Registro automático de una partida de go mediante visión por computador

Automatic recording of a game of Go through computer vision

Guillermo Siles Bonilla

Titulación: Grado en Ingeniería Informática.
Centro: Escuela Técnica Superior de Ingeniería Informática,
Universidad de Málaga.
Tutor: D. Javier González Jiménez.
Departamento: Ingeniería de Sistemas y Automática.

6 de febrero de 2017

Fecha de defensa:

El secretario del tribunal

Resumen

El objetivo de este trabajo fin de grado es construir una aplicación Android de forma que con un dispositivo móvil se analice el transcurso de una partida de Go[1] y genere un archivo que registre todos los movimientos realizados durante la partida. Posteriormente puede revisarse la partida en otra sección de la aplicación, que debe ser capaz de reproducir ésta sin ningún error.

La detección de las jugadas se realizará mediante técnicas de visión por computador, usando la librería OpenCV y procurando la mínima interacción posible entre el usuario y la aplicación. Será necesario preparar el sistema antes de empezar la partida, colocando el dispositivo móvil a una altura apropiada y calculando manualmente la posición de las esquinas para detectar el tablero.

Palabras clave

Visión por computador, procesamiento de imágenes, android, java, sistemas, automática, go, baduk, weiqi, AEGO, AMAGO, OpenCV, erosión, dilatación, Hough, circunferencias de Hough, grabación, suavizado gaussiano, correlación cruzada, homografía, perspectiva, lógica, ruido, detección, falso positivo, falso negativo, media, cámara, trípode, sgf, Lee Hajin.

[1] Juego oriental, originado en China aproximadamente 3000 años antes de Cristo. Actualmente el juego es muy popular en China, Corea y Japón.

Abstract

The objective of this work is to build an Android application that can analize a game of go with a mobile phone and generate an archive with the moves the players played during the game. Later on the game can be reviewed in other section of the application, that should be able to reproduce the game without errors.

The detection of moves will be achieved by computer vision using the OpenCV library, with the minimum interaction between the player and the application. It will be necessary to prepare the system before the game starts, setting the mobile at a proper height and manually calculating all the corners in order to detect the board.

Keywords

Computer vision, image processing, android, java, systems, automatics, go, baduk, weiqi, AEGO, AMAGO, OpenCV, erode, dilate, Hough, Hough circunferences, recording, gaussian blur, cross correlation, homography, perspective, logic, noise, detection, false positive, false negative, mean, camera, tripod, sgf, Lee Hajin.

Índice

Índice de figuras

1. Introducción

El Go[2] es el juego de mesa más antiguo y complejo que se conoce en el mundo. Se originó en China hace más de 4.000 años. Este juego en los países orientales es mucho más popular que el ajedrez en Occidente, hasta el punto que en Corea, los mejores jugadores de Go son tan famosos allí como muchos deportistas en España. De hecho, en estos países más que un juego, se considera un arte. Cuando dos jugadores profesionales juegan una partida con mucho tiempo para pensar, la partida resultante suele considerarse una *obra* de ambos jugadores.

1.1. Reglas de juego

Las reglas de juego son muy simples. Es un juego que se aprende en diez minutos y se tarda toda la vida en dominar.

Figura 1: Tablero de go con una partida en curso.

En la figura 1 tenemos una partida de go en curso. Cada jugador controla un color. Como se observa, las piedras[3] se juegan en las intersecciones.

[2]Go en Japón, baduk en Korea y weiqi en China.

[3]A las 'fichas' se les llama piedras. Actualmente la fabricación de piedras y tableros de go está en auge, llegándose a vender conjuntos por más de mil euros. Se usan materiales

Comienza jugando el jugador negro. Puede jugar una piedra en una intersección o pasar. Una vez termina su turno, el jugador blanco tiene las mismas posibilidades y, al terminar su turno, vuelve a tocarle al jugador negro. El juego termina cuando ambos jugadores pasan o cuando uno se rinde. Una vez ambos pasan, se cuenta los puntos que tiene cada uno y se determina el ganador.

1.1.1. Puntuación de una partida de Go

El objetivo del juego es conseguir más puntos que el contrario. Los puntos se consiguen rodeando territorio[4] o bien capturando piedras del contrario (cada piedra capturada se considera un punto). Cuando se capturan piedras éstas se retiran del tablero hasta el final de la partida.

Como anécdota, hay varios estilos de juegos, orientados a conseguir inicialmente territorio o influencia (territorio sería el equivalente a material en ajedrez e influencia a una buena posición). Actualmente los jugadores suelen jugar a conseguir más territorio que el contrario y luego intentar que la influencia conseguida por el contrario sea inútil.

Paradójicamente, alphaGo[5] tiene un estilo orientado a conseguir influencia, hasta el punto de hacer jugadas que todos los profesionales consideraban malas, demostrándose brillantes al avanzar la partida.

1.1.2. Captura de piedras

Para explicar la captura de piedras usaré un símil que me parece bastante acertado, si bien un poco infantil, pero creo que es el mejor que he encontrado hasta ahora para explicar de forma sencilla como funciona la conexión de piedras y la captura, dos en uno.

---

diversos como vidrio, conchas marinas, pizarra... En el caso del tablero el material estrella es la madera del árbol Kaya, muy escasa.

[4]Cada intersección del tablero cuenta como un punto de territorio. En la figura 1 con casi toda probabilidad la zona de abajo centro será territorio negro y la zona abajo derecha será de blanco, incluyendo las dos negras de la esquina inferior derecha que probablemente serán capturadas por blanco.

[5]Llamado asi el sistema creado por Google que retó y venció 4 - 1 a uno de los mejores jugadores de Go actuales, Lee Sedol. Después de la serie de partidas, Lee Sedol tuvo una racha impresionante de partidas ganadas. Su motivación para seguir jugando se vio reforzada al sentir todo un nuevo abanico de posibilidades.

Podemos pensar que cada piedra es un 'soldado'. Los soldados necesitan 'comida' para sobrevivir, que recogen de las intersecciones adyacentes. Sin embargo, si hay un enemigo en una intersección adyacente, no podrá coger comida de ese lugar, pues está invadido por el enemigo.

Pues bien, para capturar una piedra (o grupo de piedras) hay que dejarla sin comida. Vemos un ejemplo ilustrativo en la figura 2 .

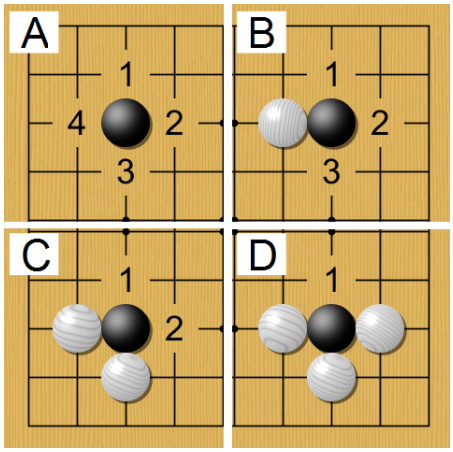

Figura 2: Captura de piedras.

Vemos que conforme blanco juega piedras alrededor, negro empieza a perder sus granjas, hasta que solo le queda una en D. Si blanco juega una piedra en 1, capturará la piedra negra, teniendo que retirarla del tablero (obligatoriamente).

Si negro quiere evitar la captura de su piedra, debe jugar él en ese lugar, para crear un 'grupo de soldados'.

Un grupo de soldados 'comparten la comida' entre ellos. Por tanto, si negro jugase en 1 para salvar su piedra, ahora serían un grupo de dos soldados, peleando contra tres soldados blancos y cogiendo comida de tres granjas. Para que blanco pudiese capturar este grupo necesitaría tres movimientos extras, uno en cada granja.

Esto complica muchísimo el juego. Se pueden formar grupos de soldados 'inmortales[6]', situaciones en las que dos grupos de soldados se están intentando capturar mutuamente pero, paradójicamente, el primero que lo intenta

[6]Grupos a los que es imposible quitarle las granjas, esencialmente porque al intentarlo, los propios soldados del jugador que intenta capturar no tienen comida, y serían capturados automáticamente. Para más información al respecto consultar BIBLIOGRAFÍA AQUÍ

será el capturado (por lo que ninguno de los jugadores intentará capturar al otro) o hasta combinaciones en la cual hay un bucle infinito que ninguno de los dos jugadores puede parar de repetir, ya que si alguno lo hace perderá la batalla y su grupo morirá[7].

1.1.3. El Go en occidente

El juego está en proceso de expansión en occidente, y cada día hay nuevas herramientas que podemos usar, como libros traducidos a inglés de chino, japonés o coreano, aplicaciones, canales[8] de televisión, material de juego, etcétera.

Pero por supuesto todavía quedan muchísimas herramientas que pueden incorporarse al juego para facilitar su aprendizaje, entre otras cosas. La aplicación que trato de desarrollar no es ni más ni menos que una herramienta que necesité en un torneo al que asistí a Madrid, la cual nadie pudo ayudarme a encontrar.

[7]Esta situación haría que la partida acabe el empate, a no ser que algun jugador decida no seguir el bucle porque cree que aún perdiendo su grupo puede ganar la partida.

[8]En Corea tienen tres canales de televisión distintos dedicados sólo al go. Como decía, allí el go es tan popular como aquí deportes como el fútbol o el baloncesto.

2. Descripción global del sistema

Como se dijo anteriormente, la idea surgió al asistir a un torneo en Madrid a finales de 2015. Cuando se juega una partida de torneo, el tiempo que hay para hacer las jugadas es limitado. Por eso, hay que usar todo el tiempo posible (incluso el tiempo del contrario) para pensar, hacer planes, cálculos, etcétera.

Las partidas de Go son muy largas (alrededor de 250 movimientos), por lo que recordar con exactitud todos ellos está al alcance de muy pocos. Sin embargo, es esencial revisarla al final para poder aprender de los errores cometidos. Por eso, la mayoría de los jugadores noveles mientras juegan la partida van apuntando en un papel cada jugada.

Esto para algunos es literalmente un suicidio, ya que es difícil concentrarse y estar desviando la atención hacia un papel en cada jugada. De ahí que una aplicación que automatizase todo este proceso (hay muchas aplicaciones que permiten ir anotando la partida en el movil en vez de en un papel, pero no que lo haga automáticamente) sería muy útil para algunos jugadores[9].

Finalmente, esto acabó siendo el trabajo fin de grado:

Realizar una aplicación que permita grabar una partida de Go de forma automática, sin que el usuario tenga que desviar su atención de la partida.

Se procede a describir el material que se va a utilizar, las metas de este trabajo fin de grado y la lógica del mismo.

2.1. Objetivos

El objetivo de este trabajo es crear una aplicación para sistemas operativos Android que sea capaz de 'grabar' una partida de Go y almacenarla en una base de datos, para su posterior reproducción.

Se asume que la cámara y el tablero estarán inmóviles durante el transcurso de la partida.

Para ello, se tendrán que cumplir una serie de objetivos:

- Aprender a hacer aplicaciones para sistemas operativos Android.
- Aprender a usar la librería OpenCV.
- Desarrollar la aplicación de forma que su uso sea lo más simple posible.

[9]Entre los que me incluyo. Nunca he apuntado una partida de Go mientras jugaba en un torneo, es totalmente ineficiente.

- Resolver, en la medida de lo posible, todos los problemas originados por factores externos incontrolables en muchas situaciones, como la iluminación o el ruido.

- Desarrollar una aplicación responsable, robusta y sencilla, que requiera el mínimo de interacción con el usuario[10].

2.2. Descripción de la aplicación

La aplicación constará de dos secciones principales. Una se encarga de grabar la partida y la otra de mostrarla.

Al entrar en la sección para grabar la partida, el usuario deberá hacer los preparativos para ésta[11]. Una vez la aplicación está lista para grabar, el usuario puede olvidarse de ésta. Al terminar la partida el usuario debe pulsar el botón de guardar, que le dará la opción de escribir los nombres de los jugadores antes de guardarla en la base de datos.

Para reproducir una partida, el usuario primero accederá a una lista con todas las partidas almacenadas en la base de datos. Elegirá la que desee y se le mostrará en una nueva ventana, con botones para avanzar y retroceder según desee en las jugadas.

2.3. Descripción de la parte física

Será necesario por supuesto un dispositivo Android y un trípode para elevar el dispositivo a una altura considerable. Esto cumple doble función, mejorar la calidad de las imágenes y estorbar lo menos posible a los usuarios.

Como dije, se asume que el dispositivo y el tablero estarán inmóviles durante el transcurso de la partida, pues en otro caso habría que tratar con infinidad de errores por diversas situaciones que puedan pasar y es mucho más interesante dedicar el tiempo a mejorar todo lo posible la detección de jugadas en el tablero. Además, no es una situación que ocurra a menudo.

En la figura 3 tenemos una imagen de todo el sistema al terminar de grabar una partida de Go.

[10]Para la parte de visión por computador.
[11]Concretamente, detectar las esquinas.

Figura 3: Sistema en funcionamiento grabando una partida.

2.4. Funcionamiento

En resumen, los pasos a seguir serían los siguientes:

- Montamos el trípode y colocamos la cámara apuntando al tablero, a una altura considerable[12]
- Seleccionamos grabar partida[13].
- Detectamos las esquinas.
- Comenzamos a grabar, jugamos la partida sin interactuar con el dispositivo.
- Cuando terminamos la partida, seleccionamos guardar en el dispositivo.
- La partida ahora aparecerá en la lista de partidas para reproducirla cuando deseemos.

[12]Medio metro, más o menos.

[13]Probablemente lo más cómodo sea tener el dispositivo en modo landscape para todo esto.

3. Interfaz de usuario

En este apartado se va a describir todo lo que puede experimentar el usuario respecto a la aplicación. Como navegar por ella, de que opciones se dispone y cual debe ser su uso.

Al comenzar se dispone de dos opciones: Reproducir o grabar una partida. En función del botón que pulsemos navegaremos a la sección correspondiente.

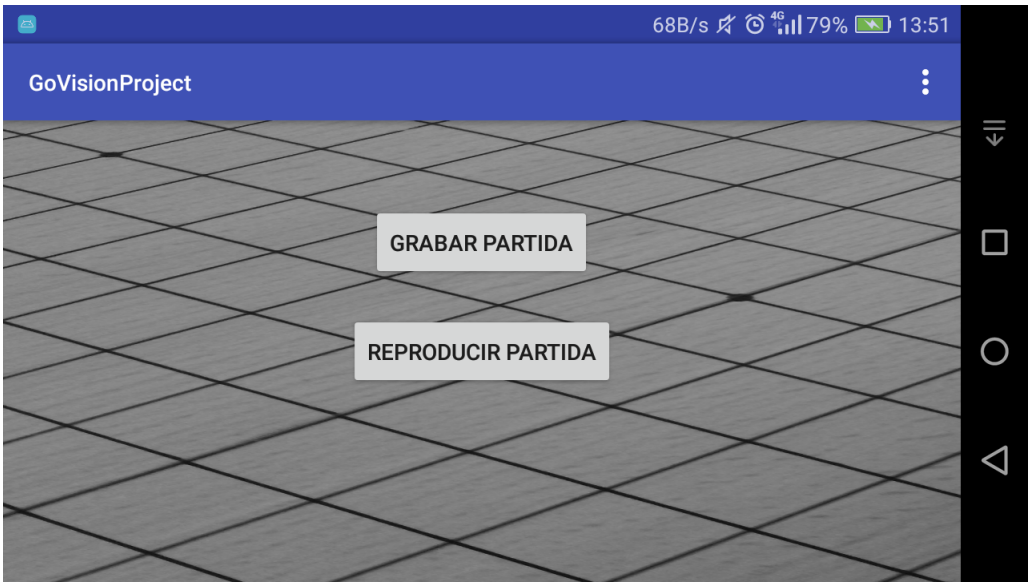

Figura 4: Pantalla de presentación con dos botones.

También tenemos un menú con las distintas opciones de las que dispone el usuario, que cambiarán según la sección en la que se encuentre.

A continuación describimos que podemos encontrar en cada una de ellas.

3.1. Reproducir partida

Si pulsamos el botón 'Reproducir partida' cargaremos la actividad que nos muestra todas las partidas de la base de datos en una lista.

3.1.1. Lista de partidas

Podemos borrar o editar partidas accediendo al menú individual de cada una. Cada elemento de esta lista está representado por los jugadores blanco y negro más la fecha en la que se jugó. Al editar una partida podemos cambiar los nombres de los jugadores. Una vez encontramos la partida que queremos visualizar, hacemos click sobre ella, cargando la pantalla de reproducción de partidas.

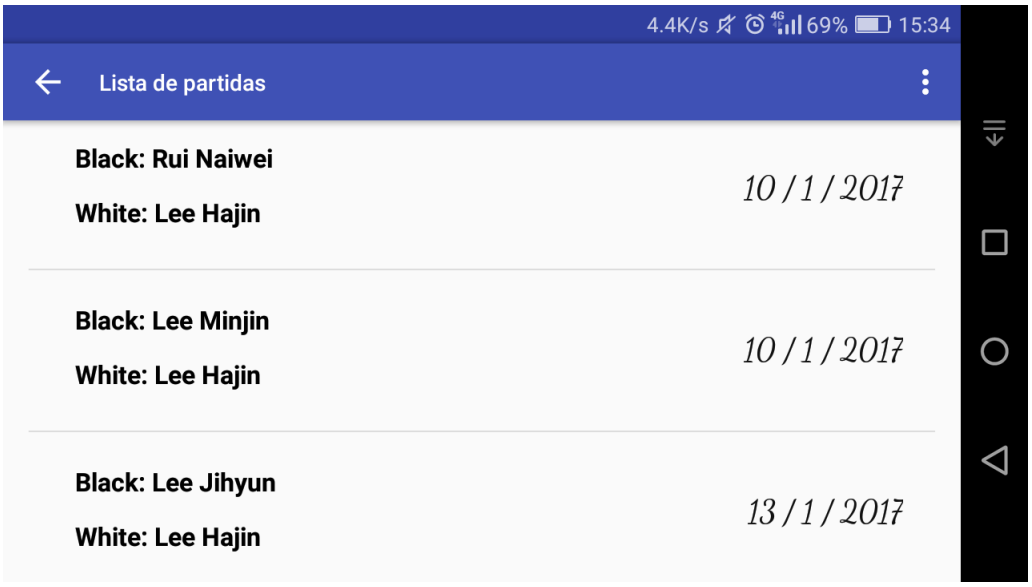

Figura 5: Lista de partidas.

3.1.2. Reproducción de la partida

Se dispone de cuatro botones, que permiten retroceder o avanzar los movimientos de uno en uno o de cinco en cinco. El tablero, que realmente es un grid de 19x19 imágenes, cambiará las imágenes correspondientes para reflejar el estado actual de la partida.

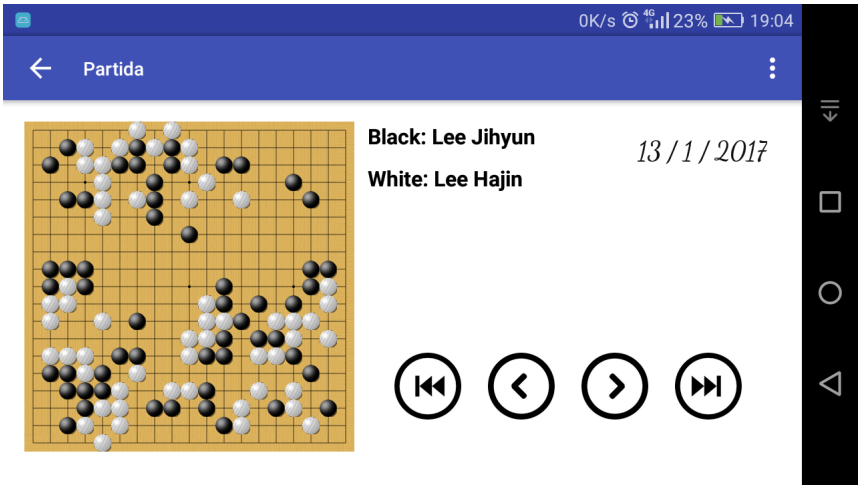

Figura 6: Partida en proceso de reproducción.

3.2. Grabar partida

Al pulsar el botón de grabar partida se carga la pantalla que se encarga de grabar la partida.

La pantalla refleja las capturas de la cámara, modificadas según las operaciones que estemos haciendo en cada momento.

También disponemos de un menú con multitud de botones, que hay que entender como usar para el correcto funcionamiento de la aplicación.

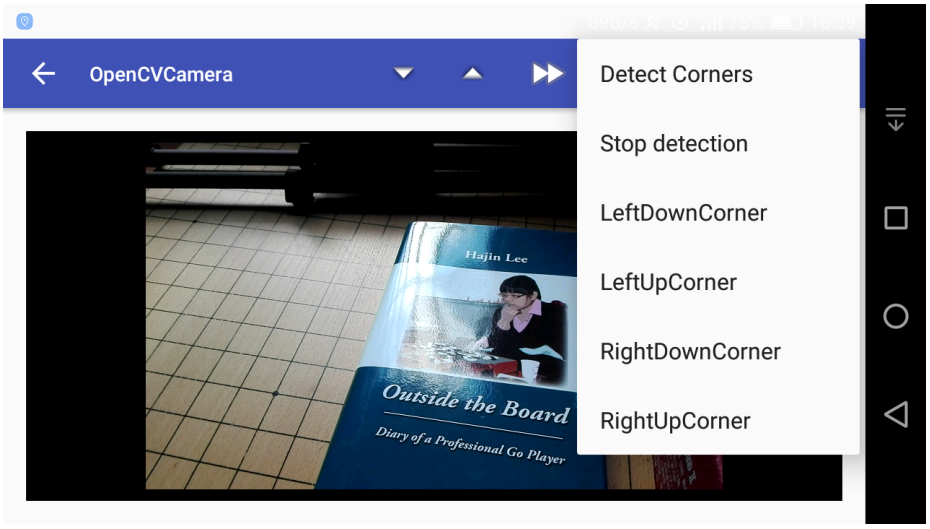

Figura 7: A la derecha el menú desplegable.

En la figura 7 vemos el menú desplegable, que nos permite identificar las esquinas del tablero.

Inicialmente la aplicación solo carga las imágenes que captura en la pantalla.

3.2.1. Detección de esquinas

Al pulsar la opción 'Detect Corners' se activa la parte de la actividad que busca círculos en la imagen. Si queremos parar de detectar las esquinas[14] solo tenemos que pulsar el botón 'stop detection', ahorrando así batería.

Para detectar las esquinas, se coloca una piedra[15] en la esquina correspondiente y se pulsa su botón. El algoritmo hace una media ponderada de los círculos detectados en la imagen, que además puede corregirse con las flechas del menú principal que podemos ver en la figura 8. Si por cualquier razón esto saliese mal[16], se puede repetir el cálculo volviendo a pulsar el botón correspondiente a la esquina.

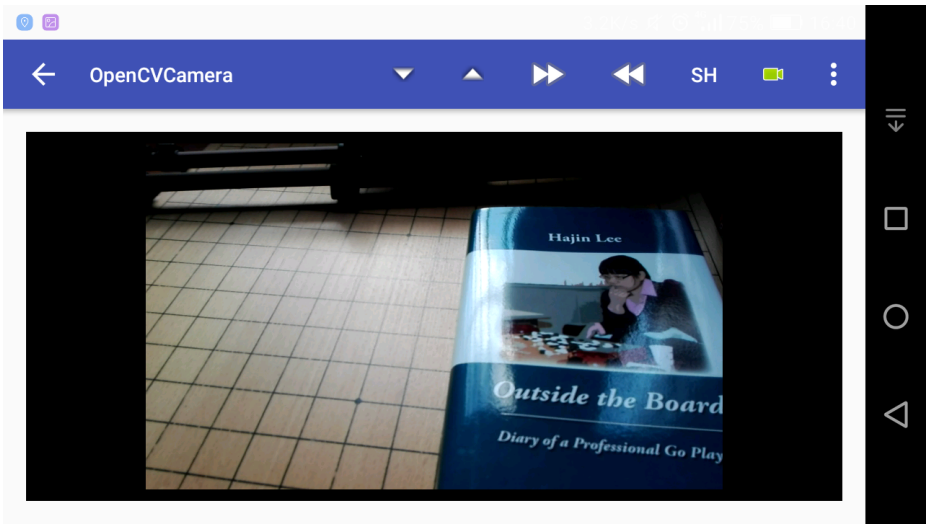

Figura 8: Vista del menú principal.

[14]Bien porque ya se han calculado todas y la partida todavía no va a empezar o por cualquier otra razón.

[15]Preferiblemente blanca, que son más fáciles de detectar.

[16]En mi experiencia casi nunca hay problemas.

3.2.2. Homografía

Una vez calculadas las esquinas, podemos comenzar a grabar pulsando el botón[17] de la cámara en color verde. Ahora se mostrará en pantalla la homografía de la imagen, de forma que el cuadrilátero que es el tablero se muestre como un cuadrado perfecto. Si queremos ver si las intersecciones se han calculado bien, podemos pulsar el botón 'SH'[18] que pintará un punto en cada intersección.

3.2.3. Detección de movimientos

A partir de este momento la aplicación está grabando. Cuando se detecta un nuevo movimiento, pintará un círculo alrededor de él, por si hay alguna duda para ver cual fue el último movimiento que se detectó.

Ahora también está visible el botón de grabar la partida en la base de datos[19]. Si se pulsa, se abre una ventana en la que el usuario puede escribir el nombre de los jugadores y con la opción de cancelar o guardar.

Si más adelante se guarda la partida de nuevo, se creará un nuevo registro en la tabla, por lo que el usuario deberá borrar manualmente el registro anterior (puede hacerlo en la lista de partidas).

También podemos observar que el botón de la cámara aparece de color naranja. Esto significa que está grabando. Si queremos pausar la grabación (para ahorrar batería o por cualquier otro motivo) solo tenemos que pulsar el botón. Veremos que se pone de color verde, por lo que para reanudar el análisis de las imágenes debemos pulsarlo de nuevo.

[17] En caso de que las esquinas no estuviesen calculadas la aplicación nos avisará de ello al pulsar el botón.

[18] Show Homography.

[19] Y han desaparecido las flechas, que ya no sirven para nada.

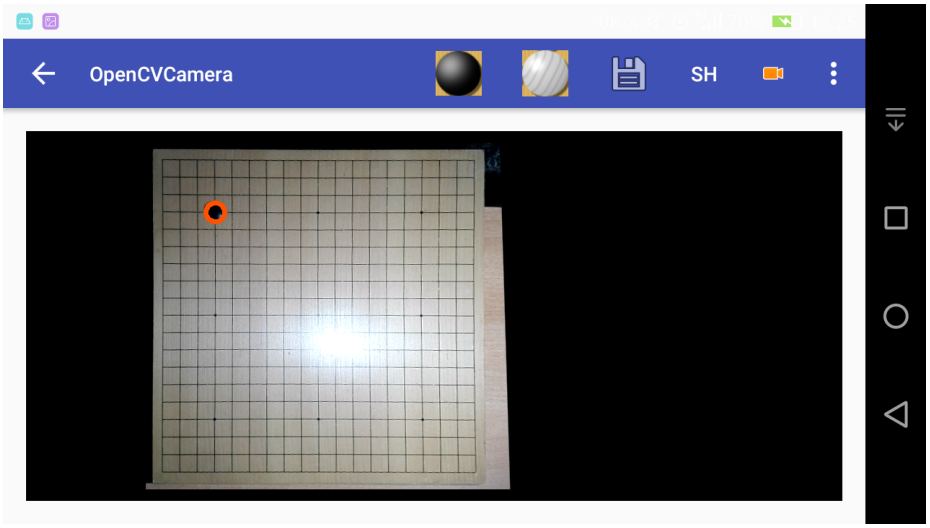

Figura 9: Vista del menú principal cuando se está grabando la partida.

Por último, disponemos de dos botones para que la aplicación nos comunique el número de piedras capturadas por el jugador negro y blanco.

4. Codificación

A continuación vamos a revisar parte del código de la aplicación, para entender que está haciendo realmente en cada momento. Así veremos cómo funciona y porque se ha hecho así.

Para simplificar la lectura del documento no se muestra código alguno. Si el lector desea inspeccionar alguna parte del código en concreto en el apéndice C puede ver como hacerlo.

4.1. Base de datos

Comenzamos describiendo la base de datos, ya que es importante saber como se guarda una partida para saber porqué estamos haciendo diversas operaciones más adelante.

La base de datos de la aplicación es muy sencilla. Consta de una sola tabla, que guarda los datos relativos a una partida:

- ID de la partida en la base de datos.
- Nombre del jugador negro.
- Nombre del jugador blanco.
- Fecha del día de la partida.
- Una cadena de caracteres que representa los movimientos que se hicieron durante la partida.

En la base de datos se guarda lo necesario para identificar y reproducir más tarde la partida. La cadena de caracteres que la representa está formada por el patrón 'C[fc];', uno por cada jugada. Su significado es el siguiente:

- C representa el color mediante la letra W o B, que indica quien ha jugado (White/Black).
- f representa la fila donde se ha jugado, puede ser una letra desde 'a' hasta 's'.
- c representa la columna donde se ha jugado, puede ser una letra desde 'a' hasta 's'.
- Los caracteres '[', ']' y ';' son delimitadores.

Por ejemplo, una partida que conste de cinco movimientos sería algo parecido a:

B[dd];W[pd];B[dp];W[pp];B[cn];

4.2. Clases de apoyo

Vamos a describir dos clases de las que es necesario entender su funcionamiento, ya que se han creado para lidiar con diversas situaciones que veremos más adelante.

4.2.1. Game

En primer lugar, vamos a hablar de la clase Game. Esta clase representa una partida completa, por lo que, como es de esperar, al leer una partida de la base de datos, se creará un nuevo objeto Game con los datos suficientes para reproducirla más tarde.

Además, cuenta con una serie de métodos de ayuda para hacer todos los cálculos necesarios durante la partida. Vamos a ver los más interesantes.

Un objeto game cuenta con una matriz tridimensional, que se crea al crearse el objeto.

Cada matriz bidimensional[20] contenida en esta matriz representa un estado de la partida. Cada vez que se añade una jugada, cambia el estado. Los valores de estas matrices bidimensionales pueden ser:

- 0: Representa una intersección vacía.
- 1: Representa una piedra negra.
- 2: Representa una piedra blanca.

Inicialmente esta matriz solo contiene ceros (el tablero está vacío).

[20] De 19 filas y 19 columnas, representando todas las intersecciones del tablero.

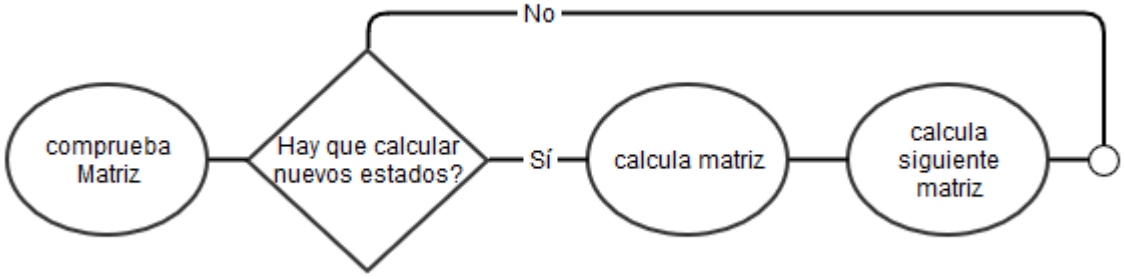

Figura 10: Diagrama representante de los calculos que se hacen dentro de la clase Game.

Cuando el usuario quiere avanzar en las jugadas (pulsando el botón correspondiente) se llamará al método 'compruebaMatriz' que podemos ver en el diagrama de la figura 10. Este método sirve para comprobar si los estados de la partida ya han sido calculados anteriormente[21] y no es necesario repetir el cálculo.

Una variable guarda cuantos estados se han calculado ya. Si hay que calcular uno o más estados, se llama al método 'calculaMatriz' que recibe como parámetro el número de estados a calcular.

El método calculaMatriz no es interesante, simplemente lee las jugadas del String, hace algunos cambios necesarios para que todo el proceso sea más claro y sencillo y llama al método 'calculaSiguienteMatriz', el cual se encarga de calcular el siguiente estado dado el actual y la siguiente jugada.

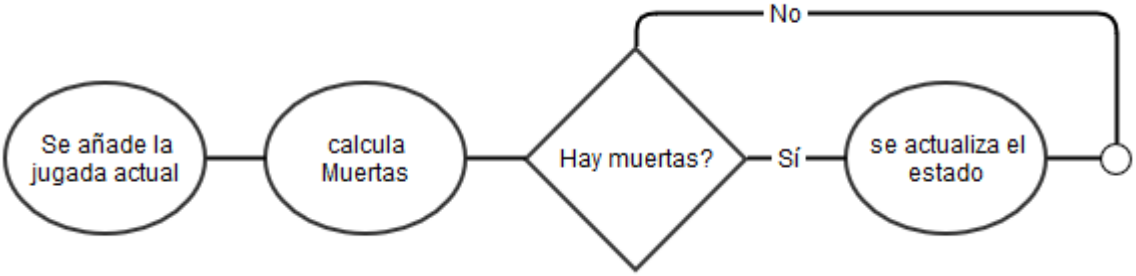

Figura 11: Método calculaSiguienteMatriz.

[21]Por ejemplo, si el usuario avanza 5 jugadas, retrocede 2, y avanza otras 5 (de golpe, recordemos que se puede avanzar o retroceder de uno en uno o de cinco en cinco) realmente solo hay que calcular 3 estados, pues los dos primeros ya fueron calculados al principio.

En la figura 11 se puede observar el método que se encarga de calcular el siguiente estado. Primero se obtiene una copia del estado actual. Sobre este estado se añade la jugada, se calcula si esa jugada captura otras piedras y, si se da el caso, se elimina del estado las piedras capturadas. Por último, actualizamos la matriz general con el nuevo estado.

El método más complicado es 'calculaMuertas'. Éste recibe como parámetros el estado actual de la partida, la posición donde se ha hecho la jugada y el turno. El método completo lo podemos encontrar en el apéndice A.

Recordemos que cuando se hace una jugada, ésta solo puede capturar las piedras adyacentes a ella (y las que estén conectadas a ella). Por eso, el siguiente algoritmo se repite para cada una de las cuatro posibles capturas, que son arriba, abajo, izquierda y derecha.

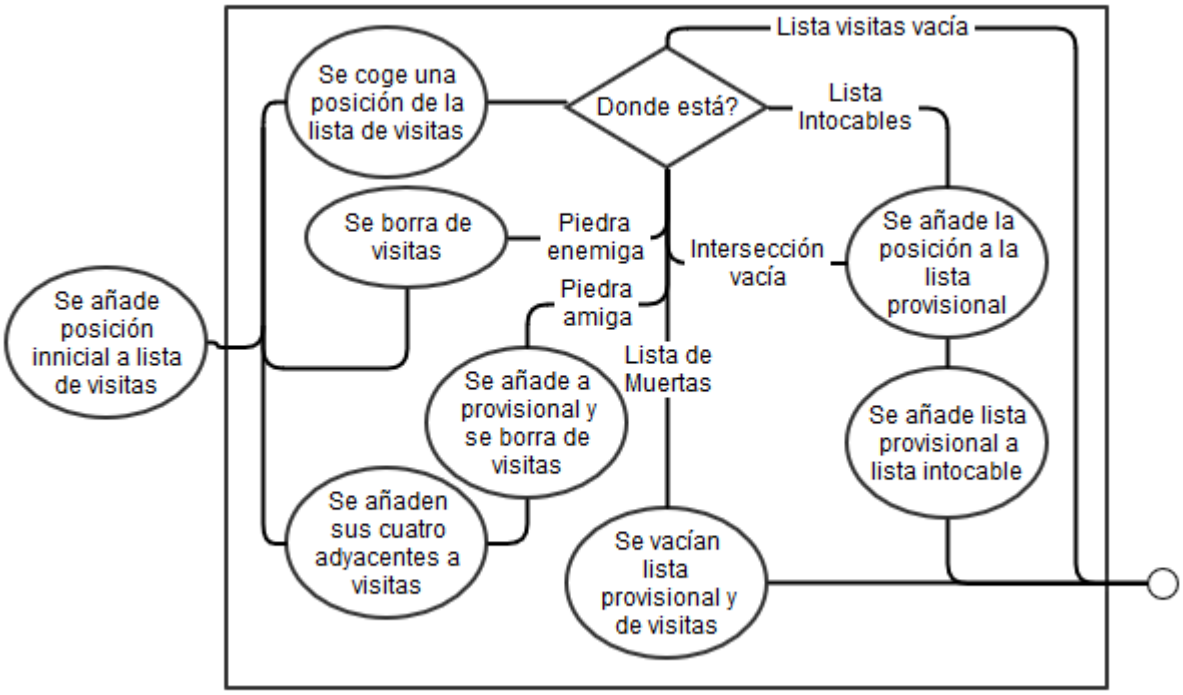

Figura 12: Algoritmo del método calculaMuertas.

Se dispone de una serie de Listas de objetos Position[22]:

- **Lista de muertas**. Esta lista almacena las intersecciones donde hay piedras muertas.
- **Lista provisional**. En esta lista se van guardando las posibles piedras muertas, no confirmadas hasta ahora.
- **Lista de visitas**. Esta lista contiene las intersecciones que tenemos que analizar para que termine el algoritmo. Cuando esta lista esté vacía, el bucle termina.
- **Lista de intocables**. Esta lista contiene intersecciones en las que hay piedras que sabemos no han sido capturadas.

En primer lugar se añade la posición[23] que se va a analizar a la lista de Visitas.

Ahora se repite el siguiente algoritmo hasta que la lista de visitas se quede vacía:

- Se comprueba si la posición que se está inspeccionando es una intersección vacía o se encuentra en la lista de intocables.

  La lista de intocables contiene intersecciones que ya se conoce no pueden ser capturadas. Por eso, si se da uno de estos dos casos, se añade la posición a la lista provisional y se vuelca ésta en la lista intocable, pues todas las jugadas que se hayan inspeccionado hasta ahora están vivas.
- Si en esta posición hay una piedra enemiga, se borra de la lista de visitas la posición y continuamos con el bucle.
- Si esta posición ya se encuentra en la lista de muertas no hay que hacer nada (pues ya se han detectado) por lo que se vacía la lista provisional y de visitas para que termine el bucle.
- Sin embargo, si hay una piedra amiga en esta posición, se añade ésta a la lista provisional, se borra de la lista de visitas y se añade las cuatro posiciones adyacentes a la lista de visitas (siempre y cuando no estén ya en la lista provisional, lo que indicaría que ya se añadieron anteriormente a la lista de visitas).

[22]Un objeto Position no es más que una clase con dos variables int, que indican la fila y columna donde se ha hecho una jugada.

[23]Por ejemplo, si se ha hecho una jugada en la intersección (3,3), para ver si se ha capturado la piedra de arriba se añade a la lista la intersección (2,3).

Una vez la lista de visitas se ha quedado vacía, si la lista provisional **no está vacía** significa que hay piedras capturadas. Estas piedras se añaden a la lista de piedras muertas, se resetea la lista de visitas y la lista provisional y se repite todo para otra de las cuatro posiciones adyacentes a la jugada inicial.

Este algoritmo puede parecer muy complicado de entender, pero la dificultad radica en entender bien las reglas de Go. Una vez se entiende bien como funciona la regla de la captura en Go, este algoritmo que a primera vista puede parecer muy complicado acaba pareciendo sencillo, pues solo se están aplicando las reglas del juego.

4.2.1.1. FormerGame

FormerGame es una extensión de Game, que se ha usado para, al detectar una jugada en una partida, calcular si ha capturado otras piedras (y no tener que detectarlo mediante visión por computador, que sería más complicado).

Para ello solo se ha modificado los métodos correspondientes para que devuelvan el Set de piedras muertas que se vio en el apartado anterior, en lugar de no devolver nada.

4.2.2. CompetitorPoint

Esta clase sirve para calcular en que intersección se ha hecho una jugada cuando se esta grabando una partida. Consta de una serie de campos:

- ID. Se usará no solo para identificar los competitorPoints, si no también para relacionarlos con sus respectivas intersecciones[24].
- Cuatro esquinas (minX, maxX, minY, maxY). Delimitan el cuadrado de influencia del competitorPoint, todos los círculos detectados cuyo dentro esté dentro de este cuadrado se asignarán a el.
- Stack. El número de votos conseguido hasta el momento.
- Found. Flag para saber si ya se detectó antes una jugada aquí o no.

[24]Si cogemos el ID y lo dividimos por 19, el cociente será la fila y el resto la columna que representa a la intersección.

La idea es tener una lista de CompetitorPoints que compiten entre ellos, cada uno representando una intersección, para ser elegidos como la siguiente jugada, ganando votos cuando se detecta círculos cerca de ellos. Cuando se cumplen ciertas condiciones que se verán más adelante, un CompetitorPoint cambia su estado de false a true (lo que indica que en su posición se ha hecho una jugada). Por supuesto, cuando se capturan piedras, los CompetitorPoints de las intersecciones con piedras capturadas cambian su estado de true a false.

4.3. Reproducción de partida

Sabiendo como funciona la clase Game descrita en el apartado 4.2.1, se verá que la reproducción de la partida es bastante sencilla.

Se dispone de un grid de 19 filas y columnas, compuesto por las imágenes de las intersecciones del tablero. Esta actividad dispone como se vio anteriormente de cuatro botones que hacen avanzar o retroceder la partida un número determinado de jugadas. Cuando se pulsa, se llama a un método interno que hace las siguientes cosas:

- Si se está pidiendo avanzar en la partida, llama al método 'compruebaMatriz' con el número de pasos que se quieren dar. Este método como vimos anteriormente se encargará de calcular los estados necesarios para llegar al estado que quiere el usuario.
- Si por el contrario se quiere retroceder, se hacen los ajustes necesarios en las variables para indicar que ahora estamos en un estado anterior.
- Una vez tenemos calculado el estado en el que estamos, la actividad se encarga de cambiar las imágenes necesarias en el grid para reflejar este cambio.

En la figura 13 se puede observar el grid antes y después de pedir a la actividad que se avance un paso.

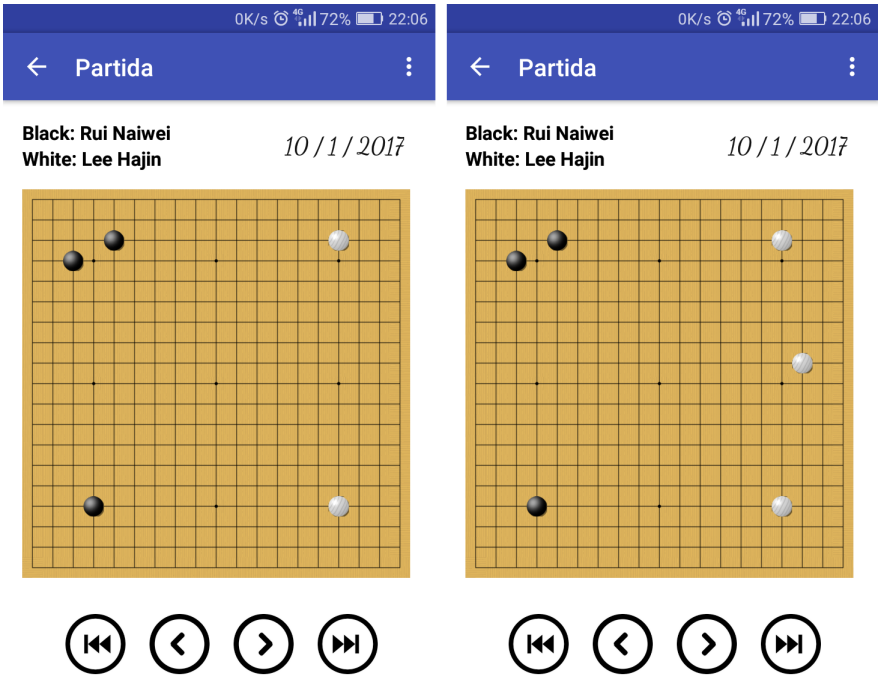

Figura 13: Grid antes y después de una jugada.

4.4. Visión por computador

Vamos a hablar ahora de la parte relativa a visión por computador. Para toda esta parte usamos una librería bastante 'famosa', OpenCV[25]. Ésta se encarga de proporcionarnos clases de apoyo para todas las operaciones relativas a la parte de visión. Con estas herramientas intentaremos construir poco a poco un algoritmo que alcance nuestros objetivos.

4.4.1. Instalación de OpenCV

Una vez está instalada OpenCV en la aplicación, se define una actividad que será la que hará uso de ella. Esta actividad en esta aplicación se llama OpenCVCamera e implementa la clase:

`CameraBridgeViewBase.CvCameraViewListener2`

Esta actividad tiene asociada un objeto JavaCameraView, que es el que se encarga de controlar la cámara.

Habrá que implementar tres métodos, que son los siguientes:

[25] `http://opencv.org/`

- **onCameraViewStarted**. Se encarga de las operaciones iniciales al iniciar la grabación.
- **onCameraViewStopped**. Se encarga de las operaciones finales al finalizar la grabación.
- **onCameraFrame**. Se encarga de realizar operaciones sobre cada imagen.

En los dos primeros solo se hacen operaciones rutinarias (inicialización de objetos y demás). El interesante es 'onCameraFrame', que se verá a continuación. Mediante varias variables booleanas se controla que parte de onCameraFrame se ejecuta en cada momento de las siguientes. El método completo se puede ver en el apéndice B.

4.4.2. Detección de esquinas

Para comenzar a detectar las esquinas se debe activar en el menú la opción detect corners. Esto activará el proceso de imágenes:

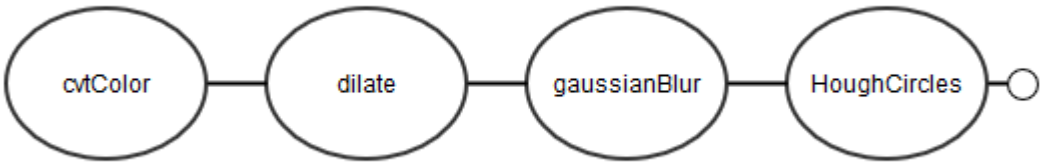

Figura 14: Diagrama de procesamiento de imágenes.

Primero se pasa la imagen a tonos de gris, nunca se hace ninguna operación sobre la imagen en color. A continuación se dilata[26] la imagen, para que sea más fácil detectar los círculos. Se aplica un filtro gaussiano para suavizar la imagen y se calculan círculos de Hough sobre ella.

4.4.2.1. Hough

La transformada de Hough es una técnica usada para detectar formas geométricas que puedan ser representadas por una expresión matemática (en este caso, circunferencias). Consiste en detectar ciertos píxeles de la imagen que otorgarán votos a cada posible circunferencia que los contenga.

[26]Más adelante se explicará porque dilatamos y erosionamos la imagen.

La expresión matemática que define una circunferencia es:

$$(x - c_x)^2 + (y - c_y)^2 = r^2$$

En la figura 15 se observa el espacio donde se detectan las circunferencias y el espacio de parámetros, donde cada punto representa a una circunferencia en el espacio imagen.

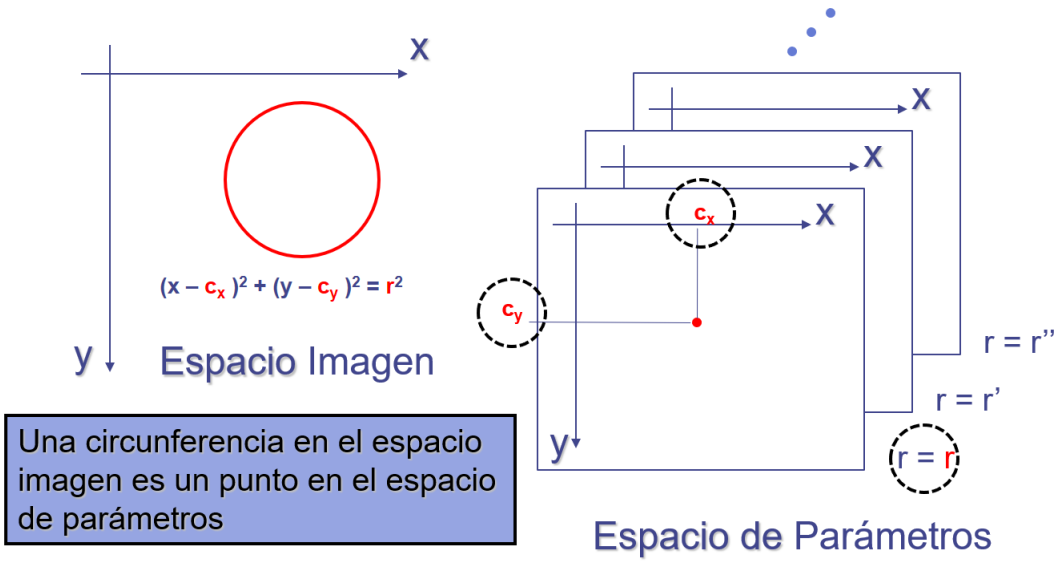

Figura 15: Circunferencias de Hough en el espacio de parámetros.

Valores diferentes de (c_x, c_y, r) proporcionan distintas circunferencias.

Para cada píxel de contorno que aparece en la posición (x_0, y_0) existe una familia de circunferencias que pasan por este punto dadas por:

$$c_x = x_0 + cos(\theta) \cdot r$$
$$c_y = y_0 + sin(\theta) \cdot r$$

Cada píxel de contorno vota por todas las posibles circunferencias que lo contienen. Si aparece un punto en el espacio de parámetros que tenga muchos votos es que posiblemente hay una circunferencia en el espacio imagen.

4.4.2.2. Detección de una esquina

Hasta ahora lo único que se hace es detectar círculos en la imagen. Para detectar una esquina, se debe **primero** colocar una piedra blanca en la esquina que se quiere detectar y a continuación pulsar el botón correspondiente a esa esquina en el menú.

A partir de este momento las circunferencias detectadas con Hough se usarán para calcular la esquina correspondiente. Para cada circunferencia se obtiene su centro y se hace la media ponderada con el centro calculado hasta ese momento. Las circunferencias cuyo centro están muy alejadas del calculado hasta ahora se descartan.

Pulsando los botones del menú de opciones se puede corregir manualmente las coordenadas de la esquina en cuestión.

Si el primer círculo detectado fuese un falso positivo[27] habría que repetir el proceso pulsando de nuevo el botón en el menú.

Este proceso hay que repetirlo para las cuatro esquinas del tablero. Una vez éstas están calculadas, podemos comenzar a grabar la partida, pulsando el botón de grabar en el menú. Adicionalmente, podemos ver las intersecciones si pulsamos el botón SH[28] aunque esto consume bastante memoria, así que es recomendable desactivarlo al comprobar que todo funciona correctamente.

4.4.3. Homografía

Al pulsar el botón 'grabar' en el menú ya se puede dejar sola la aplicación. Ésta calculará primero la homografía respecto a los bordes del tablero y los competitorPoints.

La homografía es una técnica para hacer transformaciones proyectivas entre dos planos. En la figura 16 se pueden ver los distintos tipos de transformaciones proyectivas.

[27] En mi experiencia no ocurre casi nunca, pero puede llegar a ocurrir por ruido respectivo a la iluminación y otras fuentes.

[28] Este botón muestra todos los CompetitorPoints en la imagen, así podemos ver si coinciden con las intersecciones del tablero.

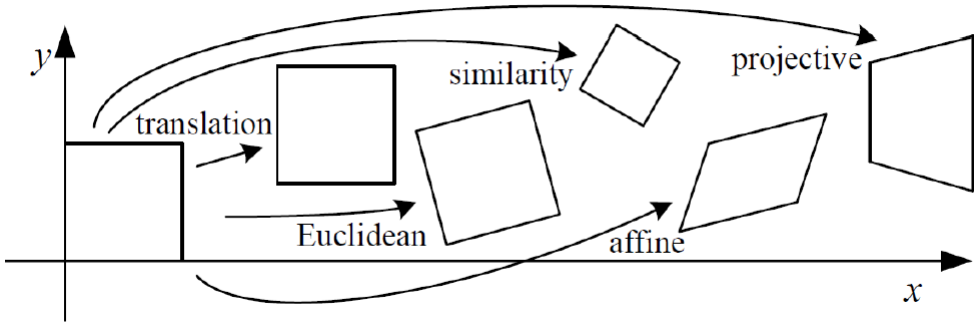

Figura 16: Distintas transformaciones proyectivas.

En este caso hay que lidiar con una transformación de tipo proyectivo (la 'peor' de todas). Con la matriz de homografía se podrá transformar el cuadrilátero que representa al tablero en la imagen original a un cuadrado en la imagen rectificada. Para hacer esto hacen falta las coordenadas de las cuatro esquinas[29] en la imagen y los cuatro puntos equivalentes en la imagen rectificada.

Para n puntos tenemos el sistema de ecuaciones siguiente:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 & -x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 & -y'_1 \\ & & & & & \vdots & & & \\ x_n & y_n & 1 & 0 & 0 & 0 & -x'_nx_n & -x'_ny_n & -x'_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -y'_nx_n & -y'_ny_n & -y'_n \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

Dado un punto en la imagen inicial podemos calcular sus coordenadas en la imagen corregida con la matriz de homografía de la siguiente forma:

$$\lambda \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

[29]Realmente solo hacen falta cuatro puntos, no hace falta que sean las esquinas, pero éstas nos darán la máxima precisión posible.

4.4.4. Cálculo de homografía y competitorPoints

Como se ha visto, para calcular la homografía hacen falta dos sets de puntos. El primero son los puntos calculados anteriormente (las cuatro esquinas). Los segundos serán los equivalentes en la nueva imagen. Como interesa crear un cuadrado perfecto, se les asigna los valores que más convienen[30]. La matriz de homografía se calcula usando la librería openCV.

También se calculan las coordenadas de los puntos de la imagen para poder dibujarlos si el usuario pulsa el botón 'show homography'.

Por último, se crea y añade a la lista de competitorPoints todas las intersecciones, cuya área (que viene a ser un cuadrado) está delimitada por las variables minX, maxX, minY, maxY. La idea detrás de esto es que más adelante cuando se detecte un círculo, si su centro se encuentra dentro de este área este competitorPoint ganará puntos de cara a sus compañeros.

Después de todos estos cálculos, las variables booleanas correspondientes se activan para comenzar la detección de jugadas.

4.4.5. Detección de jugadas

En la figura 17 puede verse el proceso al que es sometida la imagen desde su captura inicial hasta que se le aplica Hough.

En primer lugar se aplica la homografía calculada anteriormente sobre la imagen, para transformar el tablero a un cuadrado perfecto. A continuación se transforma la imagen a tonos de gris y se suaviza para eliminar ruido.

En función de si se están buscando piedras negras o blancas se aplica erosión o dilatación y transformadas de Hough para reconocer circunferencias en la imagen.

La transformada de Hough que se aplica cuando se están buscando piedras negras es más débil que la de las blancas, ya que es más difícil en general detectar estas circunferencias. A cambio se aplica más adelante restricciones extra para lidiar con los falsos positivos, que podrían detectarse por culpa de esto.

[30]El valor de height de las imágenes es 480 y el de width es 800, por lo que asignamos un cuadrado de aproximadamente 450 píxeles de lado.

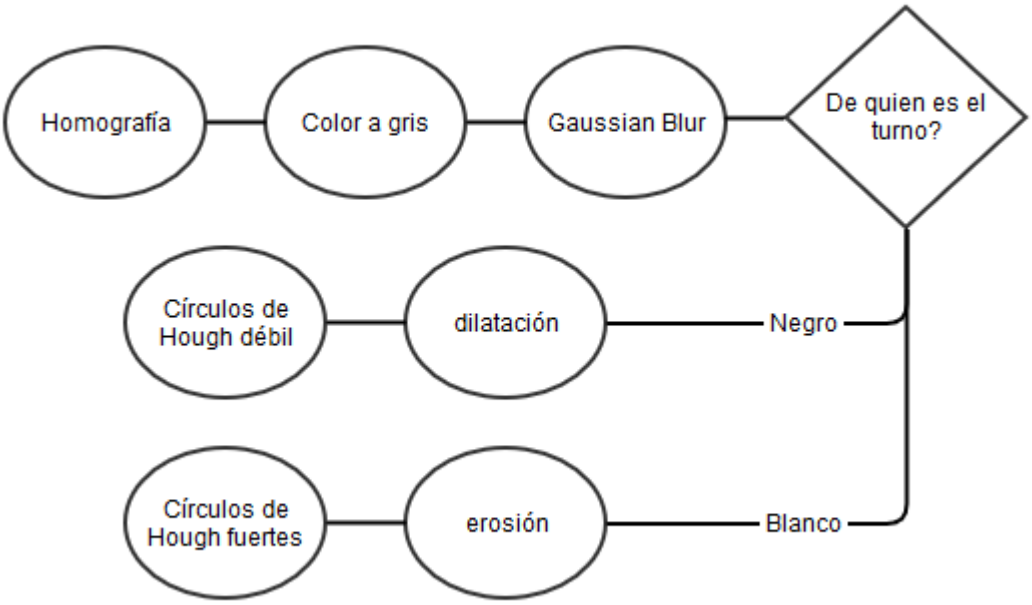

Figura 17: Procesado de la imagen durante la detección de jugadas.

4.4.5.1. Erosión y dilatación

La imagen de la izquierda en la figura 18 es la imagen sin procesar. Al aplicarle un filtro de dilatación, obtenemos la imagen central, y al aplicarle el filtro de erosión obtenemos la imagen de la derecha.

Figura 18: Efectos de dilatación y erosión.

Lo interesante es que si se quiere detectar piedras negras, conviene aplicar un filtro de dilatación. Al hacerlo, haremos más grandes las pequeñas zonas entre piedras, marcando más el círculo (al mismo tiempo que se hace más pequeño, pero esto no importa). Lo mismo ocurre con las piedras blancas.

El efecto de esto fue espectacular, inicialmente el algoritmo tenía problemas para detectar las piedras adyacentes entre si. Después de añadir esto, el problema prácticamente desapareció, sin contar algunas excepciones con las que hay que lidiar de otra forma.

También, si es la primera imagen que toma el algoritmo, se guarda una lista con subimágenes de todas las intersecciones, para hacer correlación cruzada más adelante.

Una vez se tienen los círculos, para cada uno se aplica un algoritmo para dar votos a los competitorPoints.

4.4.6. Obtención de votos

Para cada círculo, comprobamos que efectivamente corresponde a una intersección. Si esta intersección está libre entonces gana un voto. Además, en caso de que sea la intersección con más votos hasta ahora, se convierte en la ganadora actual de la ronda.

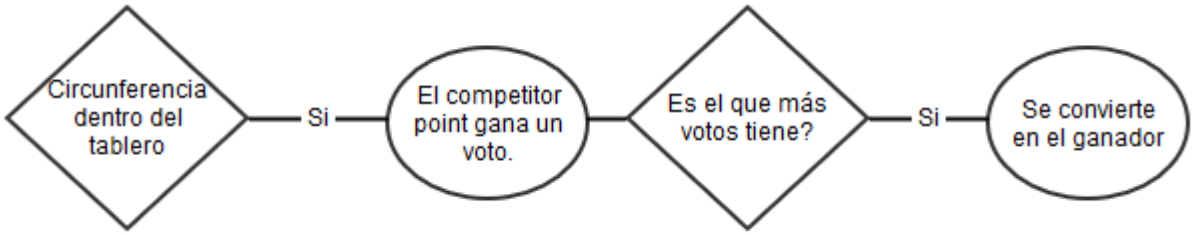

Figura 19: Algoritmo de obtención de votos.

Obviamente, si la circunferencia está fuera del tablero, ningún competitorPoint se llevará su voto, por lo que el algoritmo no se ejecuta. De la misma forma, si el competitorPoint no es el que más votos tiene no se hace nada con el, simplemente almacena su voto.

4.4.7. Ganador

Una vez todas las circunferencias han sido clasificadas, se analiza el ganador en este momento. Si el ganador ha sobrepasado un número determinado

de votos, deberá pasar unos tests para detectar si es un falso positivo.

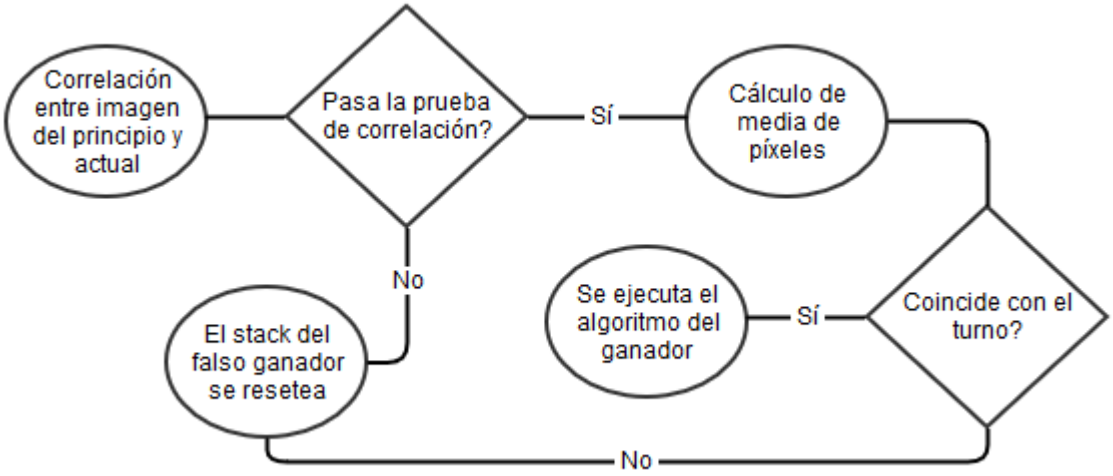

Figura 20: Secuencia que detecta falsos positivos.

Básicamente hay dos tests. El primero se encarga de comprobar si es un falso positivo, haciendo correlación entre la imagen inicial de la intersección y la actual. Si el valor de la correlación es muy alto significa que la imagen no ha cambiado apenas, por lo que es probable que se detectara un círculo debido a efectos de luces, sombras y otros ruidos[31].

Si pasa la prueba de correlación, entonces se comprueba que la piedra que se ha detectado corresponde al turno del jugador. De esta forma se asegura de guardar en el orden correcto las jugadas en la base de datos[32].

En caso de que falle alguno de los dos tests, el ganador pierde todos sus votos y se sigue analizando imágenes. En caso contrario, se ejecuta un algoritmo para registrar la nueva jugada y hacer los preparativos para seguir detectando nuevas jugadas.

[31]El algoritmo de correlación que se usa en OpenCV es capaz de amortiguar el efecto de luces y sombras entre dos imágenes.

[32]Si uno de los dos jugadores juega muy rápido, podría detectarse su jugada antes que la del jugador correspondiente y alterar el orden de jugadas.

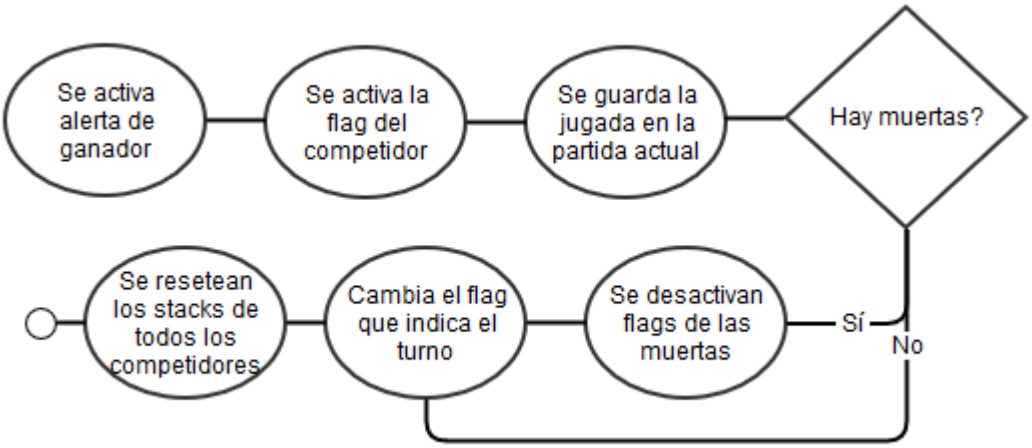

Figura 21: Pasos a seguir con el ganador.

Se ejecutan una serie de alertas que indican que se ha encontrado un nuevo movimiento. El flag del competidor sirve para saber si en esa posición ya se ha detectado una piedra anteriormente, de esa forma no competirá cuando se detecte su circunferencia correspondiente.

Se guarda la jugada con el formato que vimos en la sección 4.1 y se calcula si esa jugada ha capturado otras piedras. En ese caso, los competidores correspondientes a dichas piedras desactivan su flag para volver a competir (ya que ahora pueden jugarse de nuevo piedras en los huecos vacíos).

Por último, el flag que indica de quien es el turno cambia su valor y se resetean todos los stack[33]. Después de esto se continua analizando imágenes.

[33]Esto ayuda a eliminar falsos positivos que van acumulando votos con el tiempo.

5. Pruebas y resultados

5.1. Primera Prueba

A continuación se verán las primeras tomas de contacto y los resultados obtenidos.

5.1.1. Descripción actual del sistema

Se tiene la cámara elevada a una altura considerablemente alta, con el soporte algo inclinado para favorecer la captura de imágenes de forma que la deformación del tablero sea la mínima posible. Esto será muy importante para que las piedras no estén demasiado distorsionadas al hacer la homografía y no aparezcan como elipses.

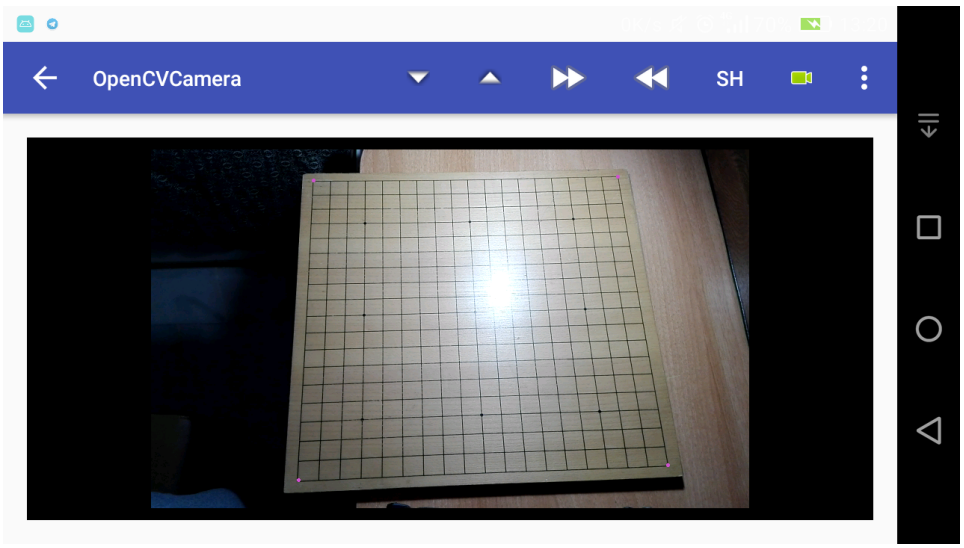

Figura 22: Detección de las cuatro esquinas del tablero (puntos rosas).

Se calculan las cuatro esquinas, una a una, mediante el sistema 'place a stone', que consiste en colocar una piedra (preferiblemente blanca) en la esquina que se va a detectar. Esta piedra la detecta el sistema mediante detección de círculos con Hough. Se obtiene el centro del círculo y mediante un algoritmo se va corrigiendo ese centro con los datos entrantes de cada imagen.

También se puede corregir manualmente este punto con unos botones en el menú.

En la figura 22 se aprecia el resultado.

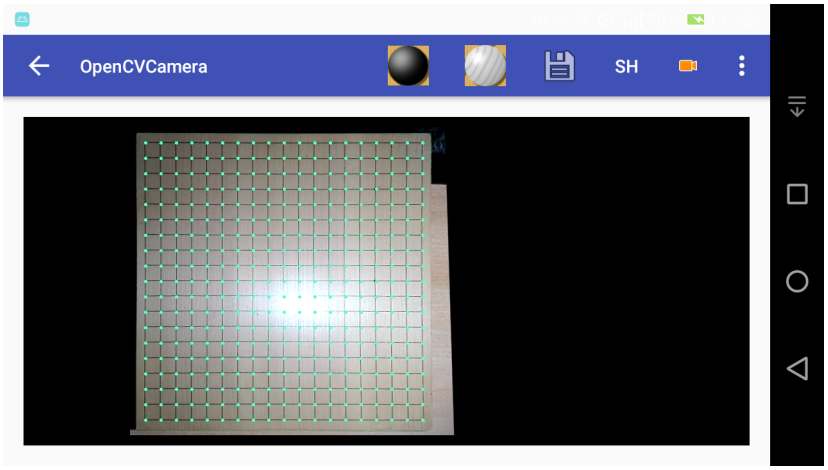

Figura 23: Detección de todas las intersecciones del tablero.

Una vez se obtienen las cuatro esquinas se puede empezar a grabar. Al pulsarse el botón, se calcula la homografía del tablero y comienza a detectar movimientos mediante detección de círculos con Hough. El resultado lo vemos en la figura 23. La imagen es procesada de la siguiente forma:

- Se transforma la imagen a escalas de gris.
- Se aplica un filtro gaussiano para eliminar pequeños efectos de luces en las piedras negras.
- Se calcula Hough con unos parámetros determinados.

En la figura 24 podemos ver el primer movimiento de una partida.

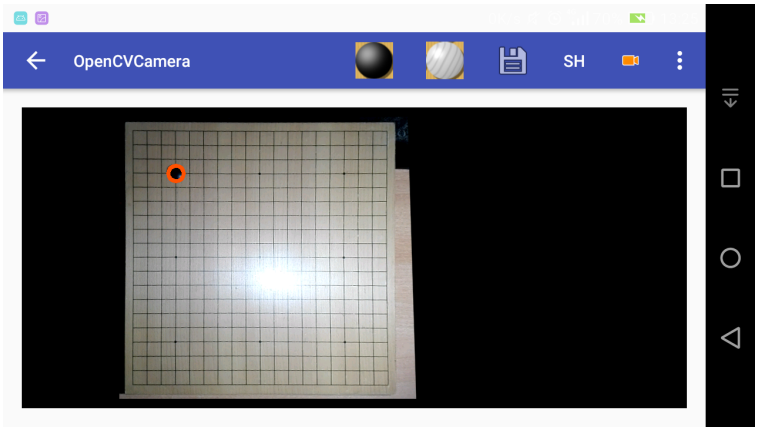

Figura 24: Primer movimiento de una partida (marcado en rojo).

Para detectar un movimiento se usa un algoritmo de votación muy sencillo[34]. Por cada círculo que se detecta en la imagen, se le da un voto a la intersección mas cercana al centro de ese círculo. Cuando una intersección almacena 10 votos, se considera que se ha jugado una piedra ahí y se almacena el movimiento, con sus respectivas consecuencias[35].

En las figuras 25 y 26 se puede observar la partida más avanzada, justo antes y después de capturar una piedra.

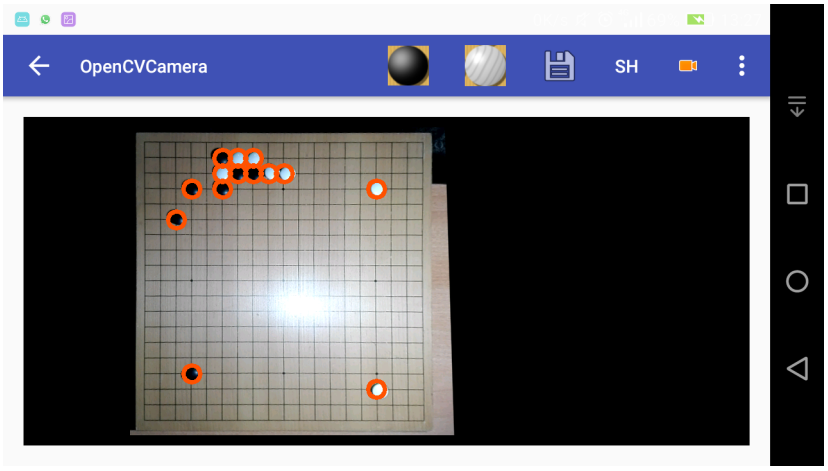

Figura 25: Antes de capturar.

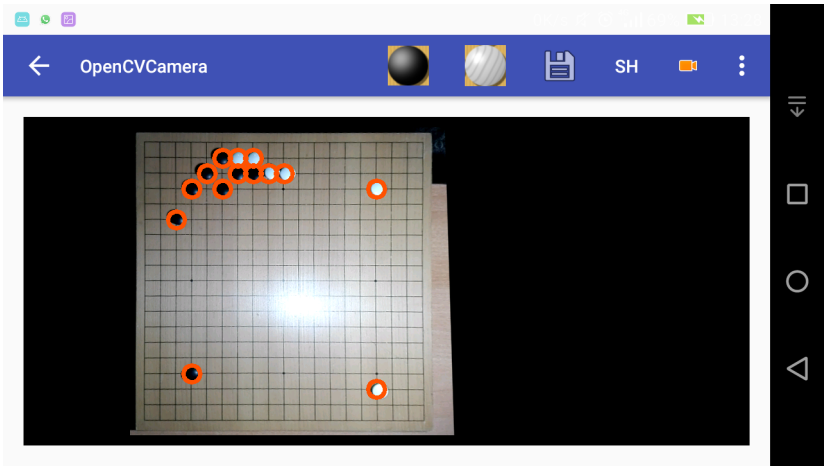

Figura 26: Captura de la piedra (nótese la desaparición del círuclo rojo).

[34]Por ahora.

[35]Calculo de piedras muertas, eliminación en el sistema de votación, etc.

5.1.2. Descripción de los resultados

Para el poco procesamiento de imagen[36], se obtienen unos resultados bastante buenos. Se detectan prácticamente todos los movimientos posibles. Parece ser que las piedras negras son más difíciles de detectar que las blancas, pues éstas se reconocen casi al instante, mientras que en el caso de algunas negras suele tardar alrededor de 1 segundo.

Hay varios problemas que hay que solucionar en las próximas pruebas:

- Al cabo de un tiempo, la aplicación deja de funcionar debido a algún problema que no es posible identificar en Android Studio. Parece ser un problema de memoria, OpenCV no calcula bien la memoria que está utilizando C en la aplicación, el cálculo de memoria falla y la aplicación se reinicia.

  Se va a investigar como darle mas memoria total a la aplicación y descubrir que es lo que está fallando exactamente.

- Algunas jugadas no se detectan bien. Puesto que se detectan círculos, cuando una jugada se juega en zonas en las que tiene piedras amigas alrededor, los bordes naturales se pierden parcialmente, hasta el punto de quedar poco borde que forme un círculo, impidiendo que se detecte por el algoritmo de Hough. Un ejemplo de esto lo podemos ver en la figura 27 .

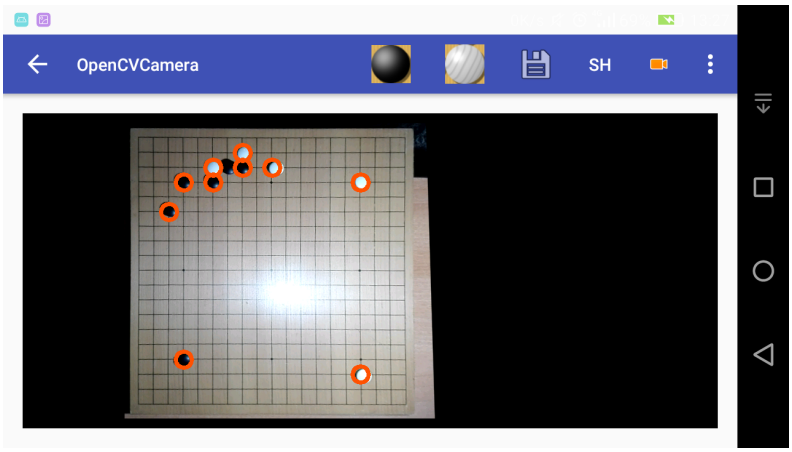

Figura 27: La piedra negra no es detectada hasta pasados 10 segundos.

[36]En este primer experimento, se dedicó muchísimo tiempo a los algoritmos, depuración de errores, creación de clases de apoyo etc. La idea era que todo funcionase sin errores, para dedicar los siguientes experimentos al afinamiento de la detección de las jugadas.

Posibles soluciones que se probarán en el próximo experimento serán dividir la imagen en sus respectivos canales y guardar con la "toma más blanca"para detectar piedras negras y la "toma más negra"para detectar piedras blancas. También se probará la dilatación para detectar piedras negras y erosión para piedras blancas.

- Debido a diversos factores[37] a veces hay falsos positivos. Para eso se mejorará el algoritmo de votación con tres posibles mejoras: Competición entre puntos mediante un campeón y subcampeón. Para que el campeón sea elegido como movimiento no solo debe tener un número mínimo de votos, si no también tener cierto porcentaje mas que el subcampeón. Eliminación de todos los votos almacenados tras la elección de un campeón. Añadir sistema para eliminar votos si no se detecta un círculo en el mismo punto al cabo de un tiempo.

  Podemos ver un ejemplo de falso positivo en la figura 28 .

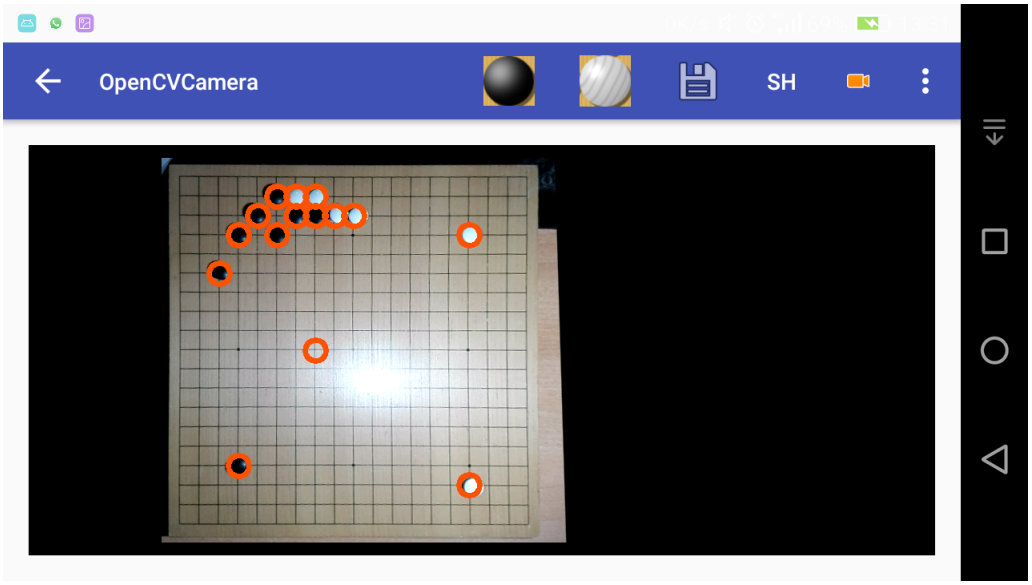

Figura 28: Falso positivo (Círculo en rojo marcando una piedra inexistente).

5.2. Segunda prueba

En esta prueba se han aplicado los cambios mencionados en la anterior, uno a uno, viendo que resultado daban. En general han sido buenos, aunque siguen habiendo cosas que arreglar.

[37]Luz, textura del tablero...

5.2.1. Cambios aplicados

Respecto a la detección de piedras, se ha probado a dilatar la imagen cuando es el turno de negro, y a erosionarla cuando es el turno de blanco. También se ha probado a dividir la imagen en tres canales y usar el más claro en el turno de negro y el más oscuro en el turno de blanco.

Se ha aumentado el tamaño de la memoria y cambiado el código para cuidarla de la mejor manera posible. También se han probado diversos métodos para controlar la limpieza de la memoria.

Por ahora no se ha hecho nada para arreglar los falsos positivos. Son muy poco comunes y en las pruebas que he hecho no ha aparecido ninguno. Además, este sistema perjudicaría la detección de las piedras en general, por lo que antes de incorporarse, hay que mejorar la detección de las piedras negras.

5.2.2. Descripción de los resultados

La división de la imagen en tres canales no sirve prácticamente de nada y perjudica demasiado a la memoria[38] como para ser una opción.

Sin embargo, la erosión y dilatación han dado unos resultados sorprendentemente buenos. Tanto es así que es posible detectar casi[39] cualquier jugada de blanco. Las negras es otro cantar, aunque ha mejorado considerablemente también. Las más difíciles de detectar pueden requerir bastante tiempo[40], así que el objetivo del siguiente experimento será principalmente perfeccionar esta parte.

En la figura 29 se ve una partida grabada hasta ese punto sin ningún fallo. Aunque avanzó un poco más sin errores, al cabo de un tiempo el ruido hizo que la cámara se desenfocase y tardó demasiado tiempo en enfocar de nuevo al tablero. En ese tiempo las sombras y el ruido combinados dieron lugar a varios falsos positivos.

Hay por tanto un nuevo problema, hay que investigar si se puede sellar el foco de la cámara o, en caso contrario, ver hasta que altura podemos tener

[38] Al dividir una imagen se crean tres imágenes nuevas. Pasamos de tener 2 imágenes por iteración a 5, algo que ahora mismo es intolerable.

[39] No me atrevo a decir todas, ya que no he probado todas las combinaciones posibles.

[40] Desde unos segundos a un minuto para las más complicadas. Puede incluso ser imposible detectarla si tiene muchas piedras negras alrededor, haciendo imposible ver un círculo (si bien esta situación en una partida real no ocurriría prácticamente nunca).

la cámara de forma que, cuando los jugadores[41] hagan sus movimientos, no desenfoquen el tablero.

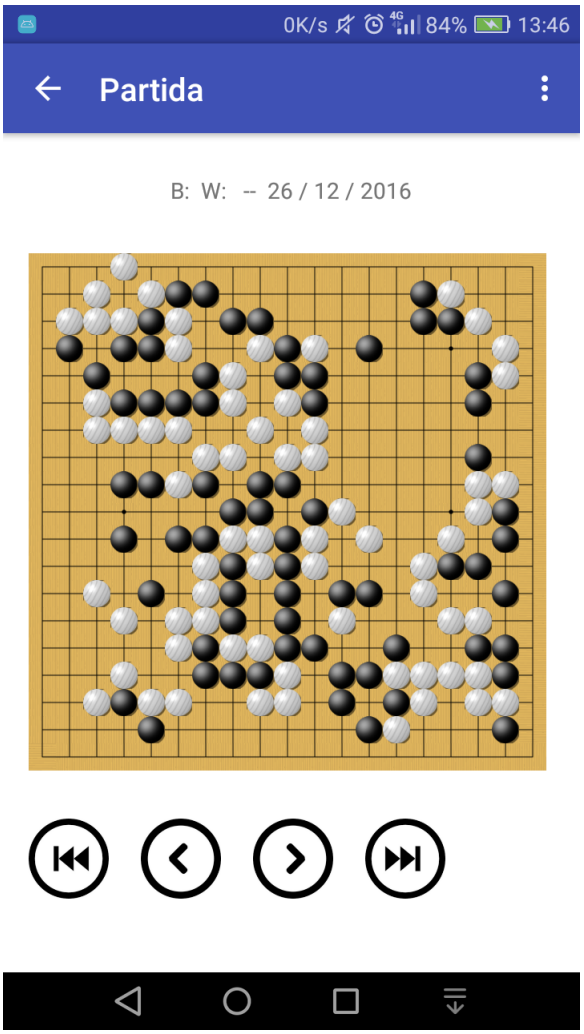

Figura 29: Partida de 142 movimientos detectada sin ningún fallo.

Respecto a los experimentos con la memoria, en general no han servido de nada, a excepción de la limpieza del código.

Llamar al recolector de basura no hace nada, pues esta línea de código lo que hace realmente es recomendar a Android que sería un buen momento para limpiar (y Android no hace ni caso). Tampoco hubo ningún cambio al

[41] Aunque sinceramente creo que la altura actual está bien. Simplemente yo estaba de pie, demasiado cerca de la cámara.

informar al sistema de que la aplicación necesita mucha memoria, ya que siguió dándole exactamente la misma de antes.

Lo que parece que ha funcionado es la limpieza del código. Se ha intentado no crear nuevos objetos en cada iteración, creándose fuera del bucle y reutilizando objetos cuando es posible. Antes, la aplicación solía dejar de funcionar al cabo de pocos minutos, si bien a veces no daba problemas. En las pruebas actuales se han grabado partidas durante media hora sin ningún problema, aunque desconozco si con el paso del tiempo fallará y cual es la probabilidad de que esto suceda. Conforme se hagan más pruebas habrá más datos para analizar el problema.

5.2.3. Experimento

Vamos a ver un experimento en concreto que nos aporta datos interesantes.

En la figura 30 podemos observar las condiciones y el estado de la partida al terminar de grabar.

Figura 30: Partida de más de 200 movimientos a punto de terminar.

Lamentablemente, en este punto falló la gestión de la memoria de la aplicación, haciéndola reiniciarse a falta de unos diez movimientos para el final de la partida.

Durante esta partida ocurrieron cosas muy desconcertantes, vamos a verlas a continuación.

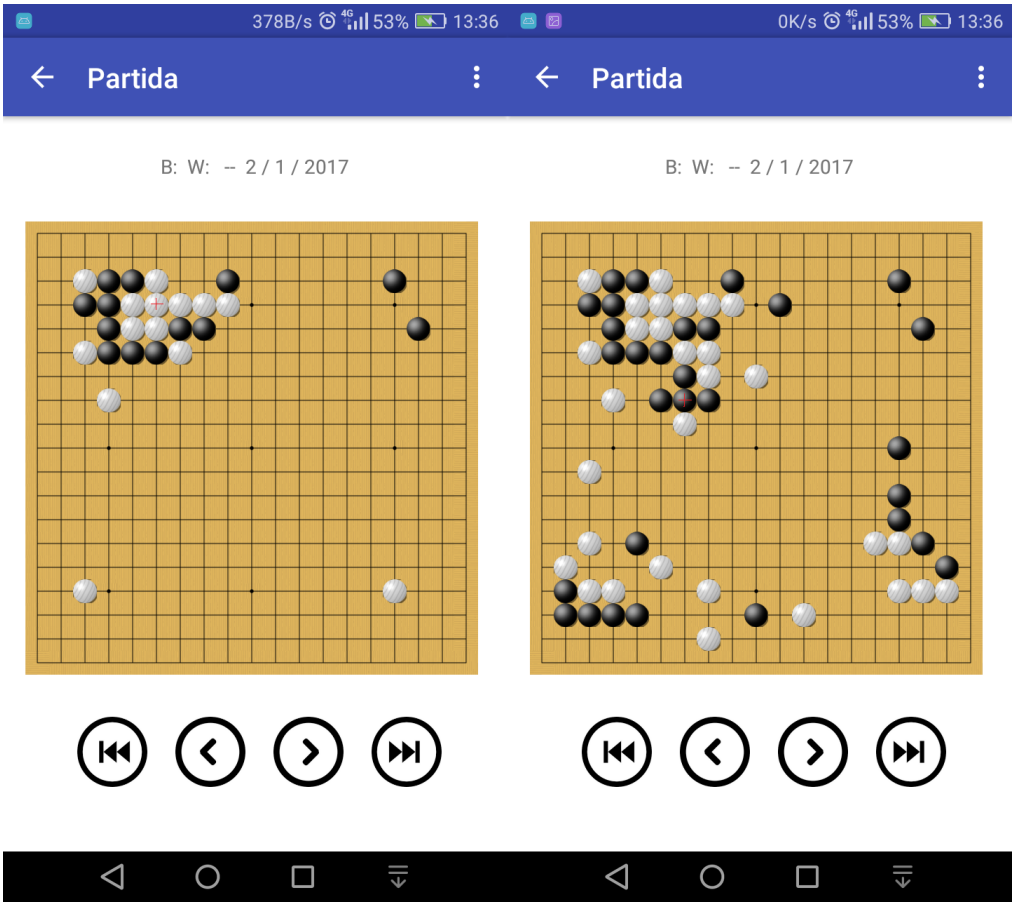

Figura 31: Jugadas interesantes.

En la figura 31 vemos dos jugadas que el algoritmo detectó sin problemas. De hecho, me sorprendió muchísimo, esperaba problemas en ambos casos. En el caso de la izquierda, aunque las jugadas blancas se detectan bien, no esperaba que en este caso con tantas piedras alrededor fuese capaz de detectar el círculo y menos con tanta seguridad como para detectar la jugada en el tiempo medio que suele tardar en detectar movimientos menos complicados.

Lo mismo pasó con la jugada negra en la imagen de la derecha. Al estar entre tres piedras amigas, pensé que tardaría al menos 3 o 4 segundos en

detectarla, pero al igual que con la blanca no tuvo problema alguno. Esto creo que es un gran avance de cara al problema que teníamos en la figura 27 , que al haber algo cerca ya dificultaba bastante la detección de jugadas negras.

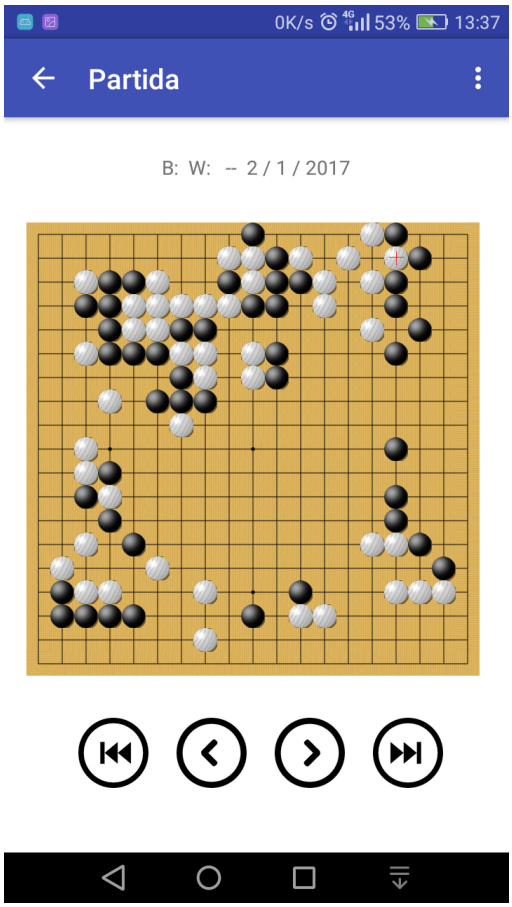

Figura 32: Ko en la esquina superior derecha.

En la figura 25 vemos una de las situaciones más famosas del juego, un ko[42]. Es interesante porque ocurre en casi todas las partidas, normalmente es un recurso para el jugador que va perdiendo, una forma de hacer 'all in', y no hubo ningún problema durante las aproximadamente 40 jugadas que duró el ko.

[42]Esto es una situación especial, en la que negro y blanco pueden comerse piedras mutuamente y la partida no terminaría nunca. Para ello se inventó la regla de ko, que dice que al hacer una jugada un jugador, la posición o estado de la partida no puede ser la misma que la que era cuando hizo su último movimiento.

La mala noticia de este experimento (aparte del fallo de memoria, pero ese era esperado) es que el movimiento de la figura 33, por razones que desconozco, no lo detectó hasta pasados unos 20 segundos (una cantidad de tiempo que no se puede permitir el algoritmo).

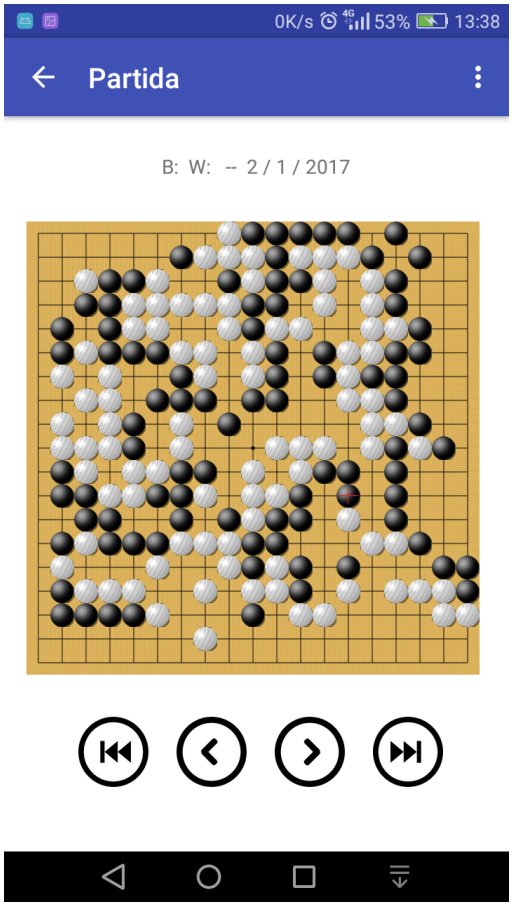

Figura 33: Movimiento difícil de detectar.

El balance general me parece muy bueno. Hemos detectado una partida casi completa, sin tener que estar pendiente del dispositivo en ningún momento, con un consumo de batería del 30 %[43] y sin fallos de memoria hasta el final de la partida, que realmente no es interesante grabar pues la parte más importante de analizar es el opening y a veces el mid-game.

[43]Un dato muy bueno ya que la batería del dispositivo no está en sus mejores condiciones.

5.3. Últimas pruebas

Parece que después de estas ultimas pruebas se tendrá la versión definitiva de la aplicación, a falta de algunos cambios menores.

5.3.1. Cambios aplicados

Hemos corregido el supuesto[44] 'fallo de memoria'. Realmente era un fallo del algoritmo, concretamente un acceso incorrecto a la lista de CompetitorPoints.

```
double circleX = circles.get(0,i)[0];
double circleY = circles.get(0,i)[1];
if(circleX < 24*19 + 12 && circleY < 24*19 + 12) {
        int index = (int) ((circleX - 12) / 24) * 19
           + (int) ((circleY - 12) / 24);
        CompetitorPoint cp =
           competitorPoints.get(index);
                .
                .
                .
```

Figura 34: Código corregido para evitar el IndexOutOfBoundsException.

Inicialmente no se comprobaba que los círculos detectados estuviesen dentro del tablero. Por eso, al no existir el if de la figura 34 , al detectar un círculo en una posición muy concreta debido a ruido de luces y otros factores, el valor de la variable index tomaba un valor inadecuado que podría resultar en un falso positivo más un posible falso negativo o en el peor caso, en un IndexOutOfBoundsException, haciendo reiniciar la aplicación y perdiéndose la posibilidad de guardar la partida.

Respecto al otro tema importante, la detección de piedras negras, se han combinado dos métodos:

Por un lado, se ha bajado un 25 % el valor que marca el umbral a la hora de detectar piedras negras. Esto tiene dos consecuencias: las piedras negras serán más fáciles de detectar, pero habrá muchísimos más falsos positivos.

[44]Index out of bounds exception. No tenía nada que ver con la memoria.

Para corregir estos falsos positivos, lo que se hace es tomar la media de los píxeles del círculo, aplicando antes un filtro de erosión[45]. Si esta media no pasa un umbral determinado, se considera que es un falso positivo y reiniciamos los votos de este.

Esta medida es muy efectiva puesto que la media entre los valores de las piedras negras y de los falsos positivos después de la erosión son muy distintos. El valor de la media de las piedras negras está en torno a 10 - 20 y los falsos positivos en torno a 125 - 225.

5.3.2. Descripción de los resultados

La primera impresión fue espectacular, todo parecía funcionar perfectamente. Sin embargo, al cabo de unas pruebas se detectaron algunos falsos positivos en el turno blanco[46]. Por tanto, hay que encontrar una solución para reforzar más la detección de piedras blancas. La solución que se aplica para los falsos positivos de las piedras negras no es muy efectiva en este caso ya que es difícil encontrar un límite entre los falsos positivos[47] y las piedras blancas para la media.

En cuanto a la detección de piedras negras, todo parece funcionar perfectamente, los falsos positivos se descartan en todos los experimentos, independientemente de la cantidad y foco de luz. En las próximas pruebas se pondrá atención a si todo funciona según lo previsto.

En cuanto a la aplicación en general, funciona sin ningún problema. Sin embargo, en una de las pruebas hubo algún error que no pude capturar y la aplicación se reinició. Actualmente me es difícil arreglar este tipo de errores, en la sección 6.2 se explica porqué.

En otras pruebas sin embargo, he tenido la aplicación funcionando durante más de una hora sin ningún problema, y desde entonces no ha vuelto a ocurrir.

[45]Sirve para eliminar el posible brillo sobre las piedras negras.

[46]En el turno negro al menos en los experimentos que he hecho no se ha dado ninguno.

[47]El valor de su media es bastante alto, algunos incluso alcanzando valores de piedras blancas.

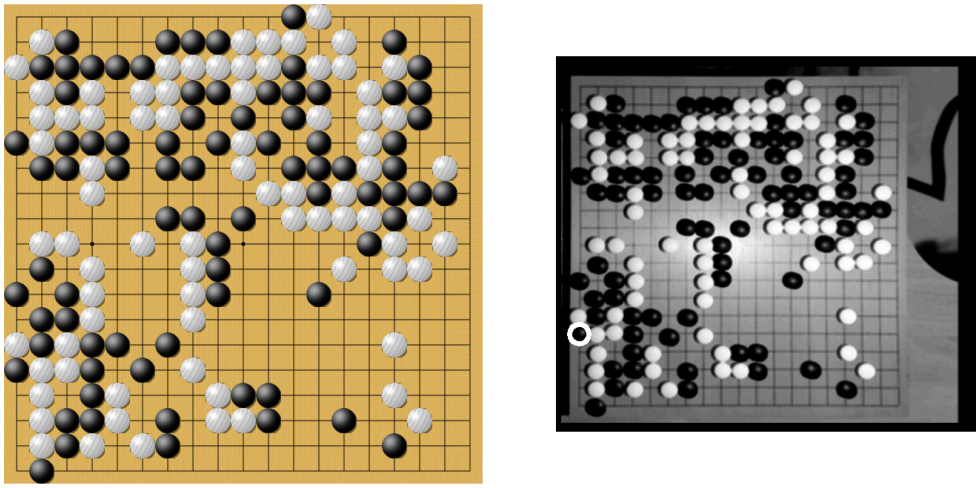

Figura 35: The 12^{th} Jeongkwanjang Cup. Main Round, played March 26, 2011.

La figura 35 muestra una de las partidas grabadas con éxito. Es una partida corta (165 movimientos) en la que blanco se rinde después de perder la batalla en la esquina inferior izquierda.

Para mejorar el tema de los falsos positivos se podría intentar dos cosas nuevas. Por un lado, hacer correlación cruzada entre un modelo y la submatriz que contiene la nueva jugada detectada. Si la correlación no pasa un cierto umbral, se deduce que es un falso positivo.

Para los falsos positivos que se van acumulando lentamente, se podría crear un contador en la clase CompetitorPoint que guarda el número de serie de la última imagen en la que se le dio un voto. Si hay mucha diferencia entre el número de serie actual y el anterior, se descarta el voto[48].

Al cabo de unas pruebas la correlación cruzada ha sido un éxito. El valor que da una piedra negra está en torno a 0.75, el de una piedra blanca en torno a 0.95 y el falso positivo que se quiere descartar en torno a 0.999. De esta forma se consigue eliminar los falsos positivos de blanco (al menos en los entornos en los que he probado hasta ahora la aplicación).

[48]Aunque esto podría perjudicar en algunos casos la detección de algunas jugadas difíciles para el algoritmo, habría que elegir cuidadosamente el parámetro de distancia entre números de serie.

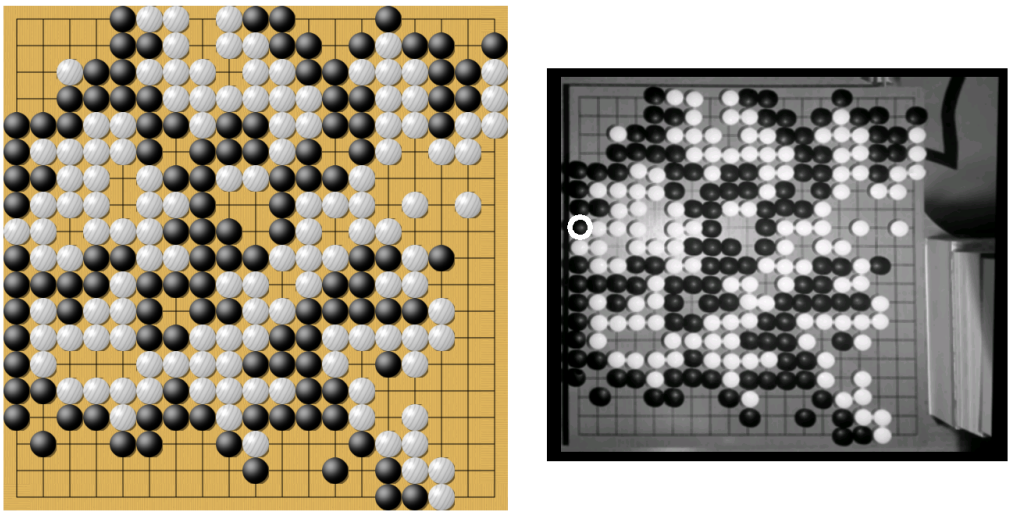

Figura 36: The 14^{th} Women's Kuksu. Semi-finals, played January 22, 2009.

En la figura 36 vemos una partida[49] bastante larga, de más de 250 movimientos, que se ha conseguido grabar sin ningún problema[50].

[49]La misma partida que vimos en el experimento anterior. Ahora no ha habido un solo problema.

[50]Gracias a la correlación cruzada no hay falsos positivos. Sin embargo, alguna vez alguna jugada de blanco se ha clasificado como falso positivo por lo que ha tardado más de la cuenta en detectarse (alrededor de 5 segundos cuando la media suele ser de 2 segundos).

6. Conclusiones

En esta sección se va a hablar aquí del resultado obtenido en general, tanto respecto a la aplicación como personal, como sobre que pienso se puede hacer para continuar con este trabajo.

6.1. Resultado final

En la sección 5 hemos visto el avance paso a paso según añadíamos contenido a la aplicación. El resultado final es bastante bueno. Es una base sobre la que se puede construir una aplicación robusta que pueda ser usada por cualquier usuario en cualquier dispositivo.

Actualmente hay que saber que hacer para que la aplicación funcione correctamente y, aún así, hay cosas que mejorar. Esto es normal, al fin y al cabo, los recursos de que disponemos son limitados, incluido el tiempo el cual sin duda ya he excedido[51], pero creo que con más recursos se puede obtener un producto muy bueno, libre en la medida de lo posible de errores.

6.2. Limitaciones

Mi equipo en concreto no me ha permitido desarrollar la aplicación de forma suficientemente eficiente. En concreto, al desarrollar toda la parte de visión no puedo usar el emulador (ya que no captura ninguna imagen, es un emulador al fin y al cabo) y mi dispositivo móvil en concreto no muestra las excepciones, reinicia la aplicación automáticamente.

Por esta razón por ejemplo, durante un tiempo pensé que la aplicación tenía problemas de memoria cuando realmente el fallo no tenía nada que ver con ella.

La solución de que disponía era usar otro dispositivo móvil[52], pero muy pocas veces lo tenía disponible y los errores tampoco ocurren siempre, dependen del entorno, la iluminación, etc.

Obviamente, también hay unas limitaciones físicas en este proyecto. Estamos realizando una aplicación orientada a ejecutarse en un dispositivo móvil, que tiene una velocidad de procesamiento y batería limitadas.

Estos algoritmos funcionarían mejor en dispositivos con una mayor velocidad de procesamiento, ya que necesitamos obtener una respuesta en el menor tiempo posible (normalmente alrededor de dos segundos es suficiente)

[51] Y no me arrepiento de ello en absoluto.

[52] Concretamente el de mi hermano.

y si podemos analizar imágenes más rápido podríamos aumentar las restricciones.

Además, para partidas de larga duración tenemos un problema con la batería del dispositivo, ya que no siempre puede estar conectado a una red eléctrica.

Añadido a todo esto están los factores que no podemos controlar, la iluminación o el ruido[53] entre otros.

Por último, recordar que el objetivo de este proyecto era grabar una partida de Go usando técnicas de visión por computador, concretamente usando la librería OpenCV. Esto es una limitación de por sí, ya que por ejemplo, para detectar jugadas en el tablero también podríamos hacer uso de redes neuronales.

6.3. Aprendizaje

Por supuesto he aprendido muchísimas cosas. Las podría clasificar en dos partes, una relativa a informática y otra relativa al trabajo personal.

Respecto a informática, algoritmos y protocolos he aprendido muchísimas cosas pero de todas ellas, creo que debo destacar tres:

- Android. Cuando empecé este proyecto no tenía ni idea de como hacer una aplicación para Android[54] y no solo he aprendido, si no que me ha encantado hasta el punto de querer enfocar mi carrera profesional en esa dirección.

- StackOverflow y Github. Son herramientas que conocía desde hace tiempo pero apenas había hecho uso de ellas anteriormente. No solo he aprendido a usarlas para este proyecto si no que he visto la verdadera utilidad que tienen. En el caso de Github me permitió volver atrás sobre mis pasos en multitud de ocasiones cuando había perdido ya la cuenta de los cambios que había hecho y teniendo el código totalmente ilegible. En el caso de StackOverflow, me ha servido para resolver multitud de problemas en cuestión de minutos, cuando por mi cuenta podría haber echado horas en resolverlos. Concretamente en el caso de

[53] Por ejemplo la manga de una camiseta negra con círculos blancos casualmente del mismo tamaño que las piedras...

[54] Por eso en parte quería hacer este proyecto, no solo iba a aprender sobre visión por computador.

Android hay muchísima documentación y problemas resueltos en esta web.

- Excepciones. Esto puede parecer un poco obvio, pero para mí no lo era hasta ahora. Nunca había hecho una aplicación de esta envergadura y por eso, no me había dado cuenta de cuan importante son las excepciones. El hecho de no poder verlas con mi dispositivo móvil me ha frustrado muchísimo y me ha hecho valorar lo que es tratar bien las excepciones.

Otras, como el uso de OpenCV por ejemplo, también son muy importantes, pero son de un ámbito más concreto, mientras que las tres mencionadas anteriormente pueden tener un peso muy grande a la hora de ayudarme en mi futuro cercano como desarrollador Android.

Respecto al ámbito personal, he aprendido lo que es hacer un proyecto de esta envergadura. Van a haber multitud de fallos inesperados, tanto de codificación como de material[55]. Voy a tener muchas dudas de distintos tipos y voy a tener que apañármelas para resolverlas de una forma u otra.

La planificación ha sido muy importante, y a veces por no haberla hecho bien no he podido avanzar al ritmo que quería. Esta no solo es importante de cara a la entrega de un proyecto si no también a la actitud de los trabajadores (en este caso yo). Cuando llevaba retraso mi actitud era más mala y trabajaba peor.

Por eso mismo, creo que he aprendido cosas de mi mismo al hacer este proyecto que creo es importante que conozca cuando en un futuro vaya a trabajar para una empresa, ya que estas situaciones se van a producir y tendré que dar lo mejor de mi no importa la situación.

Por último, también he aprendido a valorarme mejor a mi mismo, ya que he afrontado problemas, algunos de los cuales inicialmente pensaba no tendrían solución y finalmente los he resuelto prácticamente todos.

6.4. Continuación

Hay varias posibles mejoras que se podrían realizar para seguir haciendo este proyecto más robusto e interesante, hasta poder finalmente ser usado por usuarios ajenos a todo el proceso interior y convertirse en una aplicación de renombre dentro del mundo del Go.

[55]De nuevo, el móvil que se reinicia sin dar tiempo a ver las excepciones en AndroidStudio.

Se podría añadir un cálculo automático del ritmo de la partida. Esto es, conforme se van haciendo las jugadas, el algoritmo aprende el tiempo medio entre jugadas para adaptar sus restricciones a esto. El único problema es que a veces podrían hacerse un par de jugadas bastante rápido y este método podría perjudicar la detección de éstas, si antes ha habido mucho tiempo entre jugadas.

Se podría adaptar la aplicación para que funcione en dispositivos más grandes como tablets. De esta forma podríamos aprovechar la potencia de éstas en nuestro favor.

Se podría combinar todo el apartado de visión por computador con otros campos, como inteligencia artificial y redes neuronales para calcular mejor donde se ha hecho la siguiente jugada. Además se podrían añadir sistemas que calculen quien va ganando la partida[56] aproximadamente, y al final de ésta que calcule los puntos de cada jugador, sin necesidad de que lo hagan los jugadores.

Se podría añadir unas opciones para modificar el tipo de partida. Por ejemplo, partidas con handicap, one color[57] Go, cuatro + uno[58]...

En definitiva, hay varios caminos posibles hacia donde orientar esta aplicación. Espero tener tiempo en mi futuro cercano para seguir desarrollándola y poder usarla en los torneos a los que asista.

[56]Esto es complicado, uno de los grandes problemas de este juego es que hacer una buena heurística que tenga en cuenta todo es muy difícil, por no decir imposible.

[57]Ambos jugadores usan el mismo color, por lo que deben recordar que piedras son de cada uno.

[58]Si se hace un cuatro en raya, vuelve a ser tu turno.

A. Algoritmo de detección de capturas

Aquí se tiene el algoritmo completo para obtener las posiciones de las piedras capturadas por una jugada, dado el estado actual de la partida (con el movimiento incluido), la posición donde se ha jugado y el turno del jugador.

```
Set<Position> listaMuertas = new HashSet<>();
Set<Position> listaProvisional = new HashSet<>();
Set<Position> listaVisitas = new HashSet<>();
Set<Position> listaIntocable = new HashSet<>();
int i;
int j;

// Caso up
// Obtenemos la fila y columna de la piedra de arriba.
i = p.getFila()-1;
j = p.getColumna();
// Aniadimos esa posicion a la lista de visitas.
if(i >= 1) listaVisitas.add(new Position(i,j));
while(!listaVisitas.isEmpty()){
// Cojo la siguiente posicion de la lista de visitas.
Position pos = listaVisitas.iterator().next();
// Si en esa posicion en la matriz no hay nada o hay
   una piedra a un grupo vivo...
if(matriz[pos.getFila()-1][pos.getColumna()-1] == 0
   || listaIntocable.contains(pos)) {
// Limpiamos
listaProvisional.add(pos);
listaVisitas = new HashSet<>();
for (Position position : listaProvisional) {
listaIntocable.add(position);
}
listaProvisional = new HashSet<>();
}else if(listaMuertas.contains(pos)){
// En este caso simplemente vaciamos la lista de
   visitas pues todas estan muertas.
listaProvisional = new HashSet<>();
listaVisitas = new HashSet<>();
// Si por el contrario, hay una piedra amiga,
   aniadimos los cuatro vecinos a visitar y demas.
```

```
}else if(matriz[pos.getFila()-1][pos.getColumna()-1]
   != turno + 1){
listaProvisional.add(pos);
Position up = new Position(pos.getFila() -1,
   pos.getColumna());
Position down = new Position(pos.getFila() +1,
   pos.getColumna());
Position left = new Position(pos.getFila(),
   pos.getColumna() -1);
Position right = new Position(pos.getFila(),
   pos.getColumna() +1);
if(!listaProvisional.contains(up) && pos.getFila() >
   1) listaVisitas.add(up);
if(!listaProvisional.contains(down) && pos.getFila()
   < 19) listaVisitas.add(down);
if(!listaProvisional.contains(left) &&
   pos.getColumna() > 1) listaVisitas.add(left);
if(!listaProvisional.contains(right) &&
   pos.getColumna() < 19) listaVisitas.add(right);
}
listaVisitas.remove(pos);
}
if(!listaProvisional.isEmpty()){
for(Position pos : listaProvisional){
listaMuertas.add(pos);
}
}
listaProvisional = new HashSet<>();

// Caso down
// Obtenemos la fila y columna de la piedra de arriba.
i = p.getFila()+1;
j = p.getColumna();
// Aniadimos esa posicion a la lista de visitas.
if(i <=19 ) listaVisitas.add(new Position(i,j));
while(!listaVisitas.isEmpty()){
// Cojo la siguiente posicion de la lista de visitas.
Position pos = listaVisitas.iterator().next();
// Si en esa posicion en la matriz no hay nada o hay
   una piedra a un grupo vivo..
if(matriz[pos.getFila()-1][pos.getColumna()-1] == 0
```

```
   || listaIntocable.contains(pos)){
// Limpiamos
listaProvisional.add(pos);
listaVisitas = new HashSet<>();
for(Position position : listaProvisional){
listaIntocable.add(position);
}
listaProvisional = new HashSet<>();
// Si por el contrario, hay una piedra amiga,
   aniadimos los cuatro vecinos a visitar y demas.
}else if(matriz[pos.getFila()-1][pos.getColumna()-1]
   != turno + 1){
listaProvisional.add(pos);
Position up = new Position(pos.getFila() -1,
   pos.getColumna());
Position down = new Position(pos.getFila() +1,
   pos.getColumna());
Position left = new Position(pos.getFila(),
   pos.getColumna() -1);
Position right = new Position(pos.getFila(),
   pos.getColumna() +1);
if(!listaProvisional.contains(up) && pos.getFila() >
   1) listaVisitas.add(up);
if(!listaProvisional.contains(down) && pos.getFila()
   < 19) listaVisitas.add(down);
if(!listaProvisional.contains(left) &&
   pos.getColumna() > 1) listaVisitas.add(left);
if(!listaProvisional.contains(right) &&
   pos.getColumna() < 19) listaVisitas.add(right);
}
listaVisitas.remove(pos);
}
if(!listaProvisional.isEmpty()){
for(Position pos : listaProvisional){
listaMuertas.add(pos);
}
}
listaProvisional = new HashSet<>();

// Caso left
// Obtenemos la fila y columna de la piedra de arriba.
```

```
i = p.getFila();
j = p.getColumna()-1;
// Aniadimos esa posicion a la lista de visitas.
if( j>=1 ) listaVisitas.add(new Position(i,j));
while(!listaVisitas.isEmpty()){
// Cojo la siguiente posicion de la lista de visitas.
Position pos = listaVisitas.iterator().next();
// Si en esa posicion en la matriz no hay nada o hay
   una piedra a un grupo vivo..
if(matriz[pos.getFila()-1][pos.getColumna()-1] == 0
   || listaIntocable.contains(pos)){
// Limpiamos
listaProvisional.add(pos);
listaVisitas = new HashSet<>();
for(Position position : listaProvisional){
listaIntocable.add(position);
}
listaProvisional = new HashSet<>();
// Si por el contrario, hay una piedra amiga,
   aniadimos los cuatro vecinos a visitar y demas.
}else if(matriz[pos.getFila()-1][pos.getColumna()-1]
   != turno + 1){
listaProvisional.add(pos);
Position up = new Position(pos.getFila() -1,
   pos.getColumna());
Position down = new Position(pos.getFila() +1,
   pos.getColumna());
Position left = new Position(pos.getFila(),
   pos.getColumna() -1);
Position right = new Position(pos.getFila(),
   pos.getColumna() +1);
if(!listaProvisional.contains(up) && pos.getFila() >
   1) listaVisitas.add(up);
if(!listaProvisional.contains(down) && pos.getFila()
   < 19) listaVisitas.add(down);
if(!listaProvisional.contains(left) &&
   pos.getColumna() > 1) listaVisitas.add(left);
if(!listaProvisional.contains(right) &&
   pos.getColumna() < 19) listaVisitas.add(right);
}
listaVisitas.remove(pos);
```

```
}
if(!listaProvisional.isEmpty()){
for(Position pos : listaProvisional){
listaMuertas.add(pos);
}
}
listaProvisional = new HashSet<>();

// Caso right
// Obtenemos la fila y columna de la piedra de arriba.
i = p.getFila();
j = p.getColumna()+1;
// Aniadimos esa posicion a la lista de visitas.
if( j <= 19 ) listaVisitas.add(new Position(i,j));
while(!listaVisitas.isEmpty()){
// Cojo la siguiente posicion de la lista de visitas.
Position pos = listaVisitas.iterator().next();
// Si en esa posicion en la matriz no hay nada o hay
   una piedra a un grupo vivo..
if(matriz[pos.getFila()-1][pos.getColumna()-1] == 0
   || listaIntocable.contains(pos)){
// Limpiamos
listaProvisional.add(pos);
listaVisitas = new HashSet<>();
for(Position position : listaProvisional){
listaIntocable.add(position);
}
listaProvisional = new HashSet<>();
// Si por el contrario, hay una piedra amiga,
   aniadimos los cuatro vecinos a visitar y demas.
}else if(matriz[pos.getFila()-1][pos.getColumna()-1]
   != turno + 1){
listaProvisional.add(pos);
Position up = new Position(pos.getFila() -1,
   pos.getColumna());
Position down = new Position(pos.getFila() +1,
   pos.getColumna());
Position left = new Position(pos.getFila(),
   pos.getColumna() -1);
Position right = new Position(pos.getFila(),
   pos.getColumna() +1);
```

```
if(!listaProvisional.contains(up) && pos.getFila() >
   1) listaVisitas.add(up);
if(!listaProvisional.contains(down) && pos.getFila()
   < 19) listaVisitas.add(down);
if(!listaProvisional.contains(left) &&
   pos.getColumna() > 1) listaVisitas.add(left);
if(!listaProvisional.contains(right) &&
   pos.getColumna() < 19) listaVisitas.add(right);
}
listaVisitas.remove(pos);
}
if(!listaProvisional.isEmpty()){
for(Position pos : listaProvisional){
listaMuertas.add(pos);
}
}

return listaMuertas;
```

B. Método onCameraFrame

A continuación el código completo del método onCameraFrame, que se ejecuta una vez por cada captura de pantalla.

```
mRgba = inputFrame.rgba();

if(detectingCircles) {

gray = mRgba.clone();
Imgproc.cvtColor(mRgba, gray, Imgproc.COLOR_BGR2GRAY,
   0);

Imgproc.dilate(gray, gray,
   Imgproc.getStructuringElement(Imgproc.MORPH_CROSS,
   new Size(3, 3)));
Imgproc.GaussianBlur(gray, gray, new Size(13, 13), 2,
   2);

HoughCircles(gray, circles, CV_HOUGH_GRADIENT, 1, 45,
   30, 20, 5, 15);

for (int i = 0; i < circles.cols(); i++) {
circle(mRgba, new Point(circles.get(0, i)[0],
   circles.get(0, i)[1]), (int) circles.get(0, i)[2],
   new Scalar(89, 255, 202), 5);

// Para cada uno de los casos.
if(leftDownCorner){
// Si se da el caso de ser la primera vez que se
   detecta:
if(leftDownCornerCount == 0){
leftDown = new Point(circles.get(0, i)[0],
   circles.get(0, i)[1]);
leftDownCornerCount++;
}else{
double cond1 = circles.get(0,i)[0] - leftDown.x;
double cond2 = circles.get(0,i)[1] - leftDown.y;
if (cond1 < 10 && cond1 > -10 && cond2 < 10 && cond2
   > -10){
double xCoord = leftDown.x;
```

```
double yCoord = leftDown.y;
xCoord = (xCoord*leftDownCornerCount +
   circles.get(0,i)[0]) / (leftDownCornerCount + 1);
yCoord = (yCoord*leftDownCornerCount +
   circles.get(0,i)[1]) / (leftDownCornerCount + 1);
leftDown = new Point(xCoord,yCoord);
leftDownCornerCount++;
}
}
}
if(leftUpCorner){
if(leftUpCornerCount == 0){
leftUp = new Point(circles.get(0, i)[0],
   circles.get(0, i)[1]);
leftUpCornerCount++;
}else{
double cond1 = circles.get(0,i)[0] - leftUp.x;
double cond2 = circles.get(0,i)[1] - leftUp.y;
if (cond1 < 10 && cond1 > -10 && cond2 < 10 && cond2
   > -10){
double xCoord = leftUp.x;
double yCoord = leftUp.y;
xCoord = (xCoord*leftUpCornerCount +
   circles.get(0,i)[0]) / (leftUpCornerCount + 1);
yCoord = (yCoord*leftUpCornerCount +
   circles.get(0,i)[1]) / (leftUpCornerCount + 1);
leftUp = new Point(xCoord,yCoord);
leftUpCornerCount++;
}
}
}
if(rightDownCorner){
if(rightDownCornerCount == 0){
rightDown = new Point(circles.get(0, i)[0],
   circles.get(0, i)[1]);
rightDownCornerCount++;
}else{
double cond1 = circles.get(0,i)[0] - rightDown.x;
double cond2 = circles.get(0,i)[1] - rightDown.y;
if (cond1 < 10 && cond1 > -10 && cond2 < 10 && cond2
   > -10){
```

```
double xCoord = rightDown.x;
double yCoord = rightDown.y;
xCoord = (xCoord*rightDownCornerCount +
   circles.get(0,i)[0]) / (rightDownCornerCount + 1);
yCoord = (yCoord*rightDownCornerCount +
   circles.get(0,i)[1]) / (rightDownCornerCount + 1);
rightDown = new Point(xCoord,yCoord);
rightDownCornerCount++;
}
}
}
if(rightUpCorner){
if(rightUpCornerCount == 0){
rightUp = new Point(circles.get(0, i)[0],
   circles.get(0, i)[1]);
rightUpCornerCount++;
}else{
double cond1 = circles.get(0,i)[0] - rightUp.x;
double cond2 = circles.get(0,i)[1] - rightUp.y;
if (cond1 < 10 && cond1 > -10 && cond2 < 10 && cond2
   > -10){
double xCoord = rightUp.x;
double yCoord = rightUp.y;
xCoord = (xCoord*rightUpCornerCount +
   circles.get(0,i)[0]) / (rightUpCornerCount + 1);
yCoord = (yCoord*rightUpCornerCount +
   circles.get(0,i)[1]) / (rightUpCornerCount + 1);
rightUp = new Point(xCoord,yCoord);
rightUpCornerCount++;
}
}
}
}
}

if(!startRecording && !calculateHomography) {
if (leftDownCornerCount > 0) {
circle(mRgba, leftDown, 1, new Scalar(200, 80, 202),
   2);
}
if (leftUpCornerCount > 0) {
```

```
circle(mRgba, leftUp, 1, new Scalar(200, 80, 202), 2);
}
if (rightDownCornerCount > 0) {
circle(mRgba, rightDown, 1, new Scalar(200, 80, 202),
   2);
}
if (rightUpCornerCount > 0) {
circle(mRgba, rightUp, 1, new Scalar(200, 80, 202),
   2);
}
} else if (startRecording && !pausedRecording){
// RECORDING

//DETECT CIRCLES
Imgproc.warpPerspective(mRgba,mRgba,homographyMatrix,tamMats);
Imgproc.cvtColor(mRgba, gray,
   Imgproc.COLOR_BGR2GRAY,0);

// CORRELACION
// Si es la primera vez que estamos grabando, creamos
   la lista de mats.
if(firstRecording){
for(CompetitorPoint cp : competitorPoints){
correlation.add(gray.submat((int) cp.getMinY() +
   3,(int)cp.getMaxY() - 3,(int) cp.getMinX() +
   3,(int) cp.getMaxX() - 3));
}
firstRecording = false;
}

Imgproc.GaussianBlur(gray, gray, size3x3, 2);
// En funcion del turno erosionamos o dilatamos la
   imagen.
if(!turno){ // Estamos buscando piedras negras.
dilate(gray,gray,Imgproc.getStructuringElement
   (Imgproc.MORPH_RECT, size2x2));
}else{ // Estamos buscando piedras blancas.
erode(gray,gray,Imgproc.getStructuringElement
   (Imgproc.MORPH_RECT, size2x2));
}
// HEMOS BAJADO DE 20 A 15 PARA NEGRAS
```

```
if(!turno){
HoughCircles(gray, circles, CV_HOUGH_GRADIENT, 1, 5,
   30, 15, 9, 13);
}else{
HoughCircles(gray, circles, CV_HOUGH_GRADIENT, 1, 5,
   30, 20, 9, 13);
}

// Dibuja la homografia
if(showHomography){
for(int row = 0; row < 19; row++){
for(int column = 0; column < 19; column++){
circle(mRgba, homography.get(row*19 + column), 1, new
   Scalar(89, 255, 202), 2);
}
}
}

//FOR EVERY POINT MATCH CIRCLES
for(int i = 0; i < circles.cols(); i++){
double circleX = circles.get(0,i)[0];
double circleY = circles.get(0,i)[1];
// Con este if evitamos mirar un ciruclo donde no hay
   intersecciones (debido a fallos de luces).
if(circleX < 24*19 + 12 && circleX >= 12 && circleY <
   24*19 + 12 && circleY >= 12) {
int index = (int) ((circleX - 12) / 24) * 19 + (int)
   ((circleY - 12) / 24);
CompetitorPoint cp = competitorPoints.get(index);
if (!cp.isFound()) {
int stack = cp.getStack();
cp.setStack(stack + 1);
if (ganadorDeRonda.getStack() < cp.getStack()) {
ganadorDeRonda = cp;
}
}
}
}

if(lastPlayed != null){
int indexLastPlay = lastPlayed.getId();
```

```
circle(gray, new Point(24 + (indexLastPlay /
   19)*24,24 + (indexLastPlay%19)*24), 13, red , 5);
}

// SI SE DETECTA UN MOVIMIENTO
if (ganadorDeRonda.getStack() >= 10) {
Mat pruebaMat = gray.submat((int)
   ganadorDeRonda.getMinY() +
   3,(int)ganadorDeRonda.getMaxY() - 3,(int)
   ganadorDeRonda.getMinX() + 3,(int)
   ganadorDeRonda.getMaxX() - 3);
erode(pruebaMat,pruebaMat,Imgproc.getStructuringElement
   (Imgproc.MORPH_RECT, size3x3));
Scalar mean = Core.mean(pruebaMat);
//CORRELACION
Mat plantilla =
   correlation.get(ganadorDeRonda.getId()).clone();
// / Create the result matrix
int result_cols = 1;
int result_rows = 1;
Mat result = new Mat(result_rows, result_cols,
   CvType.CV_32FC1);
matchTemplate(plantilla,pruebaMat,result,
   Imgproc.TM_CCORR_NORMED);
Scalar meanResult = Core.mean(result);
plantilla.release();
result.release();
//FIN CORRELACION
pruebaMat.release();
// Este if sirve para descartar falsos positivos y
   para que se guarden en el orden correcto.
if(((!turno && mean.val[0] < 100) || (turno &&
   mean.val[0] > 100)) && (meanResult.val[0] <=
   0.995)) {
mensajeDeApoyo = "Valor de la correlacion: " +
   meanResult.val[0];
mp.start();
// TENEMOS GANADOR
lastPlayed = ganadorDeRonda;
ganadorDeRonda.setFound(true);
String turnoColor;
```

```
if (!turno) {
turnoColor = "B";
} else {
turnoColor = "W";
}
partida = partida + ";" + turnoColor + "[" +
   ganadorDeRonda.getPosition() + "]";
// CALCULO DE PIEDRAS MUERTAS.
eliminarMuertas(ganadorDeRonda);
turno = !turno;
for (int i = 0; i < competitorPoints.size(); i++) {
cpAuxiliar = competitorPoints.get(i);
cpAuxiliar.setStack(0);
}
}else{
ganadorDeRonda.setStack(0);
if(turno && mean.val[0] > 100){
mensajeDeApoyo = "Valor de la correlacion: " +
   meanResult.val[0];
countWhiteFalsePositive++;
}
}
}
mRgba = gray.clone();
}else{
// PAUSED
Toast.makeText(this,"Grabacion en
   pausa",Toast.LENGTH_LONG).show();
}
circles.release();
gray.release();
return mRgba;
```

C. Código de la aplicación

El código de la aplicación está disponible en:

`https://github.com/Trethtzer/Proyecto`

Todo el material informático usado para el desarrollo de este proyecto se encuentra ahí.

Referencias

[1] http://gobi.webnode.es/una-breve-historia-del-go/

Web con información sobre el go y su historia.

[2] http://aego.biz/

Web de la asociación española de go.

[3] http://senseis.xmp.net/

Web con muchísima información acerca de todos los aspectos del go (una de las más conocidas en todo el mundo respecto a este juego).

[4] http://stackoverflow.com/

Web con muchísima información acerca de todo lo relativo a la parte de programación. Cada vez que tenía un problema consultaba diversos hilos de esa página, la inmensa mayoría con resultados positivos.

[5] https://www.udacity.com/

Página web con cursos y nanodegrees para aprender a programar en Android.

[6] http://dis.um.es/~alberto/material/percep.pdf Sistemas de percepción y visión por computador. Alberto Ruiz García.

Documento con bastante información sobre técnicas de visión por computador.

[7] Wikipedia.

Útil para corroborar algunos datos y definiciones.

[8] http://opencv.org/

Página oficial de la librería OpenCV, que he usado para el desarrollo de la parte de visión por computador.

[9] http://tex.stackexchange.com

Página con multitud de información sobre LaTeX, necesaria para la realización de este documento.

[10] Outside the Board - Diary of a Professional Go Player. Hajin Lee.

Libro de uno de mis jugadores de Go favoritos. Las partidas con las que he hecho pruebas son del apéndice de este libro.

[11] Detección de circunferencias. Transformada de Hough - Universidad de Valladolid

Documento que explica en que consiste las transformadas de Hough.

[12] Computer Vision: Algorithms and Applications - Richard Szeliski

Libro con multitud de información sobre diversas técnicas de visión por computador.