



**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**

**Trabajo fin de Grado**

**Grado en Ingeniería Informática  
Ingeniería del Software**

**Virtualización y resolución automática del cubo de Rubik**

**Realizado por  
Alejandro Monteseirín Puig**

**Dirigido por  
Francisco Jesús Martín Mateos**

**Departamento  
Ciencias de la Computación  
e Inteligencia Artificial**

**Sevilla, Julio de 2019**



---

## Índice general

---

Índice general	I
Índice de tablas	III
Índice de figuras	V
Índice de código	VII
<b>1 Introducción</b>	<b>1</b>
1.1 Motivación . . . . .	1
1.2 Contexto Tecnológico . . . . .	2
1.3 Estructura del documento . . . . .	2
<b>2 Diseño de la aplicación</b>	<b>5</b>
2.1 OPENCV . . . . .	6
2.2 Espacios de color . . . . .	7
2.3 Funcionamiento General . . . . .	8
2.4 Sistema de virtualización del cubo . . . . .	8
2.5 Sistema de calibración de colores . . . . .	10
2.6 Sistema de resolución del cubo . . . . .	12
2.7 Interfaz de usuario, Funcionamiento . . . . .	16
<b>3 Manual de uso</b>	<b>21</b>
3.1 Elementos de la interfaz . . . . .	21
3.2 Preparación inicial . . . . .	37
3.3 Fase de calibración . . . . .	37
3.4 Virtualización de las caras . . . . .	38
3.5 Resolver Cubo . . . . .	38
<b>4 Conclusiones</b>	<b>41</b>
4.1 Resultado Final . . . . .	41
4.2 Lecciones aprendidas . . . . .	41
4.3 Elementos a mejorar . . . . .	42
<b>5 Agradecimientos</b>	<b>43</b>
<b>Bibliografía</b>	<b>45</b>



---

## Índice de tablas

---

3.1	Botón cambio de modo . . . . .	21
3.2	Selector de color . . . . .	22
3.3	Sistema de Colores de la Calibración RGB . . . . .	23
3.4	Sistema de Colores de la Calibración HSV . . . . .	24
3.5	Selector de la cara del cubo . . . . .	25
3.6	PopUp de guardar cara . . . . .	26
3.7	Botón de calibrar . . . . .	27
3.8	Botón de calibrar con click . . . . .	28
3.9	popUp de calibrar con click . . . . .	29
3.10	Input de IP y Botón de modo móvil . . . . .	30
3.11	Cubo virtualizado . . . . .	31
3.12	Pantalla izquierda Modo Auto . . . . .	32
3.13	Pantalla derecha modo auto . . . . .	33
3.14	Pantalla izquierda modo manual . . . . .	34
3.15	Pantalla derecha modo manual . . . . .	35
3.16	Botón de resolver Cubo y resultados . . . . .	36



---

## Índice de figuras

---

2.1	Cono HSV . . . . .	7
2.2	Funcionamiento general de la aplicación . . . . .	8
2.3	Imagen con los contornos detectados(izquierda) e Imagen tras aplicar las mascaras de colores(derecha) . . . . .	10
2.4	Estado Intermedio del algoritmo de Kociemba . . . . .	13
2.5	Interfaz y elementos . . . . .	15
2.6	PopUp De la imagen obtenida . . . . .	18
2.7	Cubo desmontado inicializado . . . . .	19
2.8	Proceso de virtualización del cubo . . . . .	19
3.1	Sistema de calibración con click ventana desplegable, calibrando el rojo . .	38
3.2	Notación cubo de Rubik . . . . .	39
3.3	Resultado del solver . . . . .	39





---

## Índice de código

---

2.1	Fichero <code>Video.py</code> . . . . .	9
2.2	Obtención del frame modificado del modo auto, Fichero <code>Video.py</code> . . . . .	16
2.3	Obtención del frame modificado del modo Manual, Fichero <code>Video.py</code> . . . . .	17
2.4	Generación de la imagen del PopUp ResultadosCara Fichero <code>VentanaDesplegable.py</code>	18



# CAPÍTULO 1

---

## Introducción

---

El objetivo de este proyecto es crear una aplicación ejecutable con interfaz dinámica que, mediante el uso de la cámara, permita detectar los distintos colores de las caras de un cubo de Rubik, para lograr virtualizarlo y devolver al usuario un conjunto de movimientos para resolverlo.

### 1.1— Motivación

La resolución de un cubo de Rubik siempre ha sido un problema que ha entusiasmado a los informáticos, dado que tiene tantas maneras de combinar sus piezas que hace que sea imposible resolverlo mediante los algoritmos de búsqueda más típicos, y menos aún por aquellos basados en fuerza bruta. Por otro lado, su forma y colores aparentemente sencillos atraen la idea de realizar su virtualización mediante el reconocimiento de imágenes. El objetivo de este proyecto es la obtención de ambos objetivos, lo que nos permitiría automatizar el proceso de resolver un modelo real del cubo de Rubik.

Para ver la inmensa cantidad de posibles configuraciones que puede adoptar un cubo de Rubik podemos realizar unos simples cálculos: Hay 8 vértices y 12 aristas; solo con las distintas permutaciones de los vértices tenemos 40320 ( $8!$ ) combinaciones posibles; teniendo en cuenta que cada vértice tiene 3 posibles orientaciones tenemos otras 2187 ( $3^7$ ) posibilidades (ya que siempre tendremos que dejar una fija); con las aristas tenemos ( $12! \div 2$ ) posibles permutaciones (dividimos entre dos debido a temas de paridad); y contando las rotaciones de las aristas obtenemos otras 2048 ( $2^{11}$ ) posibilidades.

El resultado final es:  $\frac{8! \cdot 12! \cdot 3^7 \cdot 2^{11}}{2} = 43252003274489856000$  configuraciones.

Dada la enorme cantidad de posibles estados que puede llegar a tener el cubo, cualquier método de búsqueda de estados convencional queda totalmente descartado, dado que el tiempo de procesamiento sería desorbitado.

Este proyecto se puede resumir en dos partes separadas, la resolución del cubo y su virtualización:

## Algoritmos de resolución

Existen diversos algoritmos que permiten resolver el cubo dado un estado inicial, los hay más caseros que resuelven el cubo como lo haría una persona y los hay más optimizados enfocados en conseguir el menor numero de movimientos y el menor tiempo de procesamiento. En nuestra aplicación utilizamos un algoritmo del segundo tipo, principalmente porque necesitamos que se resuelva rápidamente (ya que el usuario tiene que esperar la respuesta) y con pocos movimientos (para evitar como respuesta una engorrosa y larga lista de movimientos para resolver el cubo). En concreto usamos una versión modificada del algoritmo de dos fases de Kociemba.[1], en el que profundizaremos más adelante.

## Virtualización

Basándonos en la existencia de estos algoritmos, sería posible aplicarlos a un programa que introduciendo los valores de cada cara de un cubo, generara un cubo virtual, le aplicara dichos algoritmos de resolución y devolviera los movimientos que se deben hacer para resolver el cubo original. Esto tiene un problema, tendríamos que introducir 54 valores (6 caras de 9 casillas), lo cual además de ser muy tedioso y confuso es arriesgado, ya que un solo valor erróneo provocaría que el programa falle, dado que el cubo no estaría correctamente formado. La solución a este problema consiste en la virtualización del cubo mediante el uso de una webcam, cámara externa o móvil. Esto permitiría de una manera sencilla virtualizar cada cara de forma rápida, intuitiva y con una mayor fiabilidad, dado que veríamos en tiempo real los colores elegidos. Esta virtualización unida a una interfaz intuitiva y con multitud de opciones facilitan enormemente la tarea de introducir los valores del cubo en la aplicación.

## 1.2– Contexto Tecnológico

En la actualidad el reconocimiento de imágenes es una tecnología en auge, que evoluciona rápidamente y que cada vez tiene más importancia. Desarrollos como el de los vehículos auto-pilotados y el reconocimiento de rostros están en su mayoría en una fase inicial, iniciando su despegue como tecnologías solidas y de confianza debido principalmente a que no es una tarea trivial, sino bastante compleja, especialmente cuando se trata de un vídeo y se necesita un procesamiento rápido que devuelva los resultados al instante. En este trabajo afrontaremos dificultades similares a las que se han tenido que afrontar en dichos proyectos como la iluminación, los distintos formatos de detección de color (utilizaremos tanto RGB como HSV), la correcta detección de los contornos o los distintos enfoques del sistema de grabación.

## 1.3– Estructura del documento

- **Introducción** Se presenta una introducción al proyecto, su objetivo y las motivaciones, además de hablar sobre el contexto tecnológico y sobre los distintos conocimientos que se aplican en el proyecto.
- **Diseño de la aplicación** Se aborda el diseño de la aplicación de una forma más técnica y detallada, explicando paso a paso los distintos elementos y sistemas hasta explicar por completo la aplicación.

- **Manual de uso** Se muestra una versión menos técnica del proyecto, centrándose principalmente en como usar la aplicación. Se apoya en distintas capturas de la interfaz, explicando cada elemento y cómo usarlo correctamente.
- **Conclusiones** Se analiza el resultado obtenido, las lecciones aprendidas y los elementos del proyecto que podrían mejorarse en caso de querer expandirlo.



---

### Diseño de la aplicación

---

Para el diseño de la aplicación se ha optado por PYTHON, dada la gran versatilidad de este lenguaje y la existencia de múltiples librerías con multitud de utilidades específicas para este proyecto en cuestión. Cabe destacar la librería `OPENCV`, utilizada para el procesamiento de imágenes y el tratamiento de los colores, en general todo el apartado de la virtualización del cubo esta sostenido en esta librería (apoyada por `NUMPY` para las operaciones matemáticas), ya que es la encargada de capturar la imagen de la webcam y aplicar sobre ella los distintos filtros y máscaras que nos permiten detectar los colores de cada cara del cubo. El proyecto se divide en dos archivos principales y una carpeta de recursos donde se almacenan imágenes, scripts y librerías externos al desarrollo principal.

Los dos archivos son:

- `Video.py`: Genera el flujo de video y datos que se enviará a la interfaz.
- `VentanaDesplegable.py`: Contiene los elementos de la interfaz y la construye en base a los datos recibidos de `Video.py` al cual también envía ciertos parámetros como el modo en el que se encuentra la interfaz, los valores de calibración, etc.

Para iniciar el proyecto basta con lanzar `VentanaDesplegable.py` el cual generará una instancia de la clase `Video` y le pasará los parámetros iniciales. La instancia de `Video` se inicializará y empezará a enviar el flujo de datos a la interfaz.

En cuanto al almacenamiento del proyecto, este se ha ido subiendo mediante el uso de GIT en un repositorio de GITHUB con una breve explicación en cada subida de cambios. Se realizaba una subida de cambios cada vez que se consideraba un avance importante y funcional. Así mismo el número de distintas subidas de cambios no refleja la cantidad de trabajo, dado que una sola subida puede agrupar multitud de cambios, por lo que hay subidas de cambios que implementen sistemas enteros u otros que simplemente arreglan algún error. El repositorio se mantuvo privado para evitar filtraciones de código hasta algo antes de la fecha de la entrega, cuando se hizo público. Dicho repositorio se encuentra accesible en la siguiente URL <https://github.com/AlejandroMonteseirin/TFG-Rubik-solver>

## 2.1— OPENCV

Es necesario profundizar más en la librería OPENCV para entender el funcionamiento interno de la aplicación. OPENCV es una librería de código libre, como su nombre indica, que nos aporta una gran cantidad de funciones y utilidades para la visualización y la edición de vídeo.

Las utilidades más importantes que usaremos son aquellas relacionadas con la creación de ventanas emergentes para la interfaz, las relacionadas con el color, la edición de la imagen y la detección y procesamiento de contornos.

Las principales funciones que usaremos en el proyecto son:

- **VideoCapture**: Nos permite acceder al vídeo de la cámara en tiempo real.
- **imread**: Nos permite acceder a una imagen.
- **resize**: Nos permite redimensionar una imagen.
- **namedWindow**: Nos permite nombrar una ventana con una imagen.
- **imshow**: Nos permite mostrar una ventana como un desplegable.
- **moveWindow**: Nos permite mover una ventana.
- **destroyWindow**: Nos permite destruir una ventana.
- **cvtColor**: Nos permite cambiar el formato de color de una imagen (por ejemplo de RGB a HSV).
- **putText**: Nos permite escribir un texto en un lugar concreto de una imagen.
- **rectangle**: Nos permite pintar un rectángulo en un lugar concreto de una imagen.
- **MouseEventTypes** : Nos permite capturar los distintos eventos del ratón que recibe una ventana creada, que puede ser una imagen (eso nos permite por ejemplo mostrar una ventana emergente con la cara virtualizada del cubo y que al hacer click en una casilla de la cara, se capture la posición del ratón en ese momento y llame a una función sobre esa casilla para cambiar el color de dicha casilla).
- **drawContours**: Nos permite pintar un contorno previamente detectado.
- **mean**: Nos da el valor medio del color en una imagen (dentro de un contorno o una máscara).
- **inRange**: Sirve para comprobar si los elementos de la imagen se encuentran dentro de un rango dado, es utilizado en el proyecto para crear una máscara con los valores dentro de un rango (por ejemplo la máscara del rango de color amarillo, tras aplicarla muestra la misma imagen pero solo los fragmentos dentro de ese rango).
- **findContours**: Encuentra los contornos dada una máscara.
- **contourArea**: Calcula el área de un contorno.

Utilizando dichas funciones y algunas más seremos capaces de capturar la imagen y procesarla, para virtualizar correctamente el cubo.



## 2.2– Espacios de color

Para comprender el proyecto es necesario profundizar más en los dos espacios de color [2] que utilizaremos para la virtualización, dado que casi todas las funciones trabajan simultáneamente con ambos.

### Espacio de color RGB/BGR

Es el espacio de color estándar más conocido en informática. Almacena valores individuales para el rojo, el verde y el azul de manera aditiva, permitiendo así conocer el color. En el proyecto se utiliza tanto RGB, como BGR debido a que algunas funciones de `OPENCV` requieren una reordenación del rojo y el azul por motivos de diseño. No se trata de una transformación compleja, simplemente consiste en cambiar el orden de los valores.

El principal problema de este espacio de color, y por el cual necesitamos apoyarnos en otro, es que al ser aditivo, los cambios en la iluminación cambian drásticamente los valores. Esto añade una gran dificultad para detectar el color de que se trata en entornos con iluminación cambiante, en la que el movimiento (de la cámara o el cubo) provoca que la iluminación incida de una manera u otra haciendo que cambien los valores.

### Espacio de color HSV

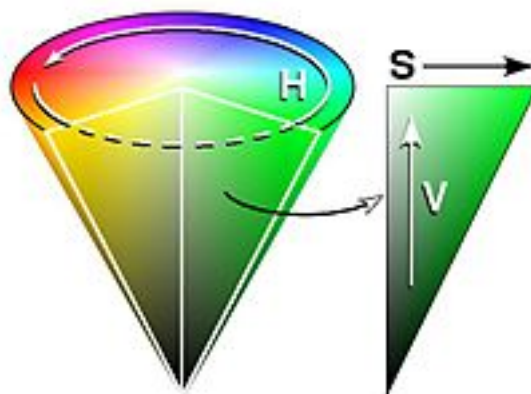


Figura 2.1: Cono HSV

El modelo HSV (del inglés Hue, Saturation, Value – Matiz, Saturación, Valor) nos permite separar claramente los valores de iluminación del valor del color, definido como Hue, esto nos da la posibilidad de fijarnos principalmente en esta característica para la detección de colores, mientras damos un menor valor a las otras.

Sin embargo no todo es tan sencillo como parece, el modelo HSV a su vez provoca ciertas dificultades a la hora de detectar el blanco, ya que puede llegar a confundirse con colores claros de baja saturación o elementos del fondo de la pantalla. Por esto se tuvo que pulir mucho el sistema de calibración para impedir falsos positivos. Otra de las dificultades a la hora de asignar un rango de color para cada color fue en relación con el color rojo, este se encuentra en los últimos valores de matiz y también en los primeros, teniendo que utilizar 2 rangos para este color.

### 2.3— Funcionamiento General

Al iniciar la aplicación se mostrará en la interfaz el sistema de virtualización, deberemos de seleccionar la cara del cubo que estamos virtualizando (frontal, izquierda, derecha, superior, inferior o trasera) y utilizando uno de los dos sistemas posibles de virtualización enviar los datos de esta cara. Una vez virtualizada una cara, proseguiremos con las siguientes hasta tener todo el cubo virtualizado. Con todo el cubo virtualizado aparecerá un botón que nos permitirá resolverlo, tras un breve cálculo nos dará los movimientos necesarios para la resolución del cubo siempre que éste esté correctamente formado.

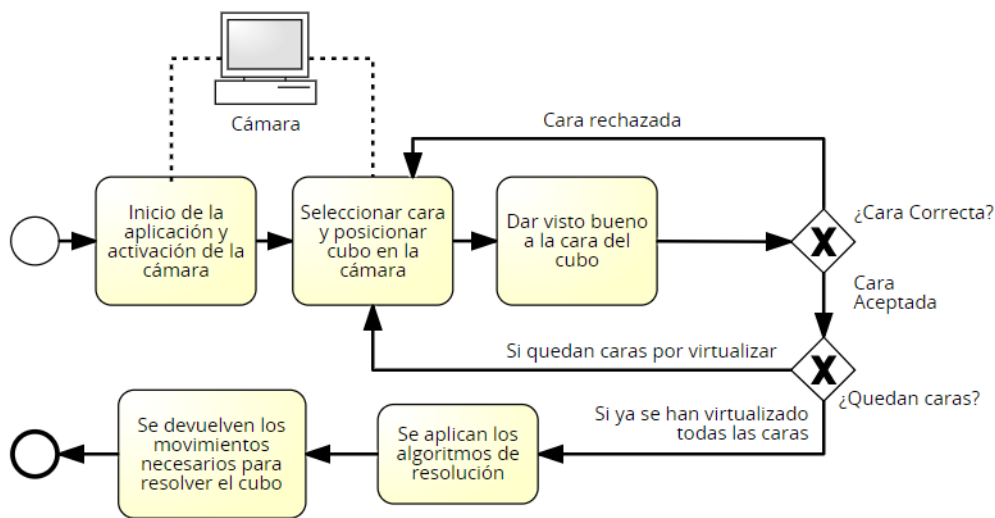


Figura 2.2: Funcionamiento general de la aplicación

Pasamos ahora a explicar en mayor detalle el diseño de cada apartado de la aplicación.

### 2.4— Sistema de virtualización del cubo

El sistema de virtualización del cubo es el apartado principal de la aplicación, permitiendo virtualizar el cubo en la aplicación de diversas maneras, para su posterior resolución.

Para virtualizar el cubo nos basamos en el uso de un vídeo en directo que viene a ser un flujo constante de imágenes. La aplicación procesa este flujo imagen por imagen, enviando los datos a los distintos algoritmos que la procesarán.

El vídeo puede ser obtenido tanto desde la típica cámara de portátil, como de una webcam externa. Además incluye la opción de integrar la cámara de un móvil de una manera inalámbrica, lo cual proporcionará una mucha mejor calidad de la imagen si se trata de un móvil de alta gama. Para la conexión mediante móvil, consultar en el manual de uso el apartado *preparacion inicial, configuración modo movil*.

La configuración inicial será usando OPENCV, que de existir una cámara extra (webcam externa) usará esa, y si no usará la definida por defecto en el sistema (webcam portátil). Si no existe ninguna cámara, se activará el modo móvil con la IP por defecto.

Si no hay ningún medio para grabar conectado, mostrará un frame de error y esperará a que se realice una conexión de un elemento de grabación de video.

```
1
2 #Inicializacion de la camara
3 self.cap = cv2.VideoCapture(1)
4 #Si no hay camara extra prueba a ver si existe una camara por defecto
5 if self.cap is None or not self.cap.isOpened():
6     self.cap = cv2.VideoCapture(0)
7     #Si no hay camaras activadas, activa el modo movil
8     if self.cap is None or not self.cap.isOpened():
9         self.movil['activado']=True
```

Código 2.1: Fichero Video.py

A continuación se describen los 2 modos principales de virtualización.

### Modo básico

El modo básico funciona aplicando una máscara a la cámara, que consiste en una cuadrícula 3x3, en la cual el usuario debe “encajar” el cubo. El usuario colocará el cubo de manera que cada cuadrado entre en su cuadrícula correspondiente, solo es necesario entonces calcular el color dentro de cada cuadrícula.

Para detectar el color de cada cuadrado, se compara con los colores definidos y se elige el más parecido, tanto en formato RGB como en HSV (para evitar así problemas debido a la iluminación).

Aun así y debido a que los colores de cada cubo pueden variar, ya sea por la marca del cubo (distintas tonalidades) o por la iluminación ambiente, se incluye un sistema de calibración que permite calibrar cada color del sistema para una mejor detección.

Finalmente, cuando la cuadrícula se corresponde con el cubo, el usuario debe pulsar el botón de guardar y se muestra una confirmación en donde podrá editar alguna casilla en caso de que exista algún error.

Realizando este proceso con cada cara se puede virtualizar el cubo rápidamente. Sin embargo, el principal problema, además de tener que estar “encajando” el cubo en la cuadrícula, es que el hecho de cambiar el cubo de sitio hace variar la iluminación en cada casilla provocando cierta ineficacia a la hora de detectar los colores. Para solventar ambos problemas, se ha desarrollado el modo auto.

Cabe comentar que es posible utilizar el modo manual dándole a guardar sin encajar el cubo (no se detectará ninguna casilla) y cuando salga la ventana de confirmación modificar cada casilla manualmente, es útil si no se dispone de un entorno adecuado para usar la cámara y/o no hay una correcta iluminación.

### Modo auto

El modo auto permite eliminar la tediosa tarea de colocar el cubo en la rejilla de la cámara y detectar mediante software los cuadrados del cubo, esto permite una mayor rapidez y comodidad a la hora de virtualizar cada cara del cubo.

El sistema funciona aplicando máscaras de cada color a la imagen recibida de la cámara, estas máscaras contienen los rangos aceptables de los colores, siendo editables desde el

sistema de calibración, del cual hablaremos más adelante en profundidad. De esta forma se filtra la imagen recibida por todos los colores del cubo de rubik, y se obtiene una imagen en la que solo se mostrarán los cuadrados del cubo de rubik.

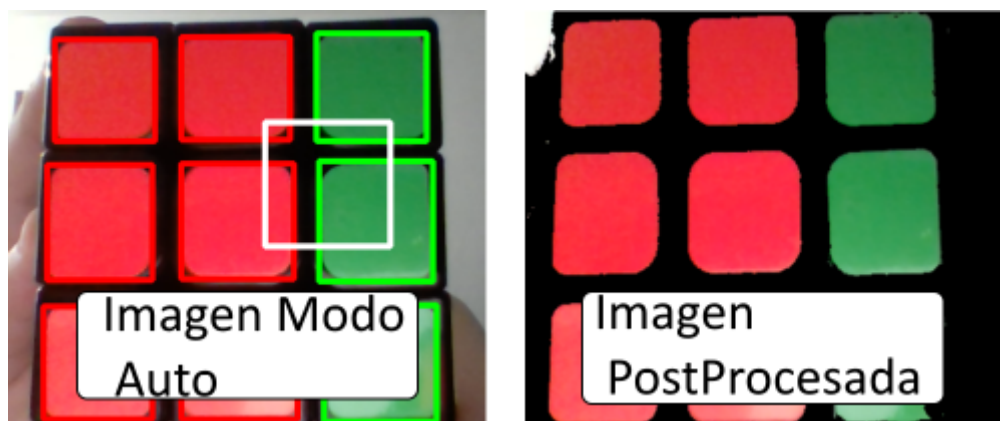


Figura 2.3: Imagen con los contornos detectados(izquierda) e Imagen tras aplicar las mascarar de colores(derecha)

Como podemos apreciar este modo logra captar los colores que necesitamos e ignorar aquellos que no pertenecen al cubo de Rubik. Una vez tenemos esta imagen, utilizando OPENCV podemos delimitar los contornos, para obtener los cuadrados del cubo. A pesar de todo, no sabremos cuál va en cada lugar así que con los cuadrados ya delimitados y sus colores reconocidos realizamos una ordenación sabiendo sus coordenadas para saber cual es cada uno. Primero los ordenamos en 3 grupos en función de su altura, así tendríamos 3 grupos de 3 cuadrados cada uno, los 3 de arriba, los 3 en medio y los 3 abajo, después ordenamos cada grupo de izquierda a derecha y así ya sabríamos cual es la posición de cada uno. El sistema incluso admite cierto margen de inclinación siempre y cuando las alturas no estén mezcladas (por ejemplo, un contorno de abajo mas alto que un contorno del centro).

Cuando el sistema detecta 9 cuadrados estables en la imagen, da la captura por válida y la muestra al usuario para que este pueda validarla y pasar a la siguiente.

Al igual que el sistema anterior, los cambios en el modelo del cubo (por ejemplo un cubo con una cara negra en vez que blanca) o en la iluminación, pueden impedir que los valores por defecto de cada color se ajusten a la realidad, para ello se incluye un sistema de calibración propio para este modo.

## 2.5— Sistema de calibración de colores

Una de las primeras dificultades que nos encontramos al intentar virtualizar el cubo son los colores. Aunque pueda parecer una tarea trivial (detectar el valor RGB de la imagen en cada cuadrado y de ahí asignarle un color), tiene una complejidad mucho mayor ya que cualquier variación en el entorno o en el dispositivo cambia totalmente los resultados obtenidos. Es decir, el hecho de apagar la luz de fondo o pasar de una webcam a otra puede hacer que el valor de cada color cambie radicalmente, provocando resultados totalmente erróneos. El nivel de cambio es tal, que la propia inclinación del cubo respecto a la luz, puede provocar que los valores cambien y dejen de funcionar correctamente.

Para intentar minimizar los problemas de la iluminación todo el sistema se basa en valores RGB trabajando en conjunto con los valores HSV [4].

El modelo de color HSV[5] nos permite separar claramente el matiz del color (Hue) de la saturación (Saturation) y el brillo (Brightness). Esto permite analizar principalmente el matiz para detectar el color (excepto para el color blanco, para el que solo analizamos el brillo y la saturación). Al usar el sistema HSV en conjunto con los valores RGB se obtiene un resultado mucho más estable que solo con los valores RGB, mejorando considerablemente el resultado.

Aún con el sistema de RGB+HSV, los valores de cada color del cubo seguían inestables en función de la iluminación del entorno y del propio cubo (ya que en función de la marca del cubo de Rubik puede tener unos tonos y/o colores distintos) Para afrontar este problema se desarrolló un sistema de calibración, el cual permitía calibrar los colores en el caso de que los valores de cada color no coincidieran con los valores por defecto de la aplicación.

El sistema de calibración ha ido evolucionando a lo largo del desarrollo del proyecto, conforme la complejidad de este aumentaba y se necesitaban nuevos métodos para calibrarlo. Finalmente se han desarrollado dos métodos de calibración.

### **Sistema de calibración fijo**

El primero y más sencillo de implementar. Sobre el vídeo recibido se dibuja un pequeño cuadrado en el centro, obtenido mediante la fusión de una imagen-máscara consistente en ese cuadrado y la imagen del vídeo recibido desde la cámara. El usuario debe colocar la casilla del cubo a calibrar dentro de ese cuadrado, seleccionar el color que está calibrando y finalmente pulsar el botón de calibrar. El programa obtiene el valor medio dentro de esa casilla y lo asocia con el color seleccionado.

La principal desventaja de este método, la cual provocó que tuviera que desarrollarse una versión más avanzada, fue que el simple hecho de mover el cubo para encajarlo en el cuadrado del vídeo provocaba que el color variase, debido a que la luz cambiaba entre una posición y otra, alterando el color o incluso el enfoque de la cámara se perdía, desencadenando un color erróneo.

### **Sistema de calibración mediante click**

Con el modo auto, era un retroceso tener que mover el cubo a la casilla para calibrarlo, así que se desarrolló un sistema que desplegaba una pestaña con la imagen en la que seleccionar la casilla del cubo. Para seleccionarla simplemente se arrastra el ratón en la imagen dibujando un cuadrado, al soltarlo se dará como seleccionado ese cuadrado y la aplicación lo utilizará como si se tratase de la casilla a calibrar. Esto permite que el cubo pueda estar quieto, mientras nosotros desde la aplicación seleccionamos las casillas que deseamos calibrar, permitiendo así que la medición del color no se vea afectada por distintos tonos de iluminación o el propio enfoque de la cámara.

### **Detección de colores**

Con el sistema de calibración y los distintos modos de virtualización ya explicados pasamos a la siguiente cuestión: ¿cómo decide la aplicación qué color pertenece a cada casilla? En el modo auto, filtra cada máscara de color individualmente antes de unir las

para enviarlo a la interfaz. De esta forma, si al filtrar con el color rojo se detectan dos contornos entonces esos dos son declarados como rojo en el sistema[3].

En el modo manual, se obtiene la media de color de cada casilla y se compara con los valores RGB y los HSV del sistema de calibración, para el RGB se compara la proporción de rojo, verde y azul entre dichos 2 valores, y se elige el color que tenga la proporción más similar con el recibido.

Tras eso se comparan los valores HSV de la media del color de cada casilla y los valores HSV del sistema de calibración. Se le da más peso a la componente *Hue* a la hora de comparar, ya que es el parámetro más importante para la detección de color, pero también se le da valor al *Saturation* y el *Value*.

Finalmente, se escoge el color que más se parece del sistema de calibración. De esta forma obtenemos el color que más se parece en RGB y en HSV. Si ambos coinciden, el color detectado cambia a ese, si no, se mantiene el color anterior. Esto evita que el color este cambiante todo el rato, y solo cambia cuando el sistema está bastante seguro de la detección.

## 2.6— Sistema de resolución del cubo

Para el algoritmo de resolución hemos integrado el algoritmo de dos fases de Kociemba modificado para cumplir las necesidades de este trabajo en concreto. Es un algoritmo bastante conocido por obtener unos resultados con pocos movimientos en poco tiempo. Además, este algoritmo esta pensado para ser realizado por un computador, ya que se basa en unas tablas de heurística que no serían emulables por un ser humano. De esta forma, proporciona unos resultados bastante buenos que completan el cubo sin tener que recorrer los típicos pasos que tendría un algoritmo pensado para ser realizado a mano como podría ser: primero la cruz blanca, esquinas blancas, capa central, cruz ultima cara, reordenar vértices, terminar el cubo... En vez de eso el algoritmo funciona como su nombre indica en dos fases, aunque a nosotros nos devolverá directamente un listado de movimientos, sin tener que preocuparnos por los procesos internos que realizó para obtener el resultado.

### Primera Fase

La primera fase realiza movimientos simples para llevar el cubo al estado intermedio, en el cual se podrá aplicar la segunda fase. Este estado intermedio que podemos ver en la figura 2.4, se caracteriza por tener tanto la cara amarilla como la blanca resueltas y dos casillas rojas y naranjas opuestas colocadas correctamente, el resto es indiferente.

Para llegar a ese estado se necesitan como máximo 12 movimientos desde cualquier estado inicial. En un principio el sistema puede parecer ineficiente, dado que para llegar a este estado intermedio teniendo en cuenta los 18 posibles movimientos del cubo (cada movimiento, su inverso y el de 180 grados), hay un total de  $18^{12}$  (18 posibles movimientos elevado a 12 movimientos como máximo) estados sobre los que buscar en la primera fase.

Ningún algoritmo, ni siquiera uno basado en heurística como el A\* puede obtener ese resultado en un tiempo aceptable. Para solventar eso el algoritmo de Kociemba genera previamente unas tablas (en formato json) con la heurística exacta en todos los estados posibles de la fase uno. Si se tienen esos valores la búsqueda pasa a ser inmediata, solo necesita generar los 18 estados del cubo para cada uno de los movimientos y coger el de menor heurística entre ellos.

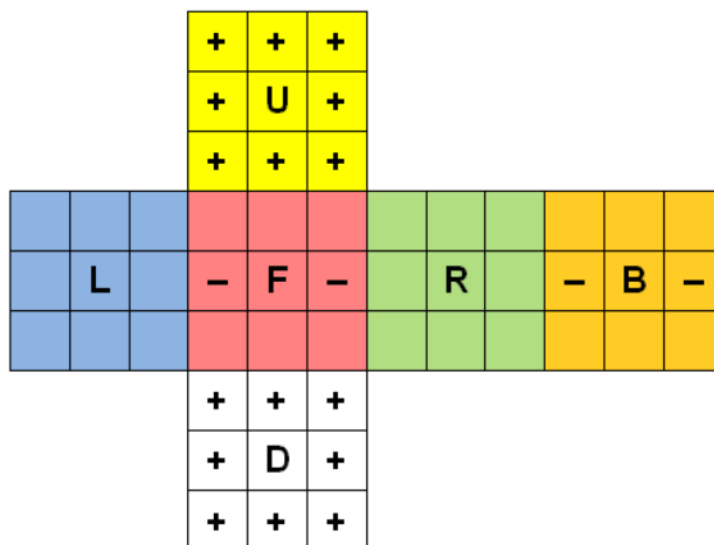


Figura 2.4: Estado Intermedio del algoritmo de Kociemba

### Segunda Fase

Gracias a la obtención del estado intermedio, la heurística de la segunda fase puede permitirse ser menos precisa y aun así no tardar demasiado tiempo. La segunda fase resuelve el cubo desde el estado obtenido de la primera fase, realizando únicamente movimientos hacia arriba, hacia abajo o dobles (de  $180^\circ$ ), con un máximo de 18 movimientos desde el peor de los estados posibles de la fase 1. La idea del algoritmo es combinar resultados subóptimos de la fase 1, que provoquen un buen estado intermedio para en la fase 2 (la cual puede ser más larga) tenga menos movimientos, dando un número total menor de movimientos. Un ejemplo sería: una fase 1 de 2 movimientos provoca una fase 2 de 15 movimientos, mientras que para ese mismo cubo, una fase 1 de 4 movimientos provoca una fase 2 de 6 movimientos, así que una subóptima fase 1 provoca un número total de movimientos menor, dado que reduce los movimientos necesarios de la fase 2.

### Rendimiento y resultados

Debido a que el algoritmo necesita generar una gran cantidad de tablas para tener la heurística de la primera fase, la primera vez que lo ejecutemos tardará un tiempo considerable, debido a que dichas tablas se tendrán que crear. Una vez generadas las tablas aparecerá un archivo *tables.json* en nuestra raíz del proyecto, cabe a destacar que ocupa sobre unos 20 MB y consta de unos 21 millones de caracteres.

Una vez se haya generado el archivo con las tablas, ya no será necesario generarlo de nuevo así que no consideraremos ese tiempo para analizar el rendimiento, ya que se trata de un proceso de preparación.

El algoritmo nos permite poner un timeout, tras el cual nos devolverá el resultado más corto encontrado, actualmente lo hemos definido en 10 segundos, así que básicamente el

algoritmo va encontrando soluciones y tras los 10 segundos de timeout devuelve la mejor de las encontradas.

Se podría aumentar el tiempo de timeout para encontrar así una mejor solución (si existiera) pero las pruebas realizadas no muestran una gran mejoría en el número de movimientos como para que merezca la pena hacerlo.



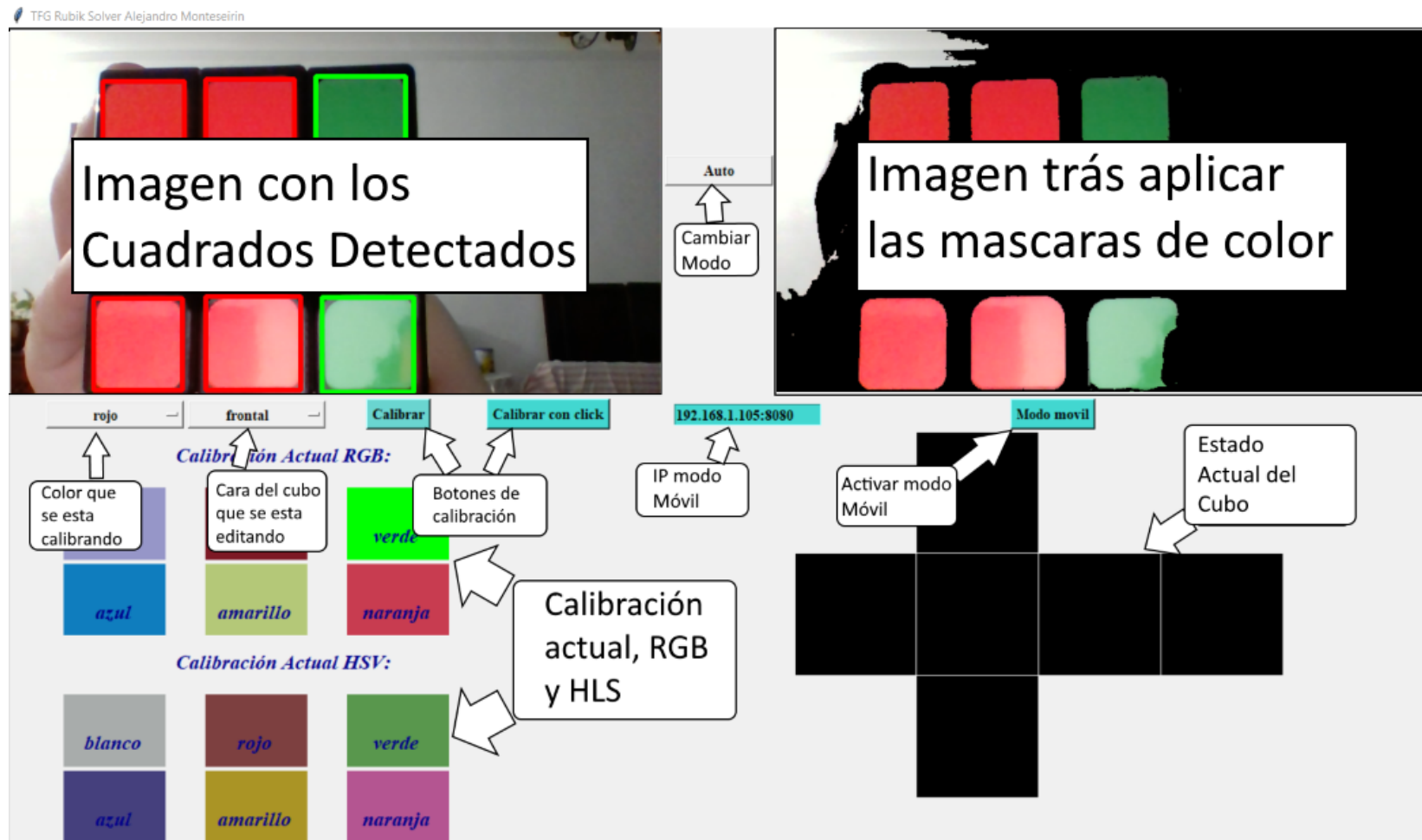


Figura 2.5: Interfaz y elementos

## 2.7– Interfaz de usuario, Funcionamiento

Para el apartado visual, nos hemos basado en TKINTER que es la librería más estandarizada de PYTHON para temas visuales. La interfaz recibe un flujo de 4 datos, consistente en:

- El frame original captado por la cámara, añadiendo la rejilla para el sistema de calibración junto a los contornos detectados en el modo auto y la cuadrícula 3x3 para éncajarél cubo en el modo manual.
- El frame modificado, con todas las máscaras de color aplicadas en el modo auto (es decir viendo solo lo que vería el programa encargado de detectar los contornos). Para generarlo aplicamos a la imagen recibida todas las máscaras de los distintos colores, cada máscara consisten en un rango en formato HSV, que se obtiene directamente de los valores actuales de la calibración (ya sean los valores predefinidos o los introducidos por el usuario). Una vez tenemos la máscara de cada color, la aplicamos a la imagen, dando como resultado solo los fragmentos de la imagen que se encuentran dentro del rango de algún color.

```

1 # Input
2 frameRGB #El frame recibido de la camara en formato RGB
3 frameHSV #El frame recibido de la camara en formato HSV
4 #-----
5 #Definimos el rango del color amarillo
6 amarillo = cv2.inRange(frameHSV, self.calibracionAuto['amarillo'][0], self.
    calibracionAuto['amarillo'][1])
7 #Definimos el rango del color naranja
8 naranja = cv2.inRange(frameHSV, self.calibracionAuto['naranja'][0], self.
    calibracionAuto['naranja'][1])
9 #El color rojo es especial, ya que debido a que el Hue va de 0 a 180, puede tener
    rangos al principio y al final, para controlar ese caso le ponemos 2 rangos
    en vez de 1
10 rojo1 = cv2.inRange(frameHSV, self.calibracionAuto['rojo1'][0], self.
    calibracionAuto['rojo1'][1])
11 rojo2 = cv2.inRange(frameHSV, self.calibracionAuto['rojo2'][0], self.
    calibracionAuto['rojo2'][1])
12 ...#Resto de colores...
13
14 #sumamos todas las mascaras para ver el resultado final en la pantalla de la
    derecha
15 res=cv2.bitwise_and(frameRGB, frameRGB, mask = amarillo+naranja+verde+blanco+azul
    +rojo1+rojo2)

```

Código 2.2: Obtención del frame modificado del modo auto, Fichero Video.py

Otra forma de recibir el frame modificado en caso del modo manual sería la cuadrícula 3x3 rellena con los colores elegidos, la cual se genera basándose en el array de colores elegido.

```

1 frame2 = cv2.flip( frame2, 1 ) #le damos la vuelta para que sea mas facil de
   encajar
2 frameRGB = cv2.flip( frameRGB, 1 ) # ya que sino tendria efecto espejo y seria
   confuso
3 for x in range (0,len(contours)): #para cada casilla la pintamos del color
   elegido
4     if self.arrayElegido[x][0]=='blanco':
5         cv2.drawContours(frame2, contours, x, (255, 255, 255), -1)
6     if self.arrayElegido[x][0]=='azul':
7         cv2.drawContours(frame2, contours, x, (40, 40, 80), -1)
8     ...
9     #Asi para los 6 colores

```

Código 2.3: Obtención del frame modificado del modo Manual, Fichero Video.py

- La cara resuelta del modo auto. Su valor siempre es *None* excepto cuando el modo auto detecta 9 polígonos con estabilidad. En ese caso se compone de los 9 colores de la cara obtenida. Si recibe un valor no *None* salta un popup para confirmar el valor de la cara, la cual se imprime generando una imagen con los 9 valores.
- Por último también recibe el array de colores elegido en el modo manual actual, es decir los colores elegidos por el modo manual en cada casilla. Este array será utilizado en el caso de pulsar el botón de guardar, que mostrara el mismo popup que el modo auto, generado de la misma manera, utilizando en vez de la cara resuelta del modo auto, este array de colores elegido.

En resumen, la interfaz se basa en una única pantalla principal, apoyada por distintos desplegables que muestra:

- El vídeo en directo recibido desde la web-cam o móvil.
- El vídeo en directo tras el procesado.
- La calibración de los colores, tanto en RGB como en HSV.
- El estado actual del cubo virtualizado.
- El resto de botones que nos permiten acceder al resto de elementos.
- Un desplegable de confirmación de cara (incluye otro desplegable al hacer click con el segundo botón del ratón en una casilla, que permite modifica el color de la casilla pulsada).
- Un desplegable de calibración mediante click.

## Generación de imágenes

Las imágenes generadas en la interfaz, como los popups o el cubo desmontado, se crean principalmente mediante una matriz bidimensional que contiene como clave la posición en el eje *Y* y en el eje *X* y como valor los valores rgb de cada punto (ej: *imagen[0,0]=[0,0,0]* para poner color negro en el punto 0,0). Utilizamos la funcionalidad slice de PYTHON para

colorear con mayor facilidad y rendimiento (en vez de punto por punto). Por ejemplo con: `imagen[0:100,0:50]=[0,0,0]` colorearíamos de negro todos los píxeles con *Y* entre 0 y 100, y *X* entre 0 y 50. De esta forma, usando `[:,:]` podríamos aplicar un cambio a todos los valores de cada eje.

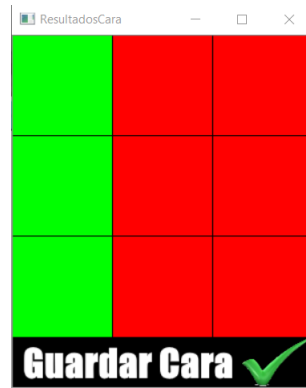


Figura 2.6: PopUp De la imagen obtenida

Pasamos a mostrar un fragmento de código, de como se genera el popup de la imagen obtenida una vez virtualizada una cara:

```

1 image=np.zeros((300, 300, 3), np.uint8) #generacion array tridimensional color
  negro
2 for index,color in enumerate(arriba): #coloreamos las tres casillas de arriba
  (100x100 cada casilla)
3     image[0:100,(index+1)*100-100:(index+1)*100] = color[3]
4 for index,color in enumerate(medio): #coloreamos las tres casillas de en medio
  (100x100 cada casilla)
5     image[100:200,(index+1)*100-100:(index+1)*100] = color[3]
6 for index,color in enumerate(abajo): #coloreamos las tres casillas de abajo (100
  x100 cada casilla)
7     image[200:300,(index+1)*100-100:(index+1)*100] = color[3]
8 #Generamos rallas separadoras para dar un poco de formato
9 image[99:100,:]= [0,0,0]
10 image[199:200,:]= [0,0,0]
11 image[:,99:100]= [0,0,0]
12 image[:,199:200]= [0,0,0]
13 image[:,299:300]= [0,0,0]
14 #cargamos el boton de guardar cara y lo unimos a la imagen de la cara que ya
  tenemos
15 im = PIL.Image.open("./Recursos/BotonGuardarCara.png")
16 np_im = np.array(im)
17 image= np.vstack([image,np_im])
18 #creamos la ventana y la ponemos en la posicion 500 30
19 window=cv2.namedWindow("ResultadosCara", 1)
20 cv2.moveWindow("ResultadosCara", 500,30)

```

Código 2.4: Generación de la imagen del PopUp ResultadosCara Fichero VentanaDesplegable.py

Una vez entendida la notación para generar imágenes, no es difícil usarla hasta generar un cubo “desplegado”, en el que iremos pintando las caras. Es interesante comentar que el array generado para pintar el cubo desplegado tiene un tamaño mayor o menor en función de la pantalla, con el objetivo de hacerlo sensible al tamaño del dispositivo.

Después de calcular el tamaño de la pantalla, se decide si generar un cubo desplegado grande o mediano y se genera totalmente vacío. Para ello lo pintamos con todas las caras de negro excepto las rallas separadoras de cada cara. Como la imagen real es un rectángulo, pintamos las zonas de dicho rectángulo que no pertenecen al cubo del mismo color que el fondo de la interfaz, dando así la sensación de que se muestra el cubo desplegado, en vez de un rectángulo con líneas separadoras. El resultado se muestra en la figura 2.7.

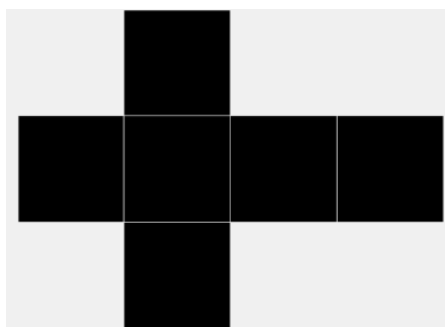


Figura 2.7: Cubo desmontado inicializado

Cuando virtualizemos la primera cara se generarán líneas auxiliares extra, que mostrarán en cada cara las distintas casillas, además de remarcar las líneas separadoras entre caras. Conforme vayamos virtualizando las caras el cubo comenzará a estar relleno con los datos introducidos hasta finalmente estar acabado.

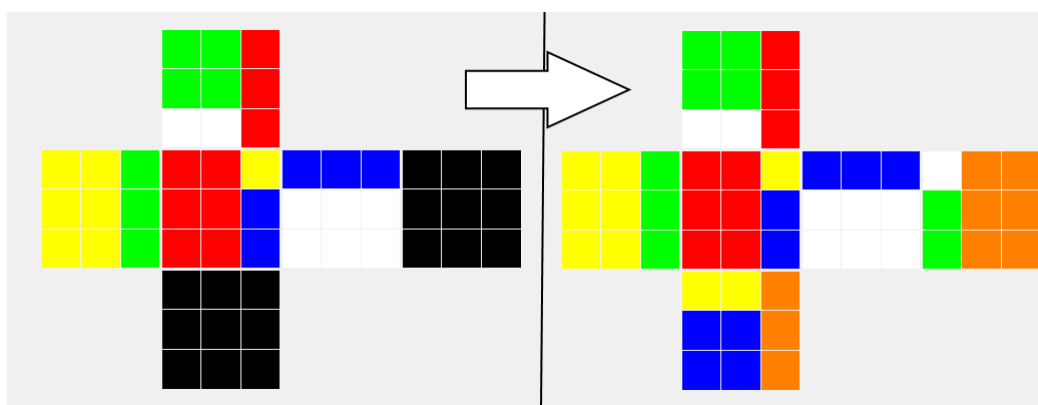


Figura 2.8: Proceso de virtualización del cubo

Finalmente, con todo el cubo virtualizado, aparecerá el botón de “Resolver Cubo”. Este botón traduce nuestros datos del cubo y los envía al algoritmo de Kociemba, el cual nos dará la respuesta más óptima que encuentre en un tiempo máximo de 10 segundos. Esta respuesta viene dada en la notación estándar de movimientos del cubo de Rubik, explicada en mayor profundidad en el manual de uso.



---

### Manual de uso

---

En este manual de uso se encuentran explicados todos los elementos de la interfaz y posteriormente se encuentran los pasos a seguir para utilizar la aplicación correctamente.

#### 3.1— Elementos de la interfaz

Pasamos ahora a explicar cada elemento de la interfaz, su título, una captura y una breve explicación de su funcionamiento:


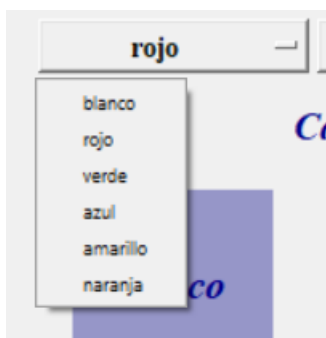
Botón cambio de modo
<div></div> <p>Situado en centro superior, entre los dos vídeos en directo se encuentra el botón de cambio de modo, el cual nos permite alternar entre los modos manual y automático. Al pulsarlo el siguiente fotograma recibido será de modo manual, y a su vez aparecerá el botón de guardar del modo manual.</p>

Tabla 3.1: Botón cambio de modo

## Selector de color



Situado a la izquierda de la pantalla principal de la interfaz, encontramos el selector de color para el sistema de calibración. Antes de calibrar cualquier color debemos seleccionar el color que estamos calibrando en dicho selector, así el sistema sabrá el color que estamos modificando con los distintos sistemas de calibración.

Tabla 3.2: Selector de color



<div><div>Sistema de Colores de la Calibración RGB</div><div><div><div>Calibración Actual RGB:</div><div><div><div>blanco</div><div>rojo</div><div>verde</div><div>azul</div><div>amarillo</div><div>naranja</div></div></div></div></div></div>
<p>Situado a la mitad de la parte izquierda de la pantalla principal de la interfaz, se encuentra el sistema de colores RGB, el cual muestra los valores actuales de cada color en el formato RGB, este sistema cambia en función de la calibración y permite ver a ojo si hemos calibrado correctamente los colores. Al iniciar la aplicación muestra los valores RGB por defecto.</p>

Tabla 3.3: Sistema de Colores de la Calibración RGB

Sistema de Colores de la Calibración HSV

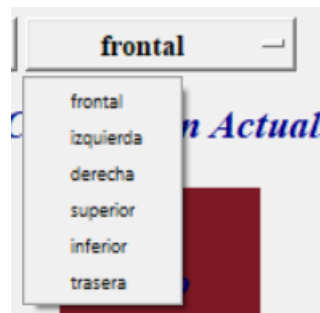
*Calibración Actual HSV:*

<i>blanco</i>	<i>rojo</i>	<i>verde</i>
<i>azul</i>	<i>amarillo</i>	<i>naranja</i>

Situado en la esquina inferior izquierda de la pantalla principal de la interfaz, encontramos el sistema de colores HSV, el cual, de manera similar al sistema de colores RGB, muestra los valores actuales de cada de cada color en el formato HSV. Esto permite a simple vista ver si ha habido algún tipo de desajuste en la calibración HSV con respecto a la RGB, lo cual puede llegar a ser común en algunos entornos de iluminación variante. Al iniciar la aplicación muestra los valores HSV por defecto.

Tabla 3.4: Sistema de Colores de la Calibración HSV

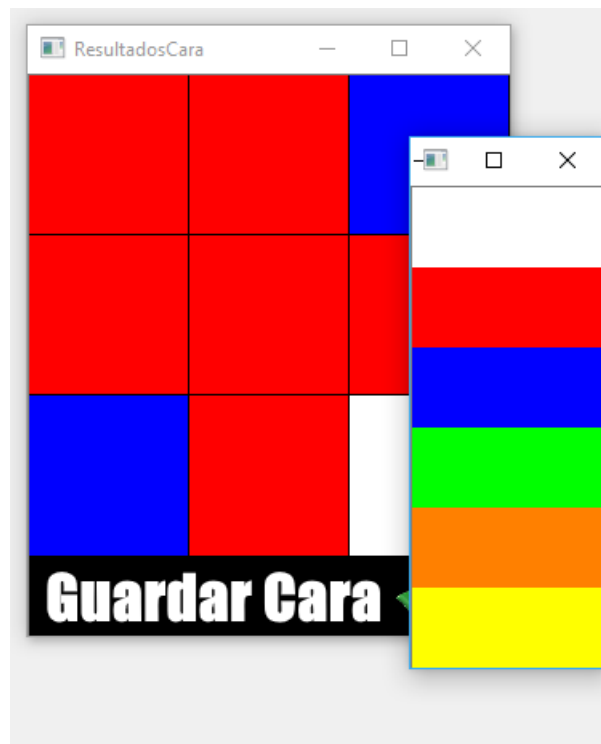
## Selector de cara



Situado a la izquierda de la pantalla principal de la interfaz, encontramos el selector de cara, el cual indica al sistema cual es la cara que estamos virtualizando ahora mismo. Esto significa que una vez virtualicemos la cara mediante el método manual o automático y confirmemos en el popup de guardar cara, en función del valor de este selector se guardará en una cara del cubo u otra, sobrescribiendo cualquier valor anterior.

Tabla 3.5: Selector de la cara del cubo

## PopUp De guardar cara + selector de modificación



Cada vez que virtualizemos una cara, ya sea con el botón de guardar cara del modo manual, o al encontrar las 9 casillas de un color estable en el modo auto, saltará el popup de guardar cara, para que confirmemos si los resultados nos convencen. En el caso de que haya algún error es posible pulsar en la casilla con el primer click del ratón para rotar el valor de esa casilla entre los distintos colores. También es posible pulsar la casilla con el segundo botón del ratón para desplegar otro popup con los distintos colores, pulsar cualquiera de ellos modificará la casilla a dicho color. Finalmente debemos pulsar el texto de “Guardar Cara” para confirmar y finalizar la virtualización de esa cara con esos datos.

Tabla 3.6: PopUp de guardar cara

### Botón de calibrar



Situado en la parte izquierda central de la interfaz, encontramos el botón de calibrar, la versión antigua del sistema de calibración con click[8]. Al pulsar este botón el color seleccionado en el selector de calibración se calibrará con respecto a la cuadrícula central de la imagen izquierda (en todo momento se muestra sobre la imagen izquierda una cuadrícula en el centro, en el modo auto, mientras que en el modo manual se utiliza la casilla central). La media de color de esa zona pasará a ser el valor de calibración de dicho color seleccionado en el selector de color.

Tabla 3.7: Botón de calibrar

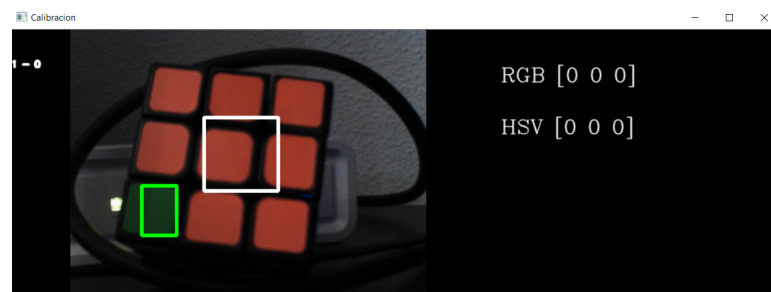
## Botón de calibrar con click



Situado en la parte central izquierda de la aplicación, encontramos la versión mejorada del botón de calibración, el cual en lugar de basarse en la cuadrícula de la imagen, desplegará el popup de calibrar con click. Desde este popup podremos realizar la calibración seleccionando con el ratón en vez de encajando la imagen del cubo en un área concreta.

Tabla 3.8: Botón de calibrar con click

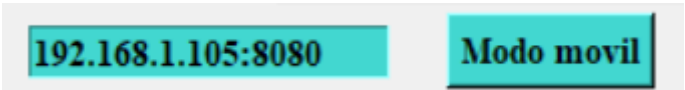
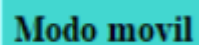
## PopUp Calibración con click



Este popup se desplegará cuando pulsemos el botón de calibración con click, y nos permite calibrar un color arrastrando el ratón en la casilla que deseamos y soltándolo. Esto generará un rectángulo que se usará de igual manera que la cuadrícula del botón de calibrar, lo que permite que sin mover el cubo, podamos calibrar todos los colores de una cara, simplemente cambiando en el selector de color el color a calibrar y posteriormente arrastrando el ratón en la imagen en la casilla correspondiente hasta obtener el valor deseado. Incluye además a la derecha los valores RGB y HSV en todo momento en el puntero del ratón, lo cual permite a los usuarios más avanzados seleccionar correctamente los colores que desean, así como ver si alguna casilla tendrá problemas debido a la iluminación.

Tabla 3.9: popUp de calibrar con click

## Ip y botón modo móvil

A screenshot of a software interface. It features a light gray background. In the center, there is a teal-colored rectangular input field containing the text "192.168.1.105:8080" in a black, sans-serif font.A screenshot of a software interface. It features a light gray background. On the right side, there is a teal-colored rectangular button with the text "Modo movil" in a black, sans-serif font.

Situado en la línea central de la aplicación encontramos el input de la IP del modo móvil y el botón para cambiar a dicho modo. Más adelante se explica con detalle como usar el modo móvil para conectar la aplicación a la cámara de un smartphone, de momento es necesario saber que en el input se ha de poner la IP en la que retransmitiremos el vídeo y el botón nos permite activar el modo móvil y debemos pulsarlo después de configurar todo el sistema.

Tabla 3.10: Input de IP y Botón de modo móvil



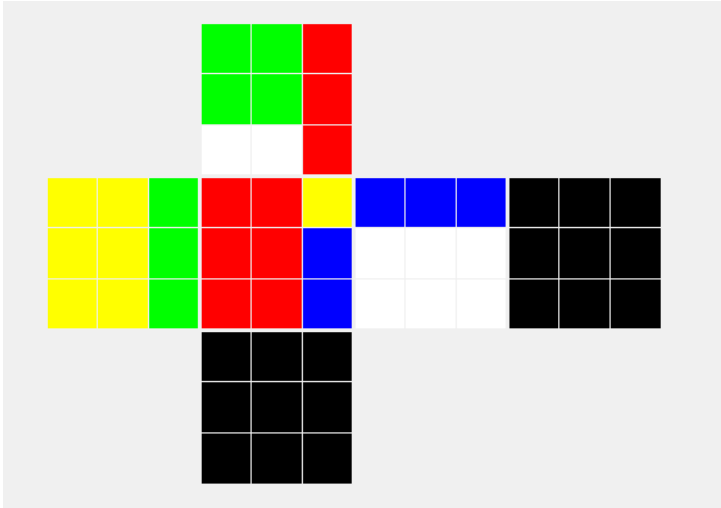
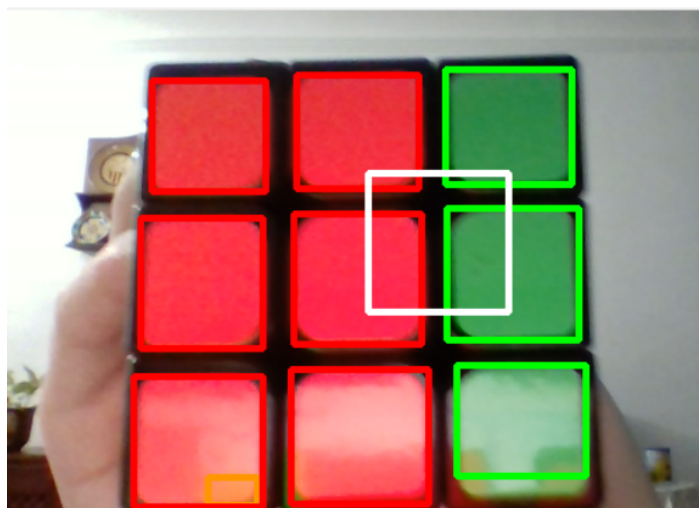
<div>Cubo virtualizado</div> <div></div> <div><p>En la esquina inferior derecha encontramos el cubo virtualizado con nuestros resultados. Conforme vayamos guardando las caras se ira rellenando con dichos datos, hasta tener al completo el cubo. Una vez este completo aparecerá el botón de resolver cubo, el cual nos permitirá obtener los pasos para resolverlo.</p></div>
--

Tabla 3.11: Cubo virtualizado

### Pantalla izquierda modo auto



La pantalla izquierda cambia en función del modo. En el modo **auto** se muestra la imagen y los cuadrados detectados en ese momento, que vienen dados en función del detector de contornos tras aplicar las máscaras de los distintos colores calibrados. Estos cuadrados se pintan con el color correspondiente. También se muestra en color blanco la cuadrícula que se tendrá en cuenta para la calibración manual, si pulsamos el botón de calibrar anteriormente explicado.

Tabla 3.12: Pantalla izquierda Modo Auto

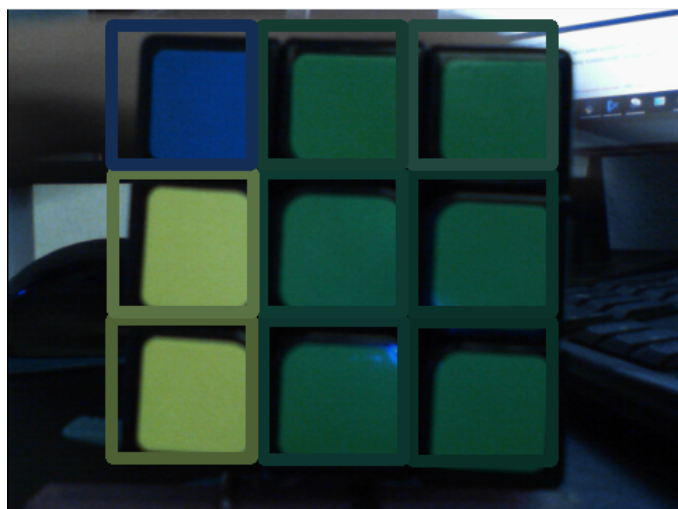
### Pantalla derecha modo auto



La pantalla derecha cambia en función del modo. En el modo **auto** se muestra la imagen después de haber aplicado todas las mascararas de color con los valores de la calibración. Esta imagen sería lo equivalente a lo que la aplicación ve para poder detectar los contornos. En la captura podemos apreciar que la esquina inferior derecha no es detectada completamente debido a la iluminación en ese punto, aunque con solo ese fragmento la aplicación es capaz de tenerlo en cuenta (en la pantalla izquierda se pinta la casilla). Es por eso que cuando vayamos a virtualizar el cubo nos tenemos que fijar en esta pantalla para ver si se detectan correctamente los colores o hace falta calibrarlos de nuevo.

Tabla 3.13: Pantalla derecha modo auto

## Pantalla izquierda modo manual



La pantalla izquierda cambia en función del modo. En este caso el modo **manual** muestra la imagen con una máscara superpuesta que incluye la cuadrícula. Cada casilla de la cuadrícula es del color medio de lo que se encuentra dentro de ella, lo cual permite saber si lo estamos colocando correctamente. A su vez la casilla central es la usada por el botón de calibración. Como podemos ver en la imagen el color de la casilla se corresponde aproximadamente con el valor del cubo, por lo que si la calibración es correcta los resultados deberían estar bien.

Tabla 3.14: Pantalla izquierda modo manual

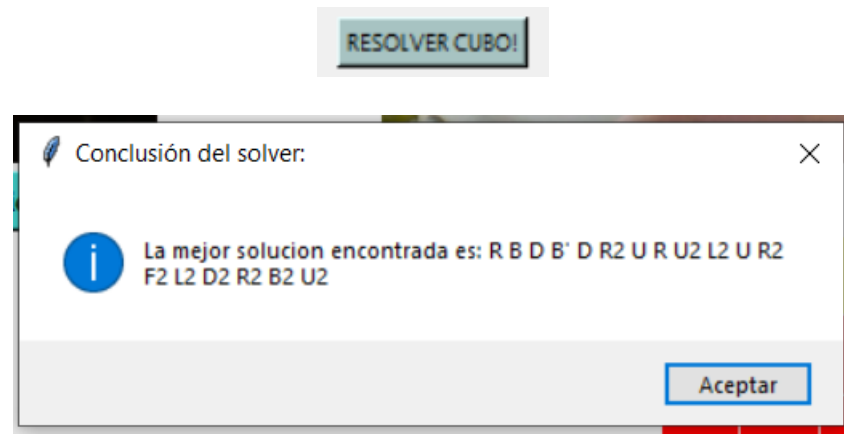
Pantalla derecha modo manual



La pantalla derecha del modo **manual** muestra los resultados sobre el cubo, el número de casilla y lo que se obtiene por cada detector. Si bien el color de arriba es el color detectado por el detector RGB, y el de abajo el detectado por el detector HSV, mientras ambos tengan un valor distinto el color no cambiará. Cuando por fin coincidan el color cambiará. Esto evita que los colores estén cambiando continuamente ya que es necesaria bastante precisión para que el color se detecte.

Tabla 3.15: Pantalla derecha modo manual

## Botón de resolver Cubo y resultados



Tras rellenar todos las caras del cubo, aparecerá bajo la pantalla derecha un nuevo botón, el botón de **Resolver cubo**, el cual tras pulsarlo enviará los datos del cubo al algoritmo de Kociemba. El algoritmo de kociemba recibirá los datos y se pondrá a resolver el cubo, si es la primera vez que se ejecuta, tardará más de la cuenta debido a que es necesario generar una serie de tablas para la inicialización, no obstante de no ser la primera vez tardará como máximo 10 segundos en devolvernos una solución en un PopUp siguiendo la notación típica del cubo de Rubik, si la seguimos paso a paso resolveremos el cubo satisfactoriamente.

Tabla 3.16: Botón de resolver Cubo y resultados

## 3.2– Preparación inicial

Si vamos a utilizar la virtualización es necesario que nos busquemos un entorno físico con una iluminación ambiente equilibrada, preferiblemente sin que toda la luz venga concentrada de un solo foco. Además debemos buscar un fondo de un color plano, preferiblemente gris o negro (si nuestro cubo tiene una cara blanca) o blanco (si nuestro cubo tiene cara negra en vez de blanca).

Antes de iniciar la aplicación conectamos el medio que vayamos a utilizar para grabar el vídeo y las imágenes, en el caso de utilizar una web-cam externa, con conectarla vía usb será suficiente, si usamos la web-cam integrada en un portátil no es necesario hacer nada.

### Configuración Modo móvil

En el caso de querer utilizar el modo móvil hay que realizar una preparación previa en el terminal. Es necesario instalar la app externa “IP Webcam” de Pavel Khlebovich o cualquier otra aplicación que envíe un flujo de imágenes a una dirección IP concreta.

Para que el móvil pueda comunicarse correctamente con la aplicación, es necesario que se encuentre en una red no demasiado restrictiva, como la red wifi privada personal del hogar. Debido a esta limitación es imposible utilizar la red wifi de la universidad *Eduroam*.

Sin embargo es posible crear una red privada utilizando un móvil que disponga del modo zona-wifi/compartir conexión (cualquier smartphone actual dispone de esa opción) y conectar el portátil con la aplicación a dicha zona wifi. Esto nos permite utilizar el modo móvil incluso careciendo de una infraestructura wifi, ya que el móvil se encarga de generar la red y la aplicación solo debe acceder a ella.

Finalmente en la aplicación móvil iniciamos la retransmisión del vídeo el cual nos pondrá en modo cámara y nos dará una IP. Esta IP tendrá que indicarse en la configuración de la aplicación y si accedemos a la ruta */photo.jpg* en la IP, obtendremos la imagen actual del teléfono.

De esta forma la aplicación es capaz de obtener un flujo de imágenes del móvil, mediante el acceso continuado a dicha IP. Al iniciar la aplicación hacemos coincidir la IP que muestra nuestro móvil con la IP de la interfaz (utilizando el campo de texto situado en el centro de la interfaz) y pulsamos el botón de modo móvil para realizar la conexión.

## 3.3– Fase de calibración

Colocamos el cubo en el sitio elegido, con fondo de color plano, y vamos realizando la calibración de cada color, utilizando preferiblemente el modo de calibración con click para evitar mover el cubo en exceso ya que cambiaría la iluminación.

Tras desplegar la interfaz del modo de calibración con click, seleccionamos el color que estamos calibrando en los selectores de la ventana principal, acto seguido arrastramos en la imagen el ratón, seleccionando la casilla del color a calibrar. Realizamos este paso con los distintos colores hasta que el sistema detecte satisfactoriamente todos los colores de la cara.

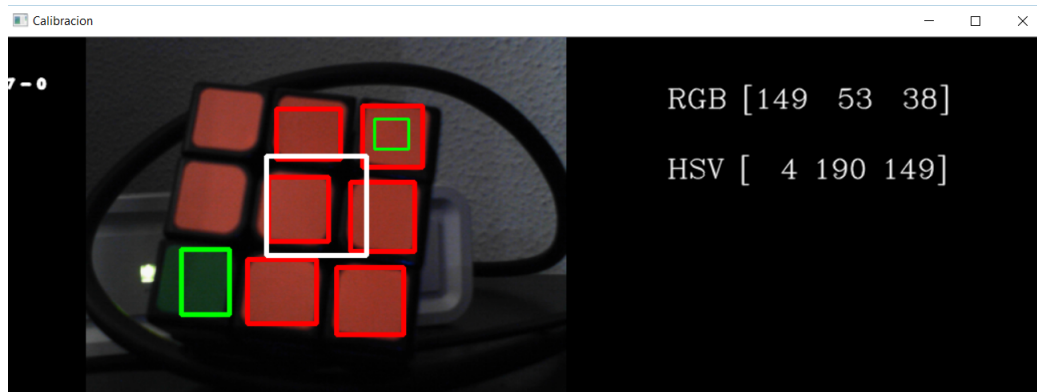


Figura 3.1: Sistema de calibración con click ventana desplegable, calibrando el rojo

### 3.4– Virtualización de las caras

Una vez realizada la calibración inicial, iniciamos la virtualización del modo auto, seleccionamos la cara que vamos a virtualizar y esperamos a que el modo auto detecte las 9 casillas. A continuación se abrirá el desplegable de confirmación el cual comprobamos. Si es necesario algún cambio pulsamos con el segundo botón para cambiar el color de la casilla al correcto. Una vez todo correcto le damos a guardar y ya tendríamos la primera cara virtualizada.

Puede darse el caso de que no se detecte correctamente alguna de las casillas, debido a cambios en el color desde la calibración. Para solucionarlo probamos usando el calibrador mediante click mientras se virtualiza. Si aun así las condiciones de luz u otros factores impiden realizar la virtualización podemos pasar al modo manual, cuadrar el cubo y guardarlo. Aunque la virtualización no sea correcta del todo, una vez aparezca el desplegable podemos modificar las casillas erróneas de ser necesario.

### 3.5– Resolver Cubo

Con el cubo entero ya virtualizado le damos un rápido repaso visual para comprobar que todo está en su sitio y pulsamos el botón de resolver cubo. Al cabo de unos instantes obtendremos la solución en la notación estándar del cubo de Rubik [12].



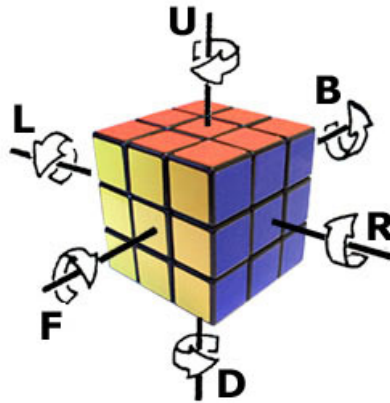


Figura 3.2: Notación cubo de Rubik

En la notación estándar del cubo de Rubik, cada letra indica un giro de una cara, en un principio en el sentido de las agujas del reloj. Si además incluye un apóstrofe, el giro sería en dirección contraria a las agujas del reloj.

Tras colocar la cara principal mirando hacia nosotros el resto de caras son:

- **B(Back)**: La cara opuesta a la frontal, es decir la de atrás del cubo
- **U(Up)**: La cara superior a la frontal
- **D(Down)**: La cara inferior a la frontal, en otros términos la opuesta a la cara superior.
- **L(Left)**: La cara directamente a la izquierda de la frontal
- **R(Right)**: La cara directamente a la derecha de la frontal

Teniendo en cuenta esto, la combinación de movimientos  $R^2 U'$  indicaría mover dos veces la cara derecha en el sentido de las agujas del reloj y 1 vez la cara superior en sentido contrario al de las agujas del reloj. Así que simplemente realizamos cuidadosamente los movimientos que nos devolvió el programa y tendremos nuestro cubo resuelto.

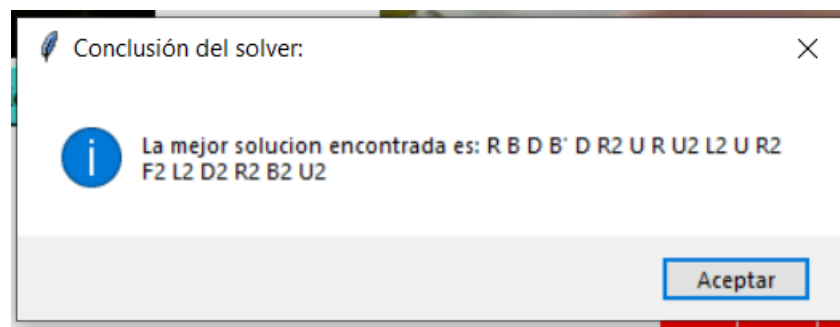


Figura 3.3: Resultado del solver

**Otras maneras de uso**

En el caso de no disponer de un entorno adecuado para la virtualización es viable utilizar el modo manual, encajando más o menos el cubo en la renjilla y después retocando los errores desde el popup de resultados. También sería posible ignorar la fase de calibración e introducir todos los valores a mano desde el modo manual, dando a guardar sin ni siquiera encajar el cubo y entonces desde el popup de resultados introducir todos los valores.

### Conclusiones

---

Inicialmente el trabajo iba a centrarse más en la parte de la resolución del cubo y dejar la virtualización como un paso secundario. A medida que el desarrollo iba avanzando, la fase de virtualización se fue ampliando más y conforme el primer prototipo del modo automático llegaba a ser funcional, era necesario explotarlo en profundidad para lograr un buen resultado. Es por ello que se vio más interesante centrar el proyecto en la virtualización y dejar la resolución del cubo como algo suplementario.

#### 4.1– Resultado Final

El resultado final es un prototipo perfectamente funcional de un virtualizador de cubos de Rubik, el cual con ayuda de una cámara puede registrar los datos del cubo y a partir de dichos datos resolverlo, por lo que consideramos que los objetivos del proyecto se han cumplido satisfactoriamente.

#### 4.2– Lecciones aprendidas

Durante el desarrollo del proyecto se han reforzado los conocimientos ya aprendidos durante el grado de PYTHON, sobre todo los relacionados con la interfaz en TKINTER y se han obtenido conocimientos nuevos, sobretodo en lo relativo al procesado de imágenes usando la librería OPENCV y al de NUMPY, dado que ambas librerías se usan extensivamente en todo el proyecto.

Se ha aprendido el uso de LaTeX ya que toda la documentación ha sido escrita en este sistema de composición de textos, utilizando su lenguaje para la creación de tablas, imágenes y figuras. Para facilitar el trabajo se ha usado la plataforma OVERLEAF, que permite un desarrollo de la documentación en la nube donde es fácilmente accesible para obtener retroalimentación del tutor.

### 4.3— Elementos a mejorar

#### Sistema de detección de colores y contorno

El sistema de detección de colores es probablemente la parte más interesante del proyecto y la que admite mayor número de mejoras. Probablemente el siguiente paso podría ser aplicar redes neuronales o algún tipo de machine learning para la detección de los colores de las caras.

#### Interfaz

La interfaz actual hecha en TKINTER nos permite integrarla fácilmente con PYTHON y nos otorga muchas funcionalidades, sin embargo en el diseño se ha dado prioridad a las funcionalidades sobre la estética, la cual siempre se puede mejorar. Además se podría hacer totalmente sensible al tamaño del dispositivo (actualmente la aplicación solo tiene 2 modos mediano y grande).

#### Algoritmo de resolución

Se podría mejorar tanto para mostrar el resultado, quizá con algún tipo de modelo en 3D que se vaya moviendo paso a paso; como para añadir nuevos tipos de algoritmos que nos den los resultados por partes, por ejemplo para usarlo para aprender a resolver el cubo (resolución paso a paso de la primera cara, cara intermedia, etc).

---

### Agradecimientos

---

Este trabajo no podría haber sido posible sin el apoyo de mis compañeros del grado, los cuales me han facilitado el aprendizaje tanto ayudándonos mutuamente como haciendo más amenos los momentos más duros, entre los que destacan por su apoyo **Camila, Mario, Pablo y Fuentes**.

A mi jefe durante las prácticas externas **Antonio**, por mostrar una gran flexibilidad y darme facilidades cuando mi horario coincidía con asuntos de la carrera.

A mi tutor **Francisco Jesús Martín Mateos** por atenderme en tutoría con ilusión, revisar la documentación con ahínco y lograr en definitiva hacer este proyecto posible.

También a mi grupo de amigos por su apoyo durante el grado, en especial a **Joseluis** que lleva conmigo desde los 12 años y esperamos que siga ahí indefinidamente.

Gracias a mi novia **Carolina** por todo el soporte durante la carrera y por ayudarme a desconectar cuando me hacía falta.

A mi familia, en especial a mis padres **Mercedes y Justo**, por permitirme realizar la carrera sin demasiadas preocupaciones extra y apoyar todas mis decisiones durante el grado.



---

## Bibliografía

---

- [1] H. Kociemba. *Kociemba Two Fase Algorithm Details*  
<http://kociemba.org/math/imptwophase.htm>
- [2] OPENCV. *Changing Colorspaces*  
[https://opencv-python-tutroals.readthedocs.io/en/latest/py\\_tutorials/py\\_imgproc/py\\_colorspaces/py\\_colorspaces.html](https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_colorspaces/py_colorspaces.html)
- [3] HACKSTER. *Colour Detection Using OpenCV and Python*  
<https://www.hackster.io/WolfxPac/colour-detection-using-opencv-python-8cbbe0>
- [4] STACKOVERFLOW. *OpenCV Calibración colores*  
<https://stackoverflow.com/questions/45926871/webcam-color-calibration-using-opencv>
- [5] SUMIT KUMAR MAITRA. *OpenCV Calibrador HSV en python*  
<https://piofthings.net/blog/opencv-baby-steps-4-building-a-hsv-calibrator>
- [6] VIKAS GUPTA. *OpenCV distintos usos de los espacios de color*  
<https://www.learnopencv.com/color-spaces-in-opencv-cpp-python/>
- [7] VIKAS GUPTA. *OpenCV distintos usos de los espacios de color proyecto github*  
<https://github.com/spmallick/learnopencv/blob/master/ColorSpaces/interactiveColorDetect.py>
- [8] GENNADIY NIKITIN. *Ejemplo de click y arrastre en tkinter*  
<https://gist.github.com/nikgens/c2f9d0dfba12f5e8789d9258d4538899>
- [9] PAUL GLASS. *Resolvedor cubo de rubik en python*  
<https://github.com/pglass/cube>
- [10] VICTOR CABEZAS. *Resolvedor Cubo de rubik en python mediante consola de comandos, proyecto github*  
<https://github.com/Wiston999/python-rubik>
- [11] TOM BEGLEY. *Implementacion del solver Herbert Kociemba's two-phase algorithm en python*  
<https://github.com/tcbegley/cube-solver>
- [12] RUBIKS FAMDON. *Notación cubo rubik*  
<https://rubiks.fandom.com/wiki/Notation>