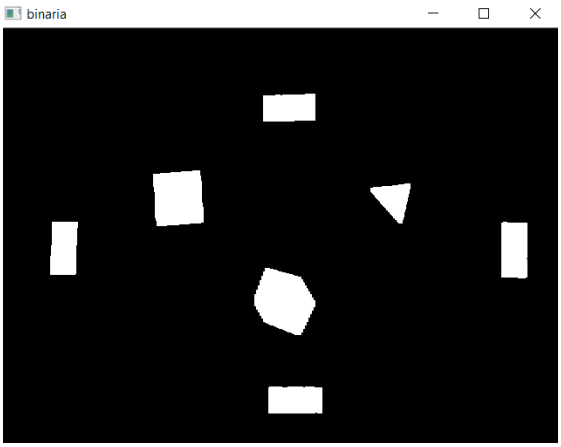


Figura A44: Imagen binaria de los obstáculos

Para mejorar detección, se va a modificar la imagen binaria. A esta imagen se le aplican una serie de operaciones de erosión y dilatación, la primera para eliminar posibles puntos erróneos detectados como objetos, pues reduce el tamaño de los obstáculos detectados y la segunda para devolver a los obstáculos a su tamaño original.

```
Mat element = getStructuringElement(MORPH_RECT, Size(2, 2));
erode(binaria, binaria, element);
dilate(binaria, binaria, element);
```

Para mejorar la detección de esquinas, se reduce el tamaño de la imagen con la función **resize**, multiplicando las dimensiones de la imagen por 0.3 (`tamanio`).

```
resize(binaria, binaria2, Size(), tamanio, tamanio, INTER_AREA);
```

Y de esta imagen, se extraen los contornos de los obstáculos, con la función **findContours.**

```
vector< vector<Point> > contornos;
findContours(binaria2,contornos,CV_RETR_EXTERNAL,
CV_CHAIN_APPROX_SIMPLE);
```

Con los contornos de los obstáculos guardados en un array, se quiere conocer la localización de las esquinas que forman cada uno de los obstáculos, para logar esto, se ha creado una imagen en negro (`negra`) en donde, para cada obstáculo, se dibujan y se rellenan los contornos que lo forman (**drawContours**).

Con los obstáculos ya detectados, hay que proceder a localizar las esquinas.

```
for (unsigned int i=0;i<contornos.size();i++)
{
	Mat negra2;
	negra2 = negra.clone();
	drawContours(negra2, contornos, i, Scalar(255), CV_FILLED);

	//Detección de esquinas
	goodFeaturesToTrack( negra2, corners, maxCorners, qualityLevel,
	minDistance, Mat(), blockSize, useHarrisDetector, k);

	//Parámetros para afinar la detección de esquinas
	Size winSize = Size( 5, 5 );
	Size zeroZone = Size( -1, -1 );
```

```
	TermCriteria criteria = TermCriteria( CV_TERMCRIT_EPS +
	CV_TERMCRIT_ITER, 40, 0.001 );

	//Calcular las esquinas definitivas con los parámetros
	anteriores
	cornerSubPix( negra2, corners, winSize, zeroZone, criteria );
	esquinafinal.push_back(corners);
	calculaesquina=1;

	//Convertir las coordenadas obtenidas en la imagen reducida a la
imagen real
	for(unsigned int j = 0; j < corners.size(); j++ )
	{
		esquinafinal[i][j].x=esquinafinal[i][j].x/tamanio;
		esquinafinal[i][j].y=esquinafinal[i][j].y/tamanio;
	}
}
```

La detección de esquinas se realiza con dos funciones, **goodFeaturesToTrack** y **cornerSubPix**. La primera lee los parámetros introducidos por el usuario y detecte las esquinas en la imagen, la con la segunda función, se obtiene unos valores más ajustados de las coordenadas de las esquinas.

Una vez que las esquinas han sido detectadas, sus coordenadas se guardan en el array `esquinafinal`, pero como la imagen de donde se han obtenido dichos valores había sido reducida, hay que realizar un cambio de escala para obtener la localización de la esquina en la imagen real, esto se consigue dividiendo cada componente del array por el factor utilizado anteriormente (`tamanio=0.3`).

Con las coordenadas de las esquinas guardadas, hay que obtener sus coordenadas en la pista, este proceso, como se he ido realizado en numerosas ocasiones en este programa, se realiza con la matriz de homografía `invHO` obteniéndose un vector (`esquinamundo`), con la localización en coordenadas mundo de las esquinas del obstáculo. Como el obstáculo detectado puede que esté dentro o fuera de la pista o ambas a la vez, se ha decidido que si la localización de alguna de las esquinas del obstáculo se encuentra fuera de la pista, se reubicarán con las coordenadas que limitan la pista.

```
esquinapixel[j].x=esquinafinal[i][j].x;
esquinapixel[j].y=esquinafinal[i][j].y;
convertPointsToHomogeneous(esquinapixel,esquinapixelH);
Mat esquinapixelHT=Mat(esquinapixelH[j]);
Mat invHO2 = Mat::eye(3, 3, CV_32FC1);
invHO.convertTo(invHO2, CV_32FC1);
Mat esquina2=invHO2*esquinapixelHT;
vector<Point3f> esquinavecH = Mat_ <Point3f>(esquina2);
vector<Point2f> esquinavec;
convertPointsFromHomogeneous(esquinavecH,esquinavec);
esquinamundo[j]=Point2f(esquinavec[0]);

if (esquinamundo[j].x>largoreal)
	esquinamundo[j].x=largoreal;
if (esquinamundo[j].y>anchoreal)
	esquinamundo[j].y=anchoreal;
if (esquinamundo[j].x<0)
	esquinamundo[j].x=0;
if (esquinamundo[j].y<0)
```

```
esquinamundo[j].y=0;
```

Este proceso se realiza para cada obstáculo detectado, por lo que la variable `esquinamundo` se sobrescribe y para no perder la información, estos valores se almacenan en un array denominado `esquina`.

```
esquina.push_back(esquinamundo);
```

Las coordenadas de las esquinas de cada obstáculo se mostrarán por pantalla con el siguiente bucle

```
for(unsigned int i = 0; i < esquinafinal.size(); i++ )
      for(unsigned int j = 0; j < esquinafinal[i].size(); j++ )
            cout<<"obstáculo["<<i+1<<"] esquina["<<j+1<<"]:
            "<<esquina[i][j]<<endl;
```

Y las de los robots con el siguiente.

```
for( int i=0;i<nrobots;i++)
{
      theta=(atan2((punto2mundo[i].y-
      punto1mundo[i].y),(punto2mundo[i].x-punto1mundo[i].x)));
      Robotmundo[i]=Point3f(Robotm[i].x,Robotm[i].y,theta);
      cout<<"Robot["<<i+1<<"](x,y,theta):"<<Robotmundo[i]<<endl;
}
```

Donde se obtiene el ángulo theta del robot con la función `atan2`.

Para mejorar la visualización de los resultados se va a crear una imagen en donde solo aparezca la con pista, los robots y los obstáculos. Para realizar esto, hay que crear una nueva matriz de homografía para pasar los puntos de la imagen con la pista deformada a la nueva imagen.

```
pistaI[0]=Point2f(0,0);
pistaI[1]=Point2f(ancho,0);
pistaI[2]=Point2f(ancho,alto);
pistaI[3]=Point2f(0,alto);
HP=findHomography(esquinaO,pistaI);
```

Y tanto a los robots como a las esquinas habrá que aplicarle dicha matriz para obtener los nuevos valores, como se ha visto en numerosas ocasiones a lo largo de la descripción de este programa.

Una vez obtenidos estos valores, habrá que dibujar los robots.

```
radioP=radioRobot*ancho/largoreal;
circle(Plataforma2, Point(RobotI[i].x,RobotI[i].y), radioP,
Scalar(0,255,0), -1);
line(Plataforma2, Point(RobotI[i].x,RobotI[i].y),
Point(puntoP[i].x,puntoP[i].y), CV_RGB(255,0,0), 2, CV_AA);
ostringstream str;
str<<"R"<<i+1;
putText(Plataforma2,str.str(),Point(RobotI[i].x-
15,RobotI[i].y),FONT_HERSHEY_DUPLEX,0.7,Scalar(0,0,0),1);
```

Los robots tienen una superficie circular por lo que se dibujan con la función **circle** con el radio (radioP) calculado anteriormente. El ángulo theta que tiene el robot se representará con una línea que nace en el centro del robot y a cada robot se le dibujara un identificativo (R1,R2,…).

Los obstáculos se dibujarán con la función **drawContours.**

```
drawContours(Plataforma2, contornos6, -1, Scalar(255,128,0),
CV_FILLED);
```

Y se crearán una serie de líneas a modo de ejes de coordenadas.

```
line(Plataforma2, Point (5,alto-5), Point (ancho/8,alto-5),
CV_RGB(0,0,0), 2, CV_AA);
line(Plataforma2, Point (ancho/8,alto-5), Point (ancho/8-10,alto-10),
CV_RGB(0,0,0), 2, CV_AA);
line(Plataforma2, Point (ancho/8,alto-5), Point (ancho/8-10,alto),
CV_RGB(0,0,0), 2, CV_AA);
putText(Plataforma2,"X",Point
(ancho/8+2,alto),FONT_HERSHEY_DUPLEX,1,Scalar(0,0,0),1);
line(Plataforma2, Point (5,alto-5), Point (5,6*alto/7), CV_RGB(0,0,0),
2, CV_AA);
line(Plataforma2, Point (5,6*alto/7), Point (10,6*alto/7+10),
CV_RGB(0,0,0), 2, CV_AA);
line(Plataforma2, Point (5,6*alto/7), Point (0,6*alto/7+10),
CV_RGB(0,0,0), 2, CV_AA);
putText(Plataforma2,"Y",Point (0,6*alto/7-
5),FONT_HERSHEY_DUPLEX,1,Scalar(0,0,0),1);
```

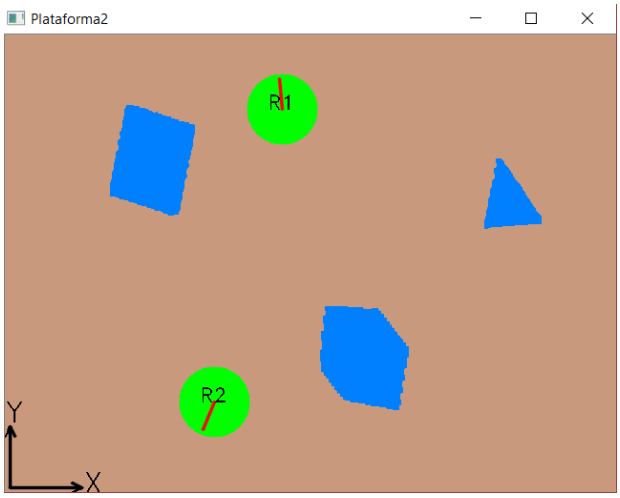


Figura A45: Pista por la que se mueven los robots con los obstáculos y los robots detectados.