

Visión Artificial. Reconocimiento de datos.

Introducción.

Para la práctica entregable de visión artificial he decidido enfocarme en la detección automática de objetos. Se ha elegido una problemática sencilla pero a mi parecer útil de resolver: ¿Cómo podemos detectar automáticamente qué número ha salido cuando lanzamos unos dados?

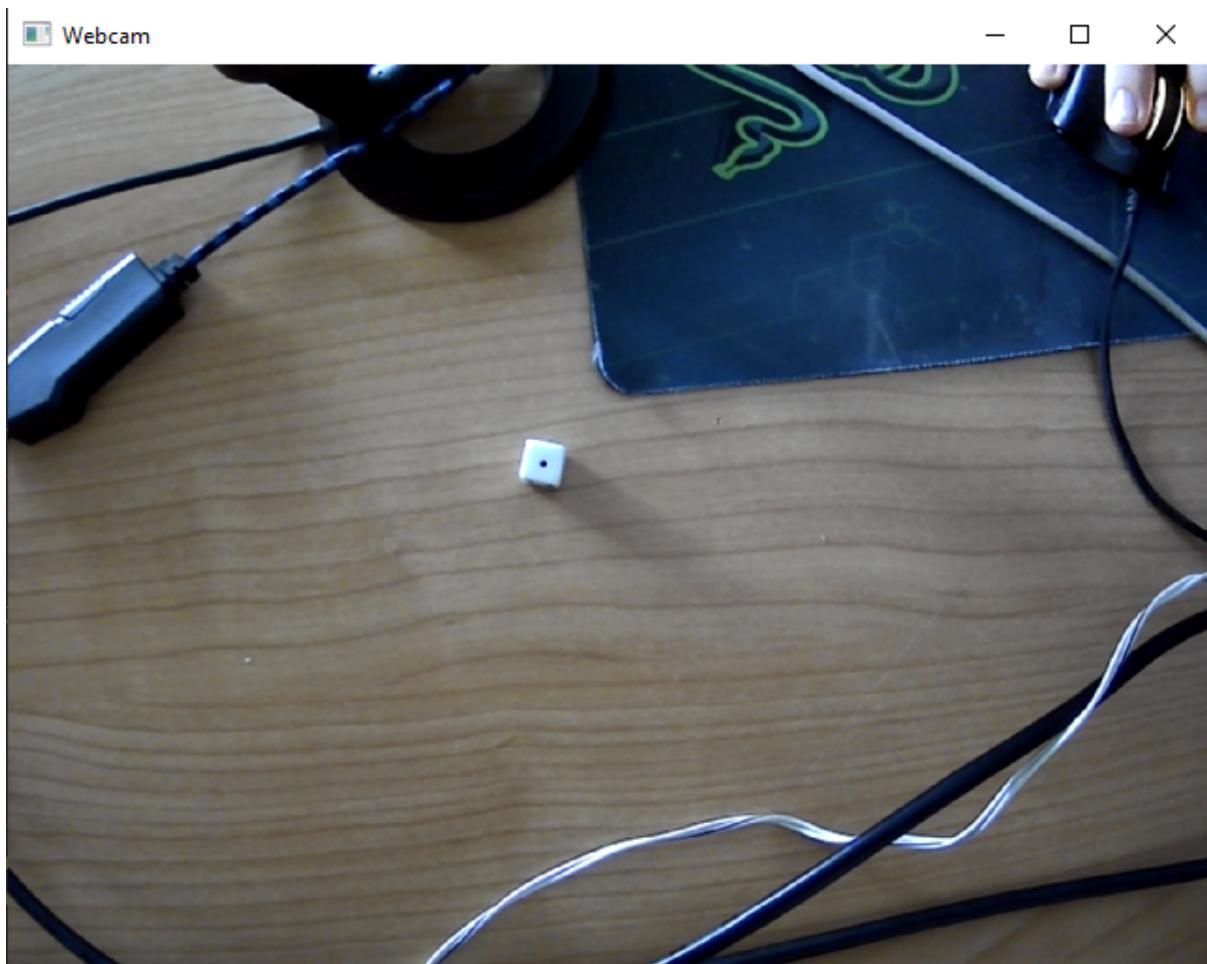
Realizar una aplicación que nos detecte el resultado de la tirada nos permitiría automatizar, por ejemplo, un sistema de arbitraje en juegos de azar etc.

Una forma de lograr este objetivo sería usar machine learning, con métodos como el “Cascade Classifier” de OpenCV para entrenar al programa y que aprenda a reconocer los resultados. El problema es que este método es muy costoso, requeriría mucho tiempo y un conjunto de imágenes positivas y negativas muy grande.

Así que se va a optar por resolver el problema utilizando métodos de tratamiento de imagen.

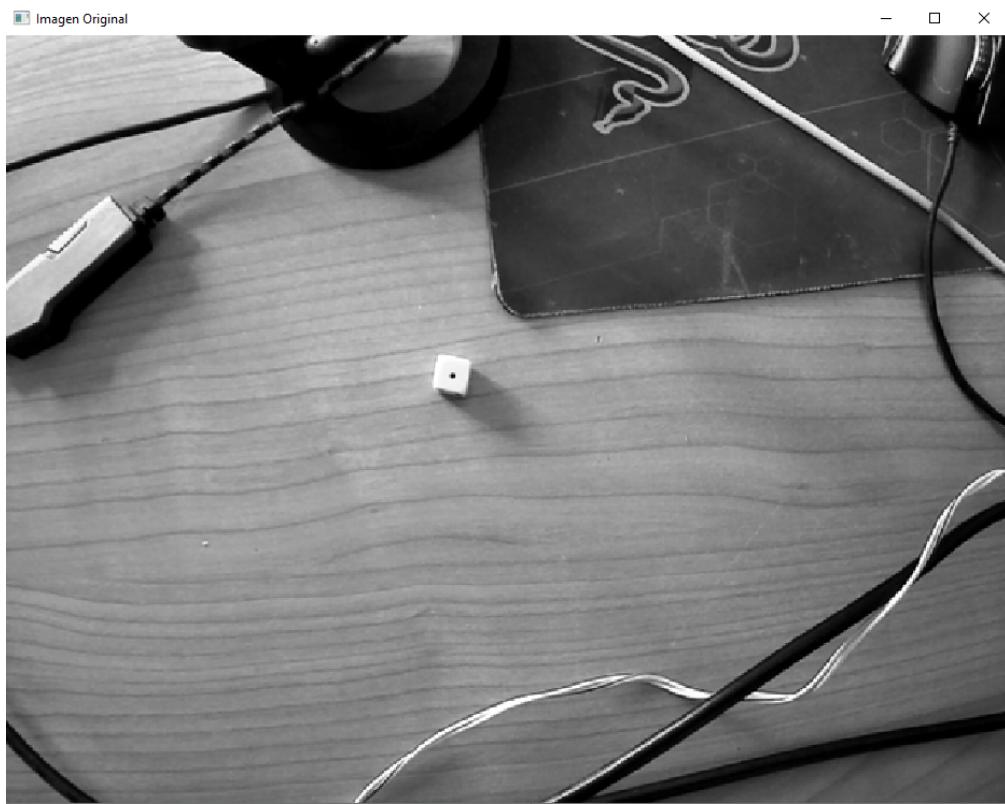
Proceso de desarrollo.

El primer paso a realizar será dotar al programa de imágenes de entrada. Para ello, escogemos abrir la cámara web de nuestro ordenador con opencv y capturar una imagen cuando el usuario presiona una tecla:



Elegimos, como escenario de prueba, un entorno con los suficientes objetos indeseados en pantalla para dificultar lo suficiente la detección, simulando que en la práctica, debemos evitar que el programa detecte como un dado cualquier cuerpo extraño que no lo sea.

El primer paso a realizar será convertir la imagen a una en escala de grises. Usaremos la función cv2.cvtColor(entrada, cv2.COLOR_BGR2GRAY) para ello.



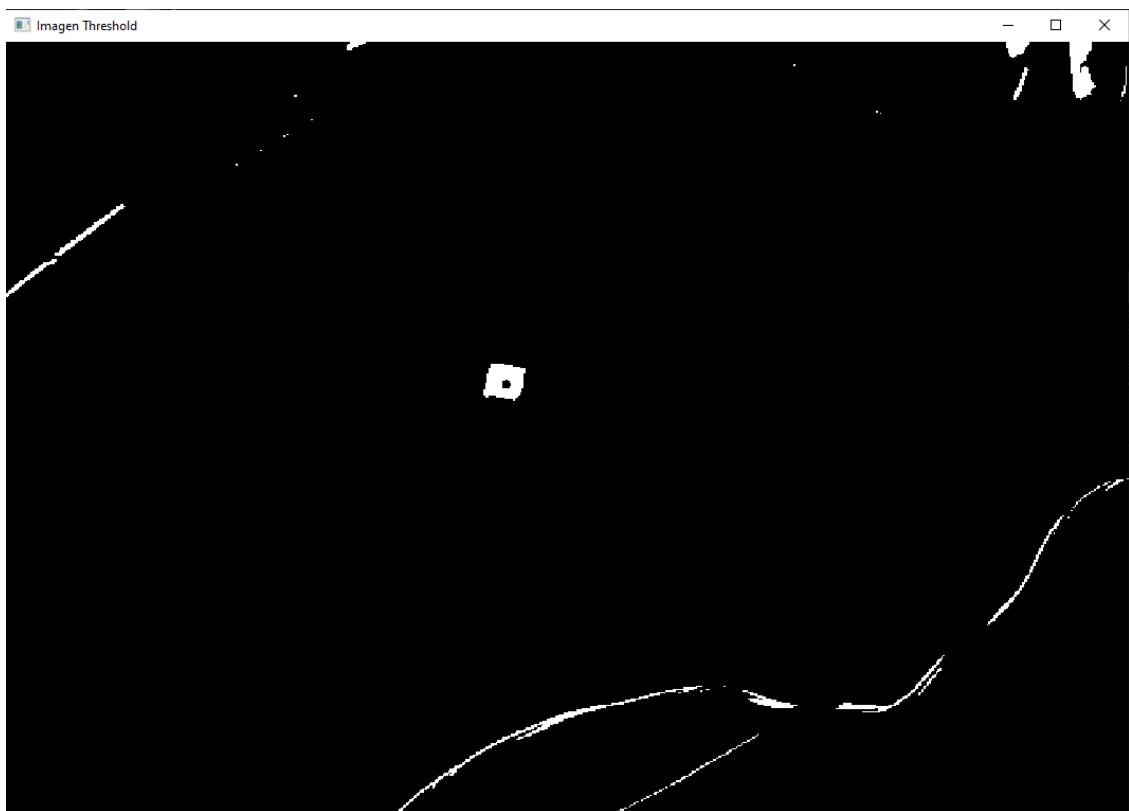
Una vez disponemos de nuestra imagen en escala de grises, probamos a aplicar un Threshold, para obtener nuestra imagen en colores binarios. Aquí encontramos la primera problemática, en función de la cantidad de luz que encontramos, nuestro Threshold cambiará mucho:



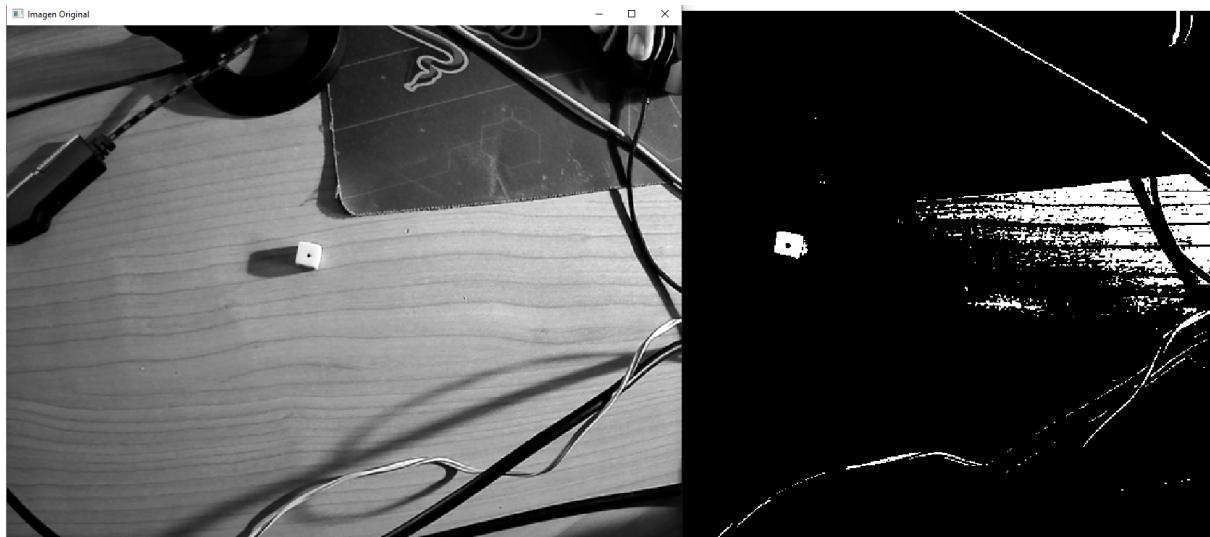
Si reducimos la la luz de la imagen:



Es prácticamente imposible detectar así el dado en ambientes bastante luminosos. Así que aprovecharemos que los dados tienden a ser blancos o de colores claros para subir el umbral:



Como idea, probamos aplicar un filtro Gaussiano y un Blur para reducir el ruido de la imagen para probar a ver si facilita el tratamiento:



Aplicar un suavizado solo conseguimos que perdamos información

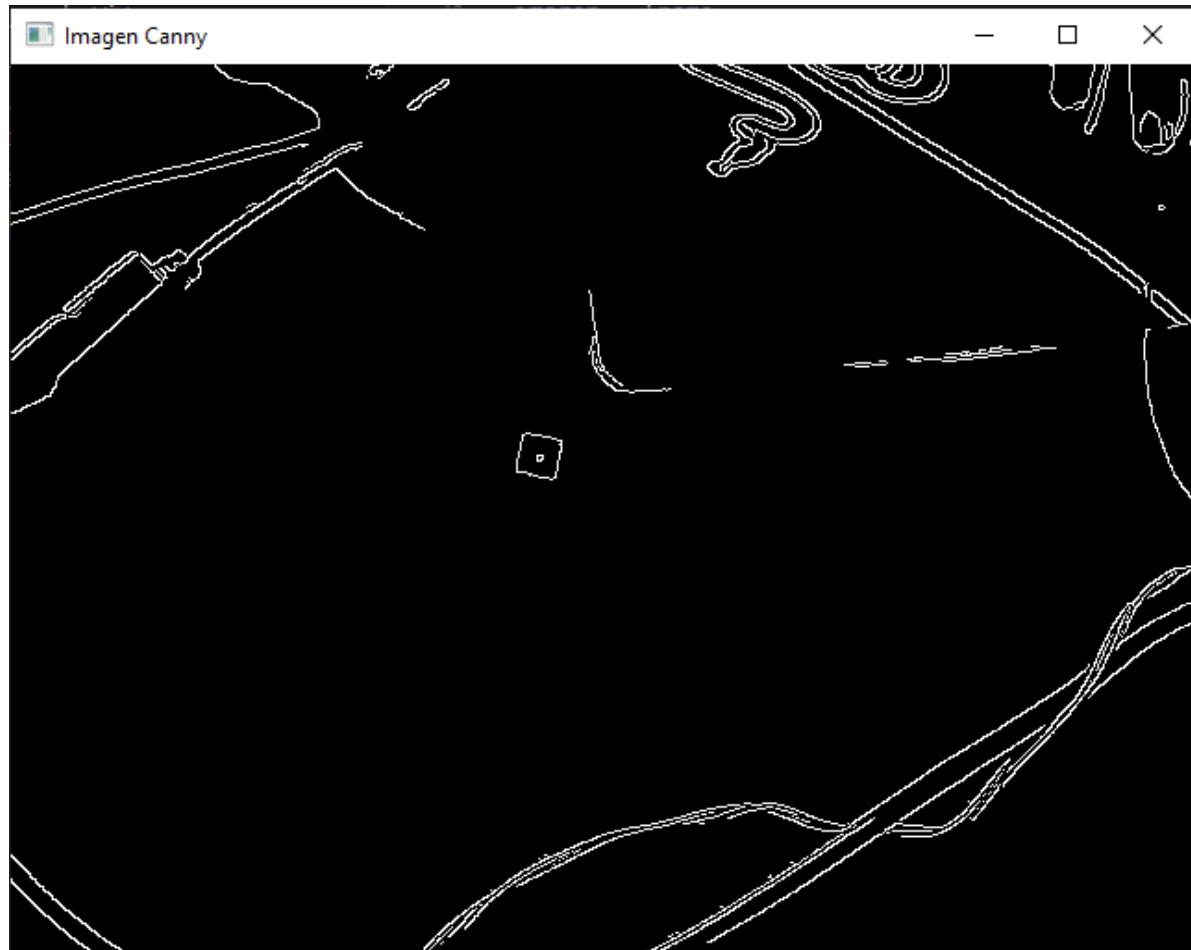


El siguiente paso es aplicar alguna técnica de detección de bordes y tras experimentar con Canny sobre varios tipos de Threshholds (diferentes umbrales) usados sobre varios conjuntos de imágenes de prueba y suavizados, vemos que en la práctica Canny se comporta mejor si lo aplicamos directamente sobre nuestra imagen en escala de grises:

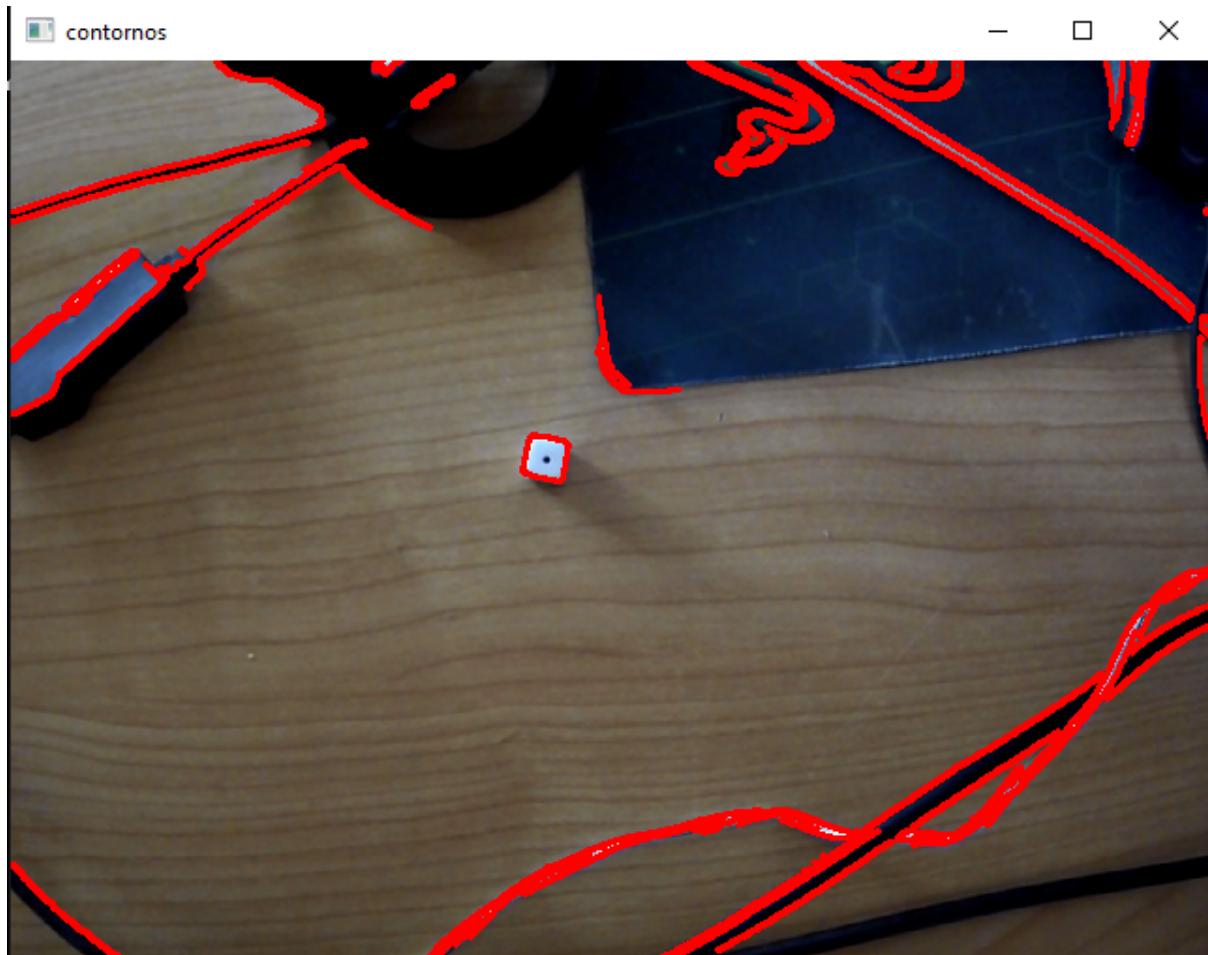
Se aplica el metodo de Canny para la detección de bordes, usando la función cv2.Canny():



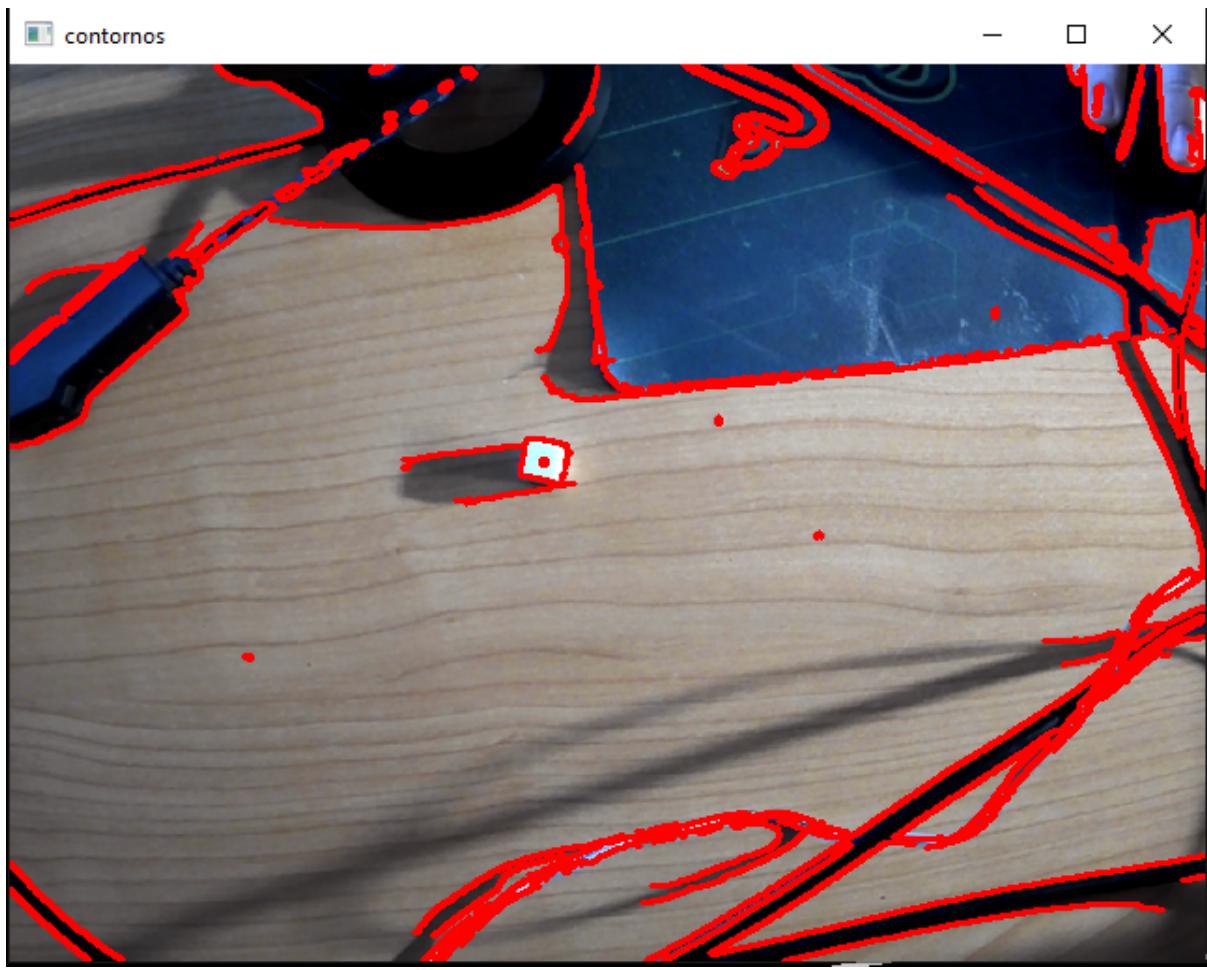
Observamos que hay detecta bordes incorrectos en los laterales de los dados, así que subimos el umbral al 200 para eliminar ruido innecesario:



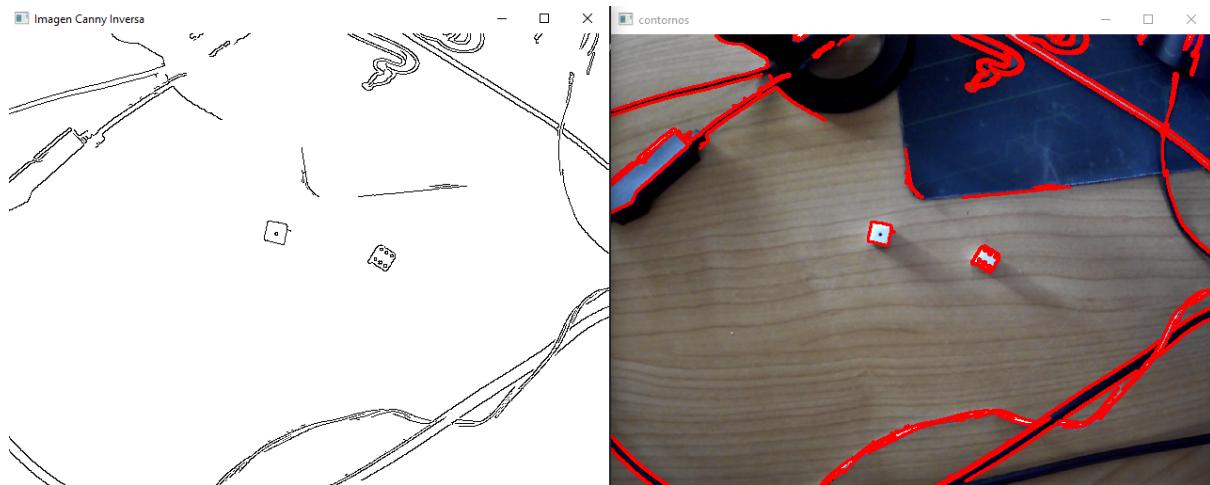
Ya que ahora sí disponemos de una representación donde se aprecian correctamente los contornos de los dados, usamos la función “cv2.findContours()” para encontrar los contornos sobre la imagen procesada con canny:



Podemos observar que detecta correctamente el dado, pero también detecta la gran mayoría de los contornos del resto de objetos de la imagen. Observamos también la gran problemática de nuestro enfoque, sin poder configurar un threshold o método de filtrado para hacer la imagen más uniforme de forma eficaz, la presencia de iluminaciones muy pronunciadas causa la detección del contorno de las sombras, no importa cuanto subamos el umbral (si no queremos que deje de detectar los dados).

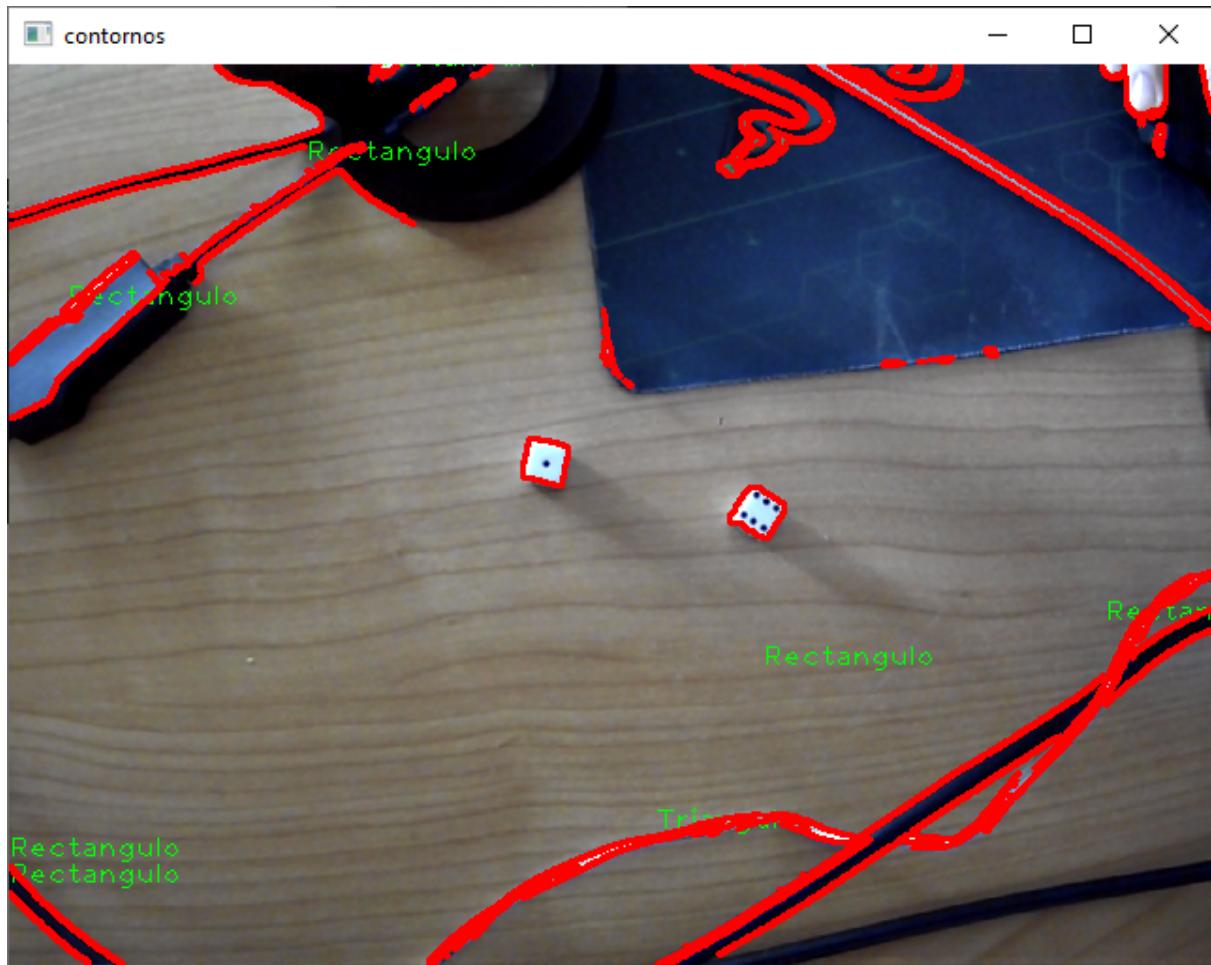


En cambio, con sombras más suaves en una iluminación menos exagerada, no tenemos ese problema:



Como alternativa, probamos a aplicar el método de Hough para la detección de líneas, visto en clase. El problema al intentar aplicar Hough, resulta en que las líneas no son totalmente rectas debido al ángulo de la captura de la imagen, entonces no encuentra la mayoría de los contornos de los dados.

Por tanto, seguimos usando Canny, aplicando la función de aproximar los contornos a figuras geométricas (`approxPolyDP()`), aprovechando que los dados son cuadrados o rectángulos vistos desde arriba:



Vemos que obtiene muchas figuras que no son realmente polígonos, pero qué el contorno de los dados no lo detecta como tal. También encontramos errores, probablemente por la detección de formas muy pequeñas. Por lo tanto tendremos que usar otro parámetro para filtrar todas estas figuras erróneas.

Decidimos usar el área de las figuras, combinada con los lados totales de las formas que encuentra, para determinar si el objeto es un dado o no. Además, cambiamos el valor de epsilon para que sea más tolerante a la hora de detectar los lados de las figuras a posibles curvaturas.

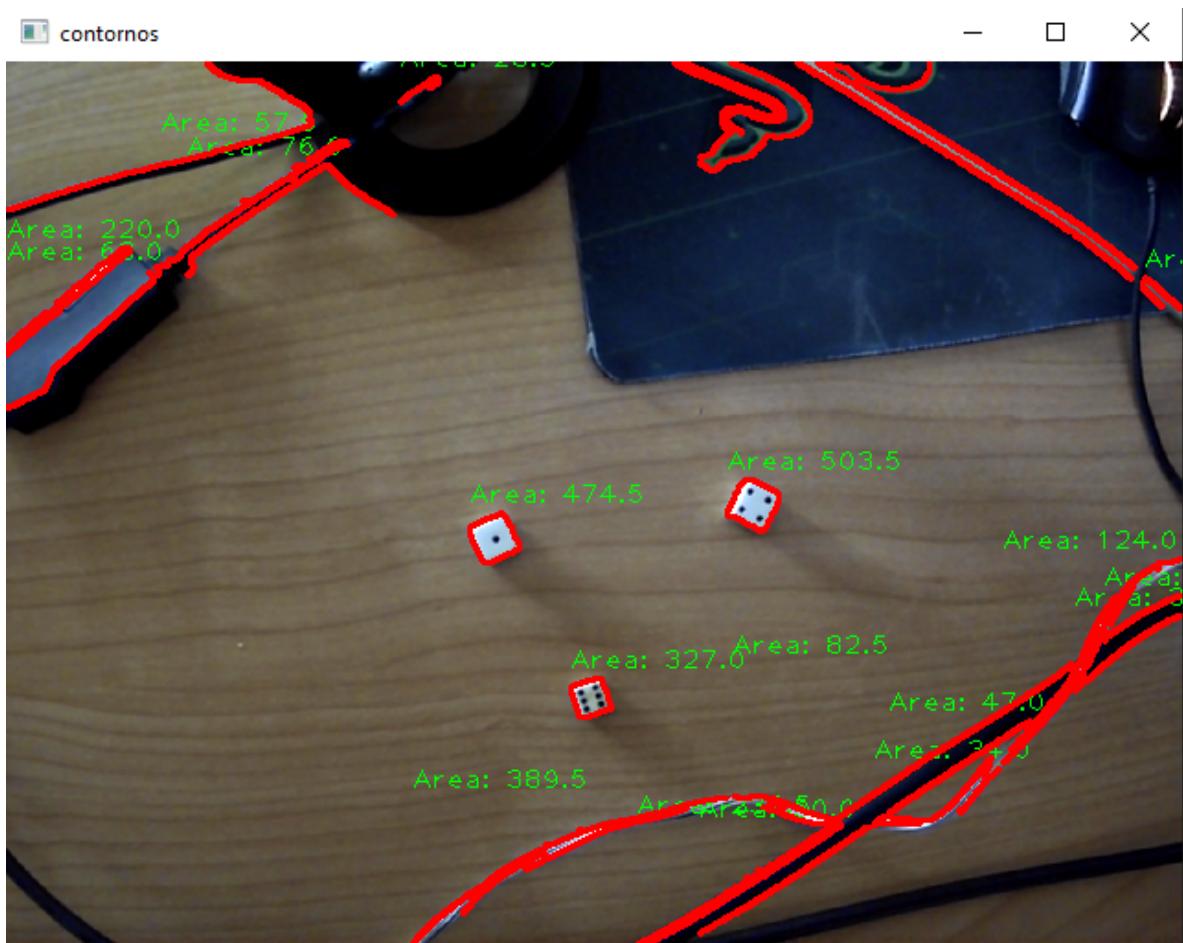
```

#Recorremos todos los contornos, con un factor epsilon relativamente laxo para detectar los polígonos. A partir de ahí, en función del área
#(para descartar objetos muy pequeños), y de los lados, nos quedamos con los cuadrados/rectángulos correspondientes.
for c in contornos:
    epsilon = 0.05*cv2.arcLength(c,True)
    approx = cv2.approxPolyDP(c,epsilon,True)
    print(len(approx))
    x,y,w,h = cv2.boundingRect(approx)
    #cv2.putText(imagen,'Area: '+str(cv2.contourArea(c)), (x,y-5),1,1,(0,255,0),1)
    if cv2.contourArea(c) >= 280 and len(approx) ==4:
        cv2.putText(imagen, 'Lados: ' +str(len(approx)) + ' Area: ' + str(cv2.contourArea(c)) , (x,y-5),1,1,(0,255,0),1)
        contornos_filtrados.append(c)

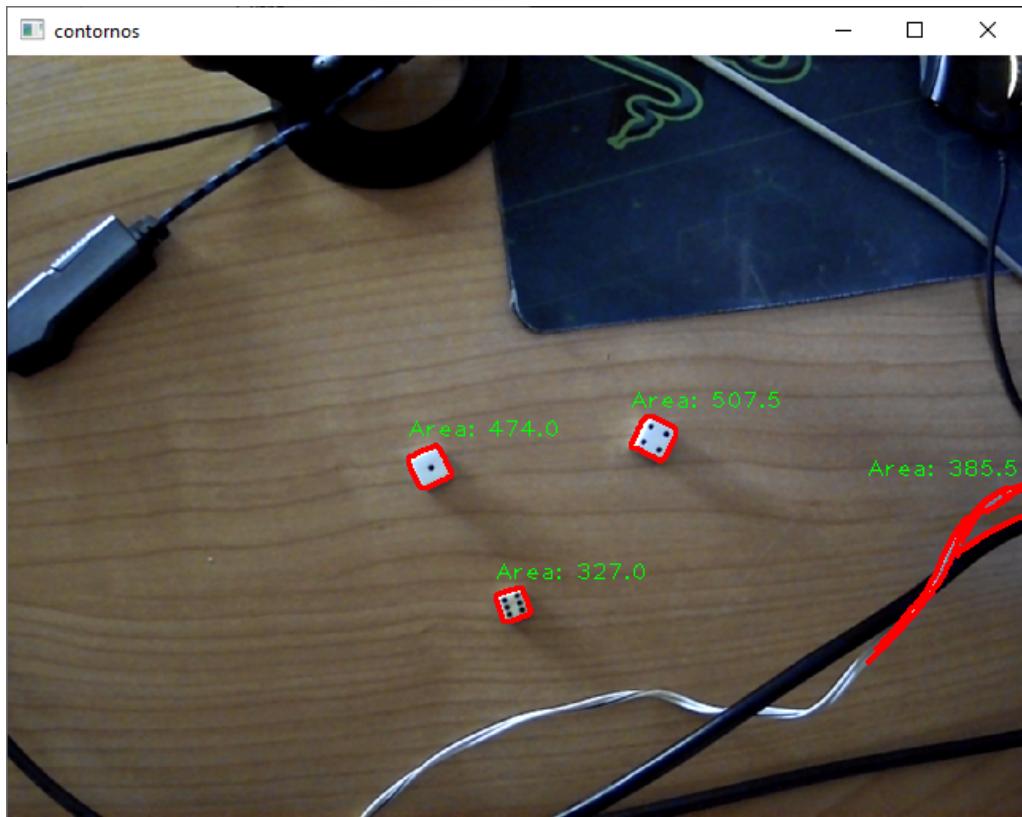
```

Para ello, elegimos un baremo de 280 para el área mínima de objeto que se considerará un dado, y le aplicamos el criterio de que además ha de tener cuatro lados.

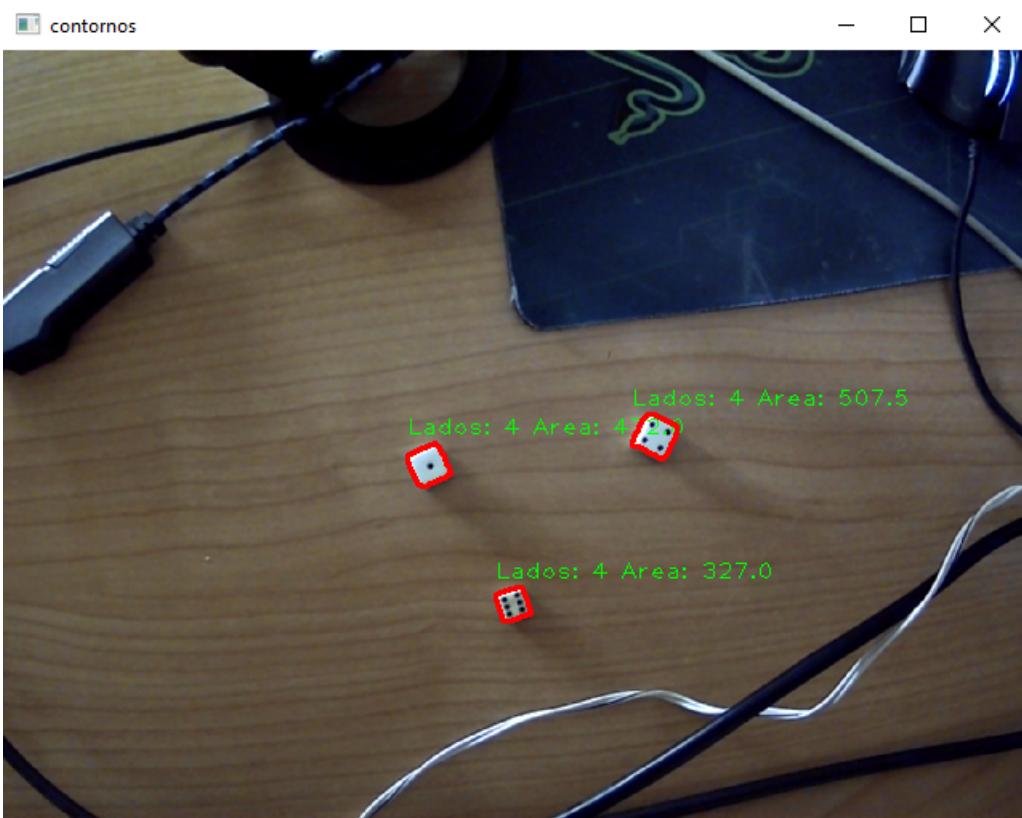
Vemos los resultados obtenidos al hacer pruebas para comprobar las áreas de los polígonos:



A continuación vemos el resultado de aplicar solamente un baremos de área mínima de 280:



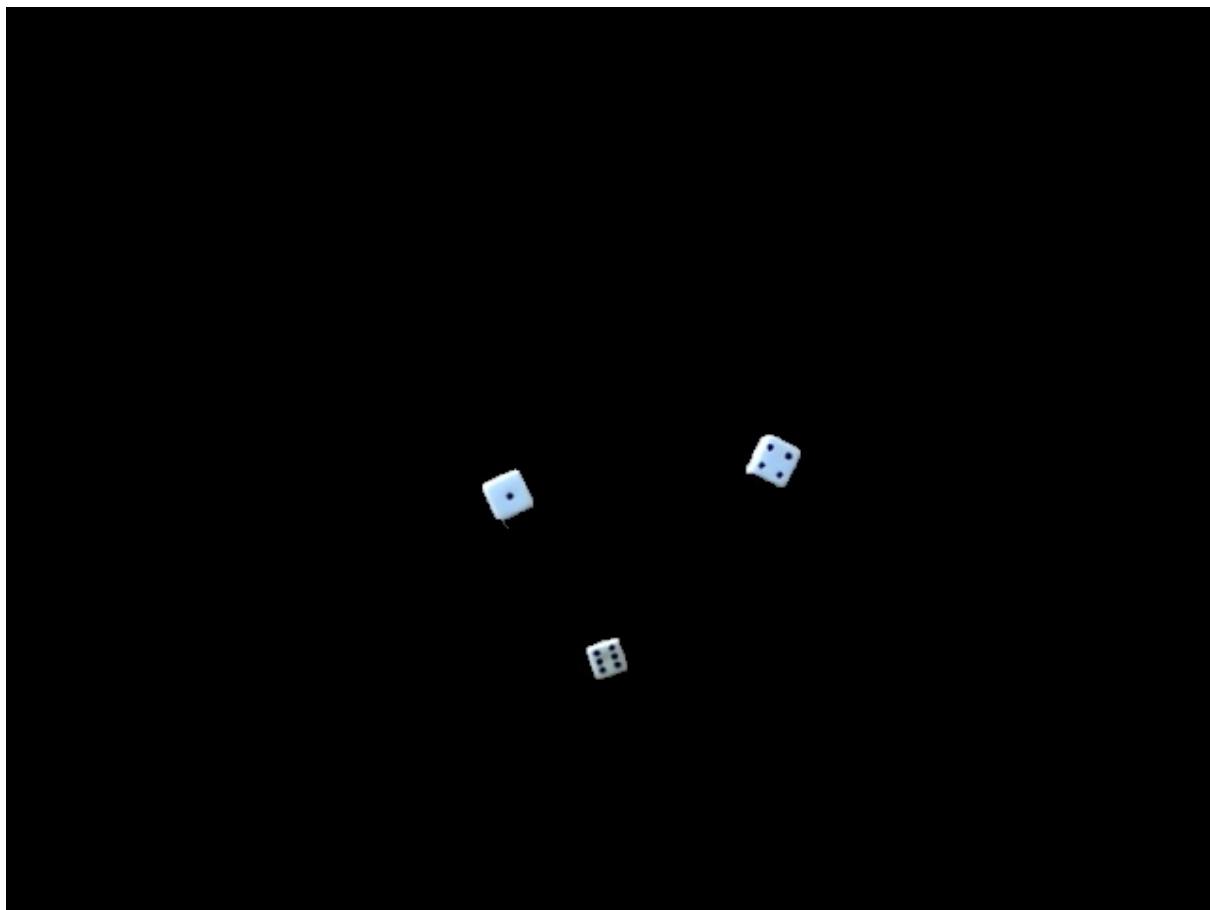
Y por último filtramos por su número de lados:



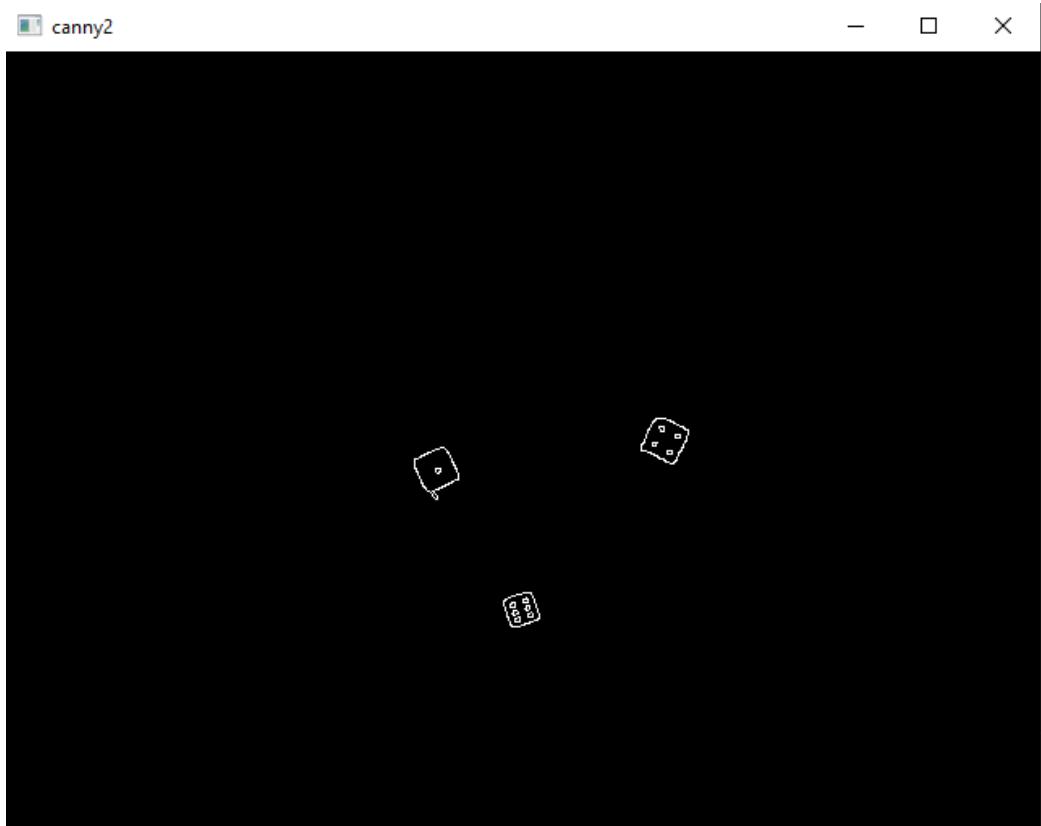
Ya disponemos de un programa que nos reconoce los dados que tiramos delante de la cámara, ahora solo necesitaríamos poder interpretar los resultados obtenidos con dichos lanzamientos. Para ello, se me ocurre como estrategia, obtener de nuevo los contornos de la figura, pero solo los suficientemente pequeños para corresponderse con los círculos de los dados.

El problema es que con esta aproximación, el programa encontrará muchas figuras como las que vimos anteriormente antes de ajustar las áreas.

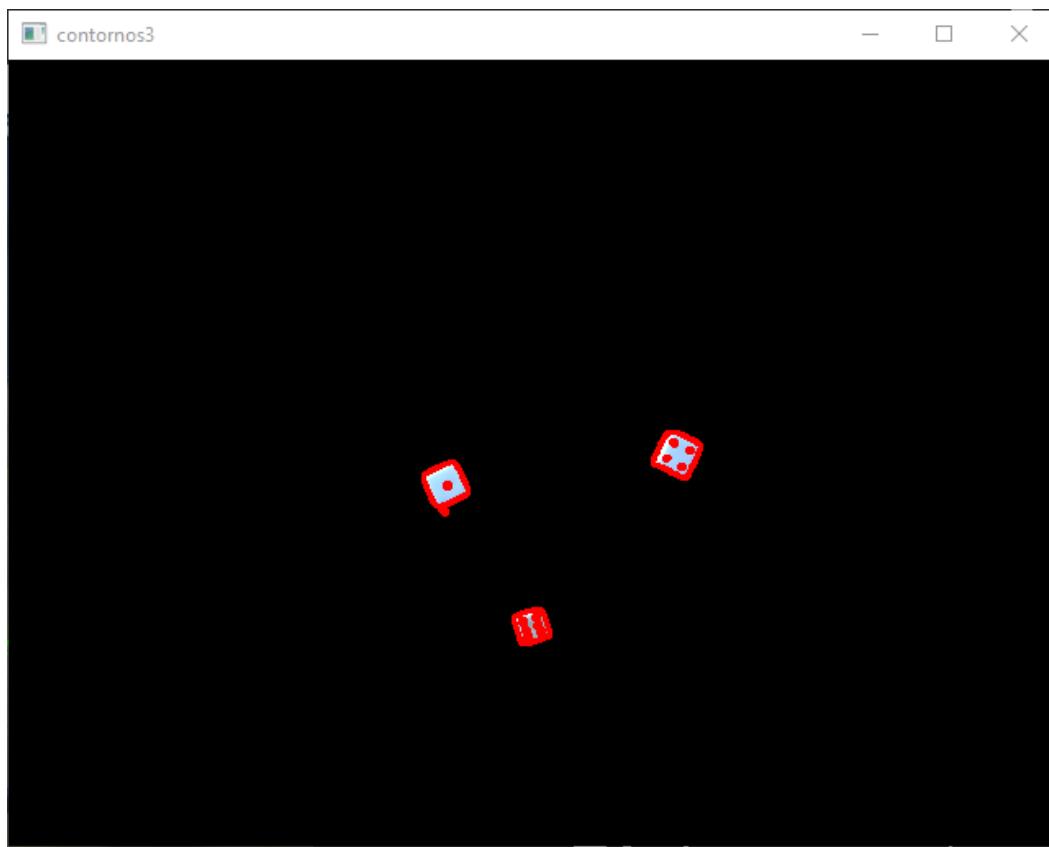
La solución elegida es guardar los dados encontrados en una nueva imagen del mismo tamaño que la original, ya filtrada sin elementos innecesarios, y repetir la detección de bordes:



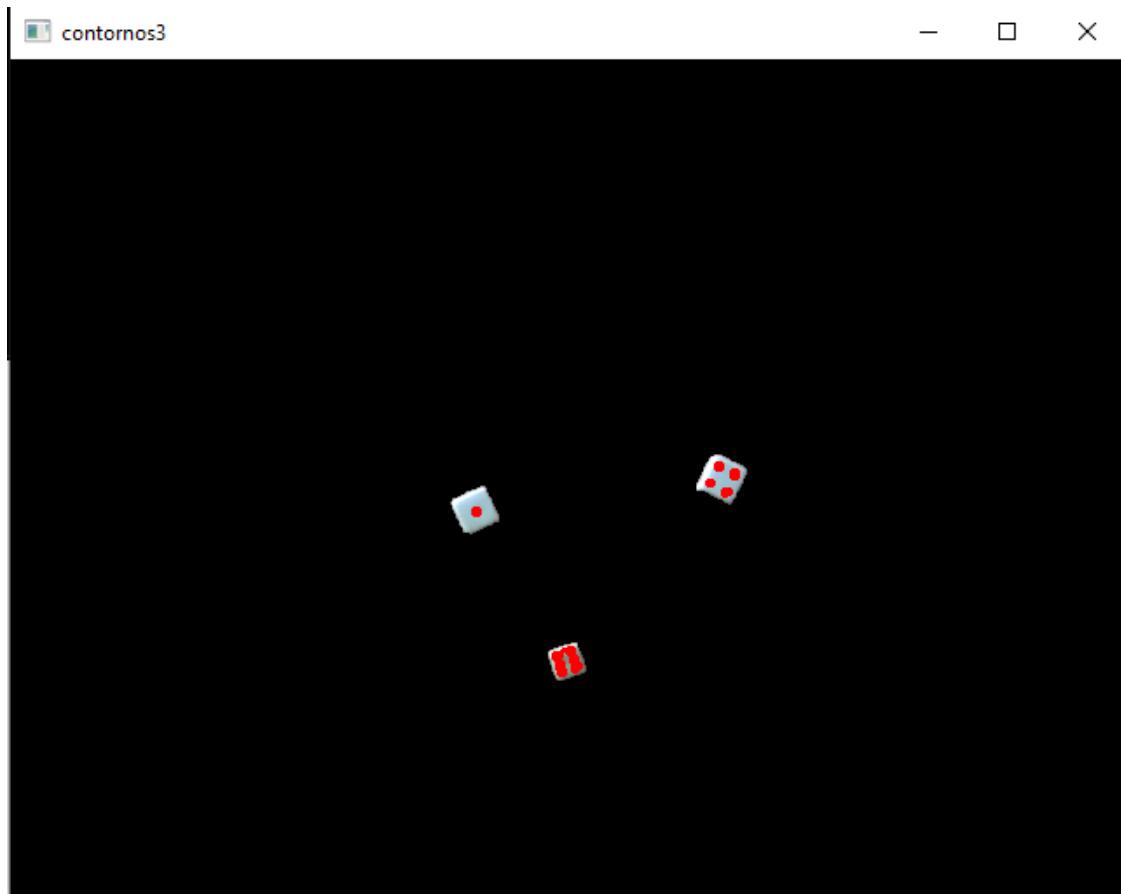
Una vez tenemos esta imagen, podemos volver a usar Canny para detectar los nuevos bordes:



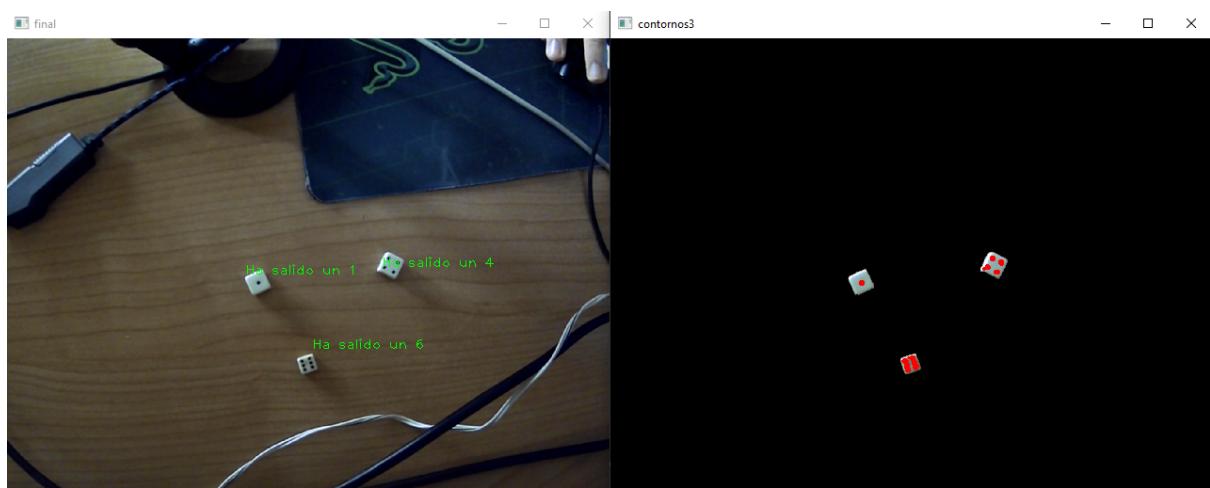
Y posteriormente los contornos:



Si en la función `findContours()` especificamos que queremos los contornos y la jerarquía que forman, podemos luego filtrar para obtener solo los internos:



Posteriormente, podemos calcular cuantos de estos contornos internos se encuentran dentro de las coordenadas de nuestros contornos externos, para calcular cuántos círculos tienen y obtener el resultado final:



Consideraciones

Habría que tener en cuenta qué, con la estrategia utilizada, el sistema podría encontrar dificultades para detectar datos anormalmente pequeños, pues no cumplirían el baremo mínimo elegido para su área. Por lo tanto, esa variable quedaría a discreción del sistema concreto donde se fuera a implementar el programa en la práctica. En función de la probabilidad de encontrarse objetos minúsculos cuyo contorno pudiera confundirse con el de un cuadrado, podría darse un valor más laxo al área mínima. Véase un ejemplo ilustrativo:

