DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Procesamiento de imágenes en videos de fútbol: una aproximación a la detección de jugadores en tiempo real

14/04/2016 Trabajo Práctico Final - Organización del Computador II

Integrante	LU	Correo electrónico
Rodriguez, Leticia Lorena	10/03	lrodrig@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
http://exactas.uba.ar

Índice

1. Introducción

Si la televisión hubiese sido inventada con la sola finalidad de transmitir fútbol en directo, ya estaría justificada ˜ Roberto Fontanarrosa

El procesamiento digital de imágenes ha sido una área de amplio crecimiento y con infinidad de aplicaciones en los últimos años.

En el presente trabajo se estudia la utilización de una combinación de estos algoritmos para la detección de jugadores en videos de fútbol en tiempo real.

La naturaleza de una solución en tiempo real hace que se cuente de un espacio de tiempo pequeño situado dos frames consecutivos en el cual se deberá llevar adelante todo el procesamiento necesario. Por lo cual, se hace conveniente la utilización de un lenguaje cercano a la arquitectura de la computadora, como el lenguaje ensamblador, que permita operar las imágenes en forma más rápida.

En particular, se desarrollaron implementaciones de estos algoritmos en Intel x64 usando la arquitectura SIMD y fueron probados en una computadora con microprocesador Intel Core i3.

2. Análisis del problema

2.1. Limitación en tiempos de procesamiento

El objetivo del trabajo es detectar eficientemente los jugadores en un video de partido de fútbol de una transmisión de televisión.

El proceso principal esta hecho en C++ utilizando la librería OpenCV mediante la cual se abre el video, se determina el tamaño de cada frame y los FPS.

Luego, se va leyendo imagen por imagen. Cada una es enviada para su análisis en Assembler y devuelto el resultado al C++ donde se procede a su visualización. Este proceso debe llevar exactamente 1/FPS segundos por frame para que el video sea mostrado en pantalla en velocidad normal.

Esos 1/FPS segundos van a estar compuestos por el tiempo de procesamiento de la imagen en Assembler, y en caso que sobre tiempo, un tiempo de espera idle.

Por ejemplo, un video de 25 FPS va a tener que mostrar las imágenes cada 1/25 segundos (40 milisegundos) y ese será el máximo tiempo que podrá tardar la detección de los jugadores. Si el procesamiento en Assembler tardara 25 milisegundos, habría 15 milisegundos de espera antes de mostrar el siguiente frame.

Si tardara más, sería imposible cumplir con el requerimiento de FPS del video, dando un efecto de lentitud en la ejecución del video.

2.2. Dificultades en la detección de los jugadores

A fin de quedarse únicamente con los jugadores de fútbol, dentro de cada imagen va a haber una variedad de elementos a descartar: hinchada, arcos, lineas del cesped, carteles de publicidad, arbitros, pelota, pasto, papelitos, etc.

Incluso, muchas transmisiones incluyen información del marcador, logo televisivo, publicidad, etc.

También, se pueden observar diferentes tomas:

- Close-Up : Close Up de jugadores, árbitros, técnicos, etc. Incluyen tomas relativas a la repetición o análisis de faltas. (descartar)

- Animaciones : Tomas con animaciones superpuestas. (descartar)

- Campo de juego: Frames que muestran el campo de juego y los jugadores. Este es el tipo de frame principal a analizar.

- Hichada: Tomas de la tribuna (descartar)

La estrategia de detección de jugadores debera sortear dichos frames y elegir las tomas que muestren en campo de juego.

3. Solución propuesta

Una solución tentativa al problema propone la aplicación de forma ordenada de una serie de filtros que van eliminando los objetos no deseados de la imagen para quedarse con aquellos que sean jugadores de fútbol.

A su vez, algunos de estos filtros, utilizan de otros para mejorar sus resultados, por ejemplo, antes de una etiquetación de regiones se aplica dilatación para agrupar mejor las regiones.

También, se aplican filtros de conversiones de color, escala de grises o blanco y negro. Así como filtros combinan imágenes.

El trabajo práctico propone la implementación de una serie de filtros de manera independiente en Assembler para ser combinados en C++ en diferentes procesos que se ejecutan secuencialmente sobre cada frame del video.

Esto permite estudiar en diferentes videos, distintos procesamientos en busca del más efectivo para la mayor parte de los videos.

La desventaja es el costo de procesamiento dado que un proceso que este compuesto por N filtros, que si bien estan implementados en Assembler, al ir y venir del C++, producen N cambios de contexto (context switch) lo cual hace que sea más lento que si se escribiera un único Assembler con todos los filtros ejecutandose uno tras otro. Sin embargo, esta lentitud raramente supera el intervalo de tiempo disponible entre dos frames.

El práctico fue diseñado de esta manera para estudiar el efecto de cada filtro sobre la imagen pero una vez seleccionada la secuencia de procesamiento que maximice los resultados, podría ser completamente implementada en lenguaje ensamblador mejorando muchísimo los tiempos de ejecución.

3.1. Filtros implementados

3.1.1. Eliminación de cesped

Existe una variedad de trabajos que estudian formas de eliminar el cesped de una imágen de un campo de juego de fútbol. Generalmente, se utiliza la regla:

$$Ground(x,y) = \begin{cases} 0, & (g(x,y) > r(x,y) > b(x,y)) \\ 1, & otherwise \end{cases}$$

Otras aproximaciones incluyen la elaboración de un histograma de píxeles.

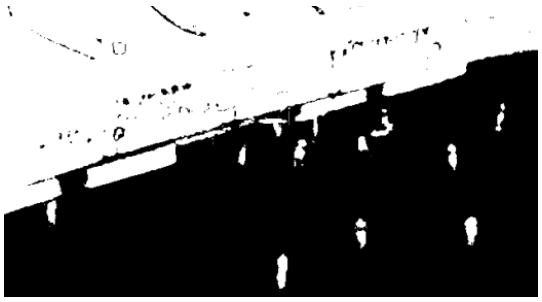
Teniendo en cuenta la velocidad de procesamiento, la necesidad de elaborar estrategias en tiempo real y la cantidad de trabajos que avalaban los resultados, se optó por la aplicación de la fórmula que verique los colores.

Las distintas gamas de verde de los estadios y de las transmisiones deportivas, ocasiona que el filtro no funcione para todos los casos y precise algún tipo de ajuste que permita obtener de la forma más nítida posible los jugadores.

Este ajuste, sirve para todos los frames contenidos en un video y se trata de un *offset* que es aplicado como corrección a la gama de colores.

A continuación, se incluyen ejemplos de la eliminación de cesped en un mismo frame para dos valores de *offset*.

Offset = 0



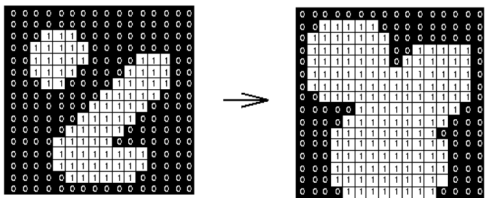
Offset=31 Mejora los resultados, agrupando más los píxeles en blanco.

Lamentablemente, en el trabajo no se incluye una detección automática del *offset* en base al video a analizar. La elección va a ser en forma manual. Se puede tomar para un futuro trabajo.

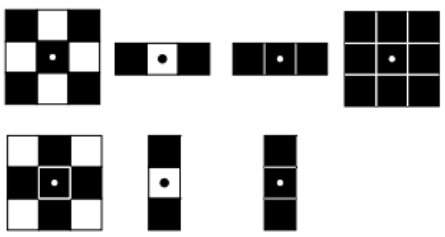
3.1.2. Dilatación

El algoritmo pone en 1 un pixel si alguno de sus vecinos es 1.

1	1	1
1	*	1
1	1	1



En el ejemplo anterior, se utilizó un cuadrado 3x3 como elemento estructura. Los elementos estructura suelen ser pequeños en relativo a la imagen y pueden tener forma de cruz, cuadrado, etc.

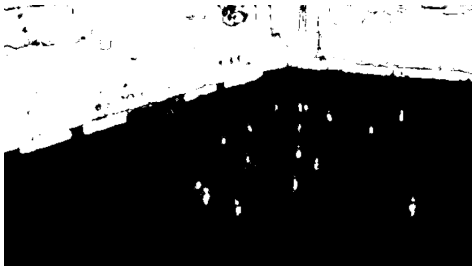


Ejemplos de estructuras binarias simples. Los píxeles en negro son parte del elemento, los blancos no. El pixel con el punto marca el origen.

Para el presente trabajo, se eligió una estructura en forma de cruz.

La idea es eliminar huecos en secciones grandes de color blanco o darle mayor definicion a las áreas.

Por ejemplo, aplicado a las imágenes de un partido se puede dar mayor definición a los jugadores y unificar la sección correspondiente a la tribuna.



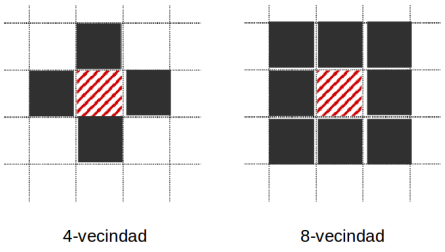
(A) Imagen en blanco y negro sin cesped

(B) Imagen (A) aplicando 6 veces el filtro de dilatación

3.1.3. Etiquetación de regiones

Esta técnica tiene por objetivo agrupar los píxeles en blanco contiguos en regiones utilizando una misma etiqueta o color.

Existen varias variantes del algoritmo. Por ejemplo:



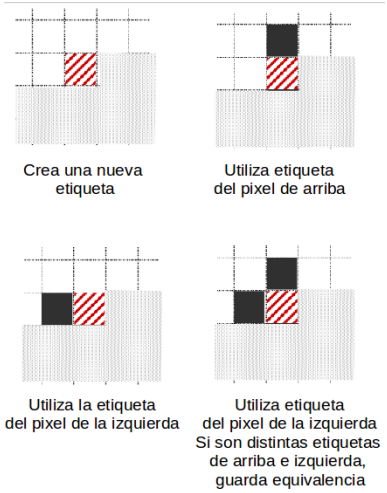
Variantes de Etiquetación de Regiones

Para la variante 4-vecindad, el algoritmo sería el siguiente:

1. Recorrer la imagen pixel por pixel llevando a cabo las acciones:

   a) Si el pixel actual es 0, no se hace nada.

   b) Si el pixel actual es 1, se evalua su vecindad de la siguiente manera:



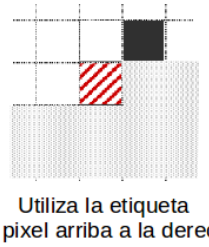
Variantes de Etiquetación de Regiones

2. Realizar una segunda pasada por toda la imagen para reemplazar la etiqueta equivalente con una misma etiqueta según las equivalencias calculadas en el paso anterior.

Se optó por la variante 4-vecindad pero los resultados no fueron demasiado buenos. Como se trabajó con la etiquetación en grayscale, que tan sólo se cuenta con 255 colores disponibles, no se podrían etiquetar más de 255 regiones distintas.

Esto último ocasiona que se precise un algoritmo que trate de etiquetar correctamente la mayor cantidad de píxeles en la primer pasada para no quedarse sin colores y que se dificulte la pasada de corrección.

Por lo tanto, se tomó también el pixel en el extremo superior derecho como posible etiqueta cuando no haya etiquetas ni arriba ni a la izquierda del pixel actual.



Modificación a 4-vecindad

Por último, cabe señalar que la mayor parte de los algoritmos trabaja en blanco y negro o escala de grises para mejorar los tiempos de ejecución dado que se reduce significativamente la cantidad de píxeles a revisar.

(A) Imagen original

(B) Imagen después de aplicar region labeling

(C) Imagen después de aplicar region labeling
(coloreada para visualizar en el informe)

3.1.4. Contabilización de píxeles en regiones

El algoritmo toma una imagen segmentada en regiones, en escala de grises y contabiliza la cantidad de píxeles que contiene cada región.

La resultante del procesamiento es un listado valores de colores en escala de grises junto con la cantidad de píxeles de cada color encontrados en la imagen y ordenados de mayor a menor.

La finalidad es brindar información sobre el tamaño de las regiones para su análisis posterior. Podría servir, por ejemplo, para buśqueda de relaciones entre las regiones y la consecuente elaboración de filtros.

3.1.5. Filtrado de regiones

Este algoritmo recibe un rango de cantidad de píxeles y filtra aquellas regiones cuyo tamaño este fuera del rango especificado.

Para funcionar precisa tener como entrada una imágen en escala de grises en la cual se hayan coloreado regiones y la información correspondiente de cuantos píxeles hay de cada color.

3.1.6. Copia de imágenes

Copia una imagen de entrada a otra de salida, es decir, copia los bytes entre dos posiciones de memoria. El último parámetro indica si la imagen a copiar es color o no.

3.1.7. Conversión a blanco y negro

Toma una imagen en escala de grises y la convierte a blanco y negro.

3.1.8. Detección de bordes

Existen varias publicaciones que abordan el problema de la detección de contornos en imágenes. Se basan principalmente en detectar las variaciones fuertes de intensidad que corresponden a las fronteras de los objetos.

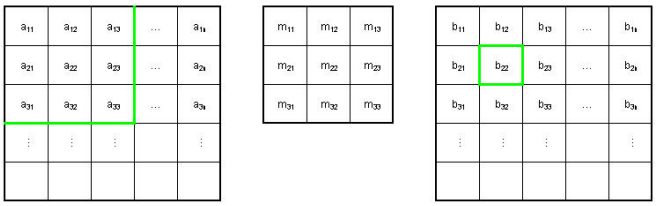
Para este trabajo, se optó por un método basado en gradientes llamado Sobel.

Sobel define dos máscaras: Gx y Gy que se aplican a los píxeles de la imagen A.

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \qquad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

Mascaras para el filtro Sobel

La máscara Gx resalta los bordes en dirección horizontal mientras que la máscara Gy resalta los bordes en dirección vertical.



Aplicación de mascara para el filtro Sobel

Cada máscara es aplicada sobre los mismos píxeles de la imagen y son combinados para detectar los bordes en ambas direcciones con la siguiente fórmula:

$$\mathbf{G} = \sqrt{\mathbf{G}_x{}^2 + \mathbf{G}_y{}^2}$$

Combinación de bordes horizontal y vertical

3.1.9. Marcar contornos en una imagen a partir de otra

Filtro diseñado para el trabajo práctico. Recibe dos imágenes, la imagen color de la transmisión de fútbol y una imagen blanco y negro con las siluetas de los objetos que se quieren resaltar.

Básicamente, el filtro las compone resultando la imagen del partido original con los píxeles en blanco de la segunda imagen, marcados en esta en color naranja.

3.2. Procesos

Para el presente trabajo, se implementó en C++ un *proceso* que va llamando en forma secuencial a los filtros anteriomente detallados. La idea es que la resultante de un proceso, sea una imagen color con el cortorno de los jugadores, partiendo de la imagen del video origen (frame).

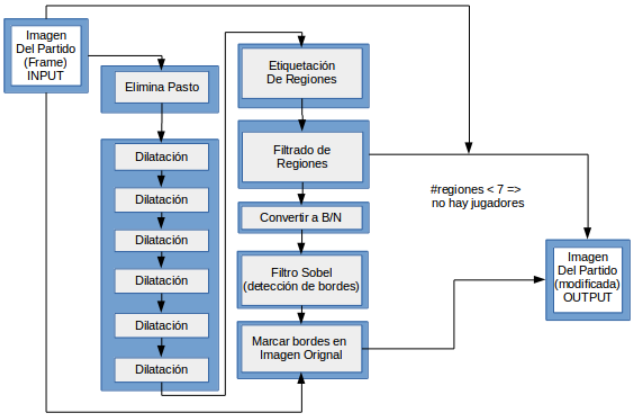
Vale comentar que el intercambio de contexto entre C++ y el Assembler, llamado switch context hace que se vean degradados los tiempos de ejecución en comparación a un implementación del proceso completo en lenguaje ensamblador.

Queda fuera del alcance de este trabajo brindar las implementaciones en lenguaje ensamblador de los procesos completos.

Así el trabajo brinda una plataforma extensible, disponible para la experimentación de distintas secuencias de filtros e incluso agregado de nuevos para el estudio de una mejor técnica para la detección de jugadores. Se podría partir de este trabajo para una futuro análisis de otras variantes de procesos.

3.2.1. El Proceso

Siguiendo las ideas de trabajos publicados ([1][2], otros) se diseñó la siguiente secuencia de filtros que constituyen un *Proceso*.



Pasos del *Proceso*

3.3. Código en lenguaje ensamblador Intel x64

Todos los filtros fueron implementados en lenguaje ensamblador aprovechando las ventajas que brinda la técnica SIMD de los procesadores Intel.

Se explican algunas implementaciones a fin de brindar información sobre como se trabajó con la arquitectura.

3.3.1. Eliminación de cesped

Como se detalló en la sección anterior, para eliminar el cesped se realiza una comparación entre los bytes R, G y B que componen cada pixel.

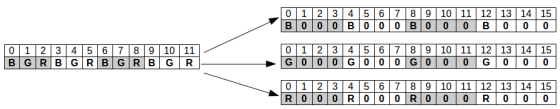
Se lee la imagen de memoria llenando un registro XMM de 128 bits (16 bytes). Al tratarse de una imagen en color, los bytes tendrán al inicio, la siguiente forma:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B	G	R	B	G	R	B	G	R	B	G	R	B	G	R	B

Lectura de píxeles color imagen a registro

Se observan cinco píxeles completos y el último incompleto. Se optó por hacer tres lecturas para leer 16 píxeles de tres bytes completos.

Estos 16 píxeles leídos van a ser procesados de a cuatro por vez. El primer lote de cuatro píxeles, es separado por banda (R,G o B) y dispuesto en tres registros XMM de la siguiente manera:



Separación en bandas

Estos tres registros se comparan utilizando funciones SIMD y luego, son agrupados en un registro tomando los 4 píxeles blanco y negro resultantes.

Se van desplazando los registros leídos de la memoria para ir alineando los píxeles de a cuatro. Primero, se arman los píxeles 12, 13, 14 y 15, se comparan y se escribe el resultado en un registro. Luego, se prosigue por los píxeles 8,9,10,11 , despus 4,5,6,7 y finalmente, 0, 1, 2, 3 cuyos resultados se van agregando al registro XMM final que va ha ser escrito en la dirección destino indicada al filtro.

El proceso se repite para todos los píxeles de la imagen.

3.3.2. Dilatación

Este proceso, también va a requerir hacer tres lecturas a la imagen, pero a diferencia del anterior, las lecturas no van a ser contiguas sino que se va a utilizar los píxeles que esten en la linea de arriba, misma línea y debajo del píxel que se quiera actualizar.

El procesamiento se hace sobre una imagen blanco y negro por lo cual, cada color, va a estar compuesto por 1 byte. Dicho byte va a tener dos valores posibles: 0xFF o 0x00.

Según lo descripto en la sección anterior, para determinar el color de píxel se precisa el vecino que se encuentra arriba, arriba a la derecha y a la izquierda del mismo.

Se va a armar un registro adicional desplazando en 1 byte el registro del medio, para generar un registro que contenga aquellos píxeles que se encuentren a la izquierda.

Esto va a hacer que por cada lectura haya tan sólo 14 píxeles que se puedan actualizar con el valor calculado.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
b0	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15	Medio
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
X	b0	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	Izquierda
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
a0	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	a11	a12	a13	a14	a15	Arriba
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	a11	a12	a13	a14	a15	X	Arriba Derecha
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
c0	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15	Abajo

Acomodamiento de píxeles para operar en el filtro Dilatación

Por lo cual, en esta implementación hay especial cuidado con los píxeles de los bordes y aquellos que contienen datos erroneos.

3.3.3. Conversión a blanco y negro

Toma una imagen en escala de grises y la convierte en blanco y negro.

Se iteran los píxeles de la imagen leyendo 16 bits de la memoria y almacenándolo en un registro XMM.

Cada byte corresponde a un pixel en escala de grises. Para convertirlos en blaco y negro, se los comparan con una máscara de 0s. Si es mayor enciende el byte en el registro, sino, lo deja en 0.

La operación de Intel utilizada trabaja con datos de tamaño WORD por lo cual, antes de operarlos se procede a desempaquetar los bytes en dos registros XMM.

Finalmente, luego de la comparación, se empaquetan ambos registros.

3.3.4. Etiquetación de regiones

Es el filtro más complejo para implementar. Según los explicado anteriormente, hace dos pasadas sobre la imagen blanco y negro de entrada.

En la primer pasada, para cada pixel puede asignar un nuevo color, tomar el color del pixel de arriba, del costado o de arriba a la derecha, esto va a depender si alguno de estos tiene color y si el pixel actual es 1. Se toma nota de equivalencias de color si difiere entre los píxeles que lo rodean.

En la segunda pasada, recorre los píxeles y los actualiza según las equivalencias encontradas durante la primer pasada.

Dado que el color que va tomar el pixel actual depende de los que se encuentran arriba o al costado, debe ir actualizandose pixel por pixel, byte por byte. Esto presenta un inconveniente para aprovechar el procesamiendo en paralelo bytes que posibilita la técnica SIMD.

Por lo cual, este método fue implementado realizando lecturas de 16 bytes pero procesando byte por byte.

Adicionalmente, el filtro require espacio en memoria adicional para guardar las equivalencias que es administrado desde C++ y transferido al método al invocarlo.

3.3.5. Contabilización de píxeles en regiones

Sobre las regiones encontradas, se contabilizan los píxeles correspondientes a cada una y se ordenan de mayor a menor.

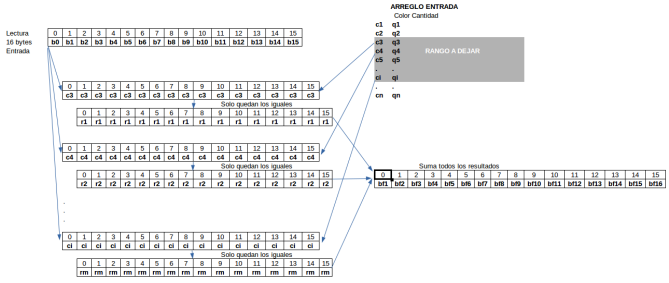
Se producen lecturas de 16 bytes y se va recorriendo byte por byte actualizando una estructura en memoria que lleva cuentas del color y cantidad.

Finalmente, el proceso realiza un selection sort en lenguaje ensamblador para devolver los resultados ordenados.

3.3.6. Filtrado de regiones

Este proceso toma un arreglo ordenado de mayor a menor con la información de color y cantidad de cada una de las regiones de la imagen de entrada y un rango de píxeles. El resultado es la misma imagen de entrada sin las regiones cuyo tamaño este por fuera del rango especificado.

Se logra un procesamiento en paralelo leyendo 16 bytes e iterando el arreglo para filtrar los mismos 16 bytes por cada color y luego, componerlos, como se muestra a continuacin:



Filtrado de regiones

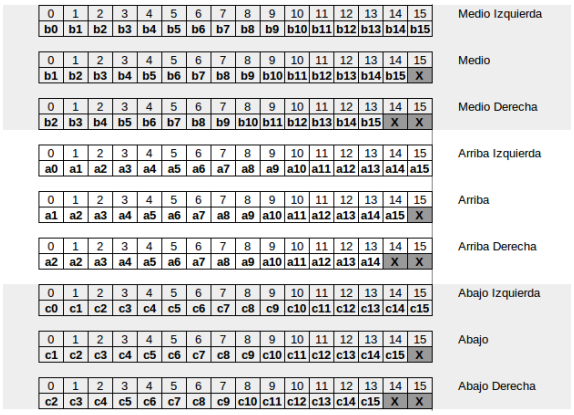
3.3.7. Detección de bordes

Este filtro recibe una imagen en escalas de grises o en blanco y negro y detecta el contorno de las imágenes.

Utiliza la técnica de detección de bordes Sobel que opera con una matriz de 3x3.

Se arman tres registros XMM con los valores correspondientes a 16 píxeles en la posición de memoria actual, los que estan encima y debajo en la imagen.

Estos registros, son copiados y desplazados una y dos posiciones para que finalmente queden 9 registros alineados de la siguiente manera:



Acomodamiento de Píxeles para operar en el filtro Sobel

Así se pueden procesar todos los píxeles en paralelo y sólo los resultados de 14 píxeles van a ser válidos por lectura.

Para operarlos fue necesario convertirlos a WORD y punto flotante para realizar las cuentas indicadas en la sección anterior.

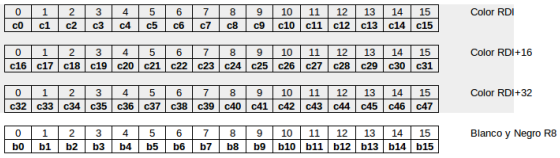
Finalmente, se empaquetan obteniendo el resultado.

3.3.8. Marcar bordes en imagen

Toma la imagen del video y le aplica los bordes marcados en una imagen blanco y negro.

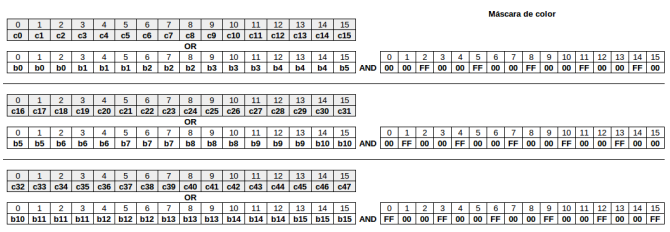
Vale comentar que este filtro maneja imágenes de dos dimesiones distintas: una imagen color que contiene tres bytes por pixel y una imagen blanco y negro que contiene tan sólo un byte por pixel.

Por lo tanto, para operar ambas imagen, se hacen tres lecturas de 16 bytes de la imagen color por cada una en blanco y negro.



Lectura de bytes en marcar bordes

Para operar se precisa transformar la imagen blanco y negro a color. Para esto todos los bytes de la imagen son repetidos tres veces, ocupando tres registros, como se puede observar:



Proceso de marcar bordes en imagen

3.4. Código en C++

El trabajo brinda una plataforma expansible, en la cual se pueden programar procesos y visualizar el comportamiento de los filtros paso por paso.

El diagrama de clases es el siguiente (incluyendo las implementaciones en ensamblador).

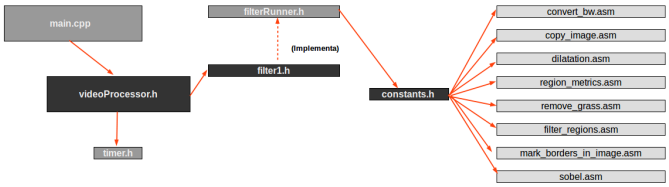


Diagrama de clases C++

Se observa que se puede extender la clase **FilterRunner** para agregar más secuencias de filtros a investigar.

4. Experimentación

En la presente sección se realizan pruebas con el *Proceso* a fin de cuantificar la efectividad y los tiempos de ejecución. Esto permite tener una medida de que tan bueno es el procesamiento indicado.

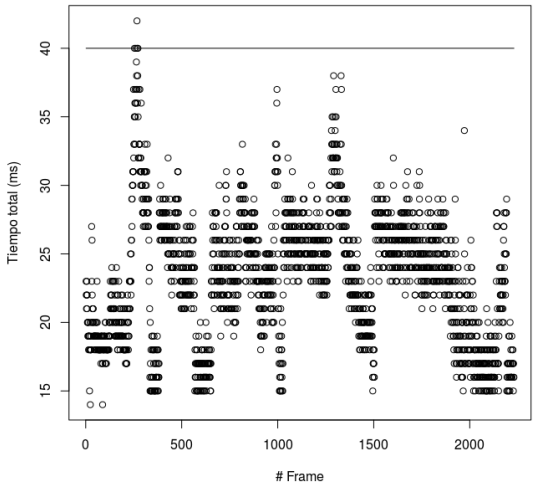
Las pruebas fueron llevadas a cabo sobre un fragmento de un partido de Primera División de la transmisión oficial Fútbol para Todos de Argentina. El video transcurre a 25 FPS y esta compuesto por 2230 frames.

Se ha experimentado con otros videos: fragmentos históricos, liga española, otros partidos de fútbol argentino, pero queda fuera del alcance del presente trabajo el análisis cualitativo y cuantitativo de los mismos.

4.1. Tiempos de ejecución

El valor absoluto de los tiempos de ejecución varía según la arquitectura de la máquina sobre la que se está ejecutando.

Este análisis tan sólo propone evaluar si el tiempo que lleva procesar todos los filtros cabe en el tiempo existente entre los frames.



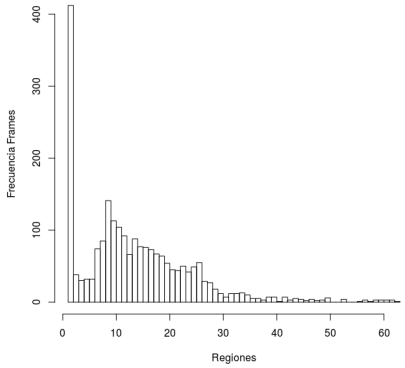
Tiempos absolutos del Proceso por Frame

Como se puede observar en el gráfico, el procesamiento en algunos pocos frames superan el tiempo disponible (aproximadamente 0,22 %).

4.2. Eliminación de frames no deseados

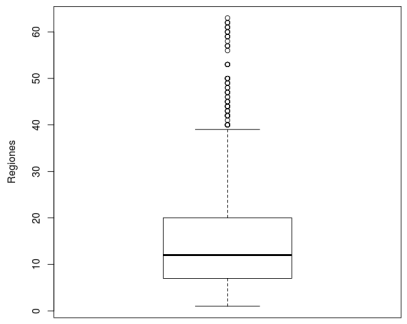
En esta sección, se propone evaluar la relación existente entre la cantidad de regiones y los frames que se desean eliminar debido a que se tratan close-up, animaciones, tomas de la tribuna, etc.

Se puede observar la cantidad de regiones que quedan en un frame al aplicarle la *Etiquetación de regiones* del *Proceso*. El gráfico a continuación muestra la cantidad de regiones por frame en el video experimental:



Cantidad de Regiones por Frame

Existen muchos frames de pocas regiones y, llamativamente, algunos pocos que tienen cantidades muy altas de regiones. Esos valores corresponden a outliers:

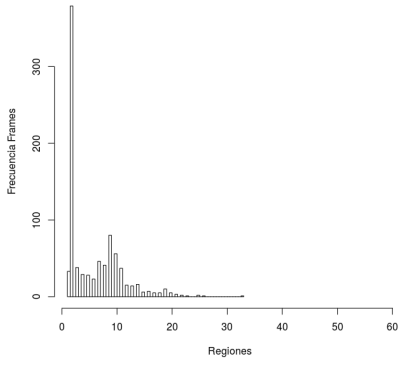


Outliers en cantidad de Regiones por Frame

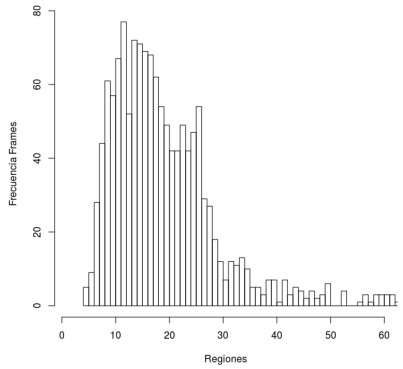
La mayor parte de estos outliers son en imágenes donde hay "papelitos" en el cesped. Cada papel, no es filtrado, sino que es tomado como una región a parte. Esto puede traer problemas y transformarse en un caso especial al momento de buscar los jugadores.

Como se explicó en la introducción, algunos frames no poseen jugadores para detectar. La determinación de si una imagen se quiere analizar o no, se realizó en forma manual, analizando el video e indicando para cada frame si hay jugadores o no para encontrar.

Si los datos sobre la cantidad de regiones se combinan con esta nueva información, se separan en dos grupos: cantidad de regiones en frames que tienen jugadores para detectar y las que no, obteniendo los siguientes:



Cantidad de Regiones en Frames no analizables



Cantidad de Regiones en Frames analizables

Se puede observar que en el caso de imágenes no analizables, hay una concentración de frames que tienen pocas regiones. De hecho, se ve que con más de 33 regiones, de seguro se trata de un frame analizable.

Por otro lado, en el gráfico de frames analizables, se ve que no hay jugadores para detectar en aquellos con menos de 5 regiones, por lo cual directamente, se pueden excluir del análisis, no buscar jugadores, y mostrar el frame original.

En el presente trabajo se experimentó configurando el *Proceso* en 7 regiones como límite inferior. Eso significa que sólo los frames con 7 o más regiones, van a ser seguir procesandose a fin de obtener los contornos de los jugadores y que, en caso contrario, se va a mostrar el frame original.

Con este valor, el error de exclusión de una imagen que debería ser analizada es 0,006 % (14 frames) y se estarían analizando 353 frames (15 %) que debieron ser excluídos.

Se puede realizar análisis en más videos y sobre distintos valores a fin de encontrar un valor que minimize el error. Incluso se podría mejorar investigando nuevos filtros que quiten regiones indeseables (por ejemplo, líneas) antes de la etiquetación o hacer algún procesamiento especial que achique el 15 % que se dejó pasar en filtros posteriores. Algunas ideas al respecto se verán en la sección de Propuestas para trabajos futuros.

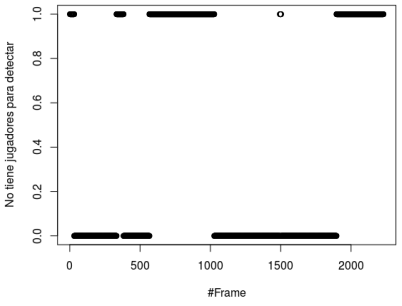
4.3. Detección de jugadores a partir del tamaño de la región

Las regiones tienen diversos tamaños. Se busca una aproximación que permita filtrar las regiones que no son jugadores en base a su tamaño.

Esto presenta varios inconvenientes. Por ejemplo, si varios jugadores se encontraran cerca, podrían quedar asociados a una única región de tamaño más grande al esperado para un jugador.

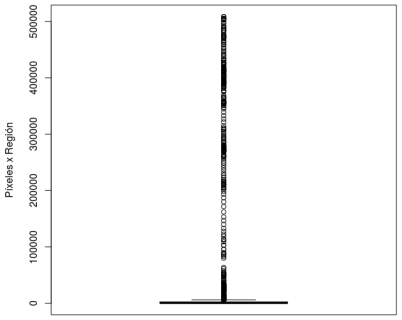
Otro problema es la perspectiva, los jugadores al acercarse hacia la cámara tendrían regiones más grandes que cuando se esten alejados.

Se propone a continuación un análisis estadístico sobre las regiones de los frames en los que hay jugadores. Para esto, primero, se localizaron rangos de frames en los que debe encontrar los jugadores.

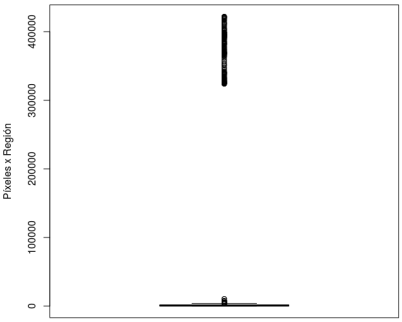


Frames donde se puede y donde no se puede detectar jugadores

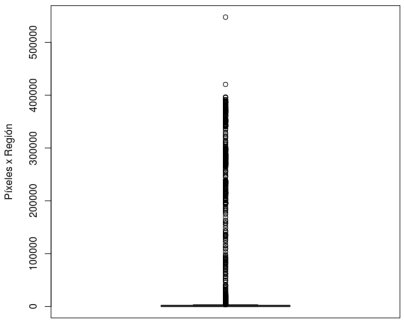
Se observa que el video posee cuatro rangos de frames bien definidos que son pasibles de la detección de jugadores.



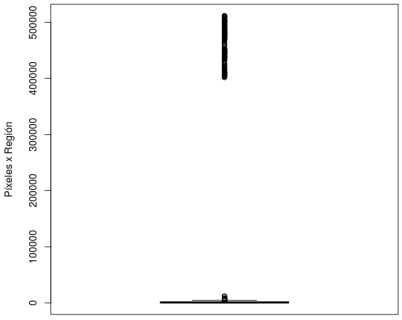
Píxeles por regiones en rango de frame 32 a 332



Píxeles por regiones en rango de frame 383 a 566



Píxeles por regiones en rango de frame 1028 a 1494



Píxeles por regiones en rango de frame 1503 a 1896

Frames: 32 al 332	
Min.	1
1st Qu.	183
Median	407.5
Mean	13346.3
3rd Qu.	2542
Max.	508639

Frames 383 al 566	
Min.	49
1st Qu.	182
Median	797.5
Mean	21435.9
3rd Qu.	1677.2
Max.	422172

Frames 1028 al 1494	
Min.	1
1st Qu.	188
Median	764
Mean	12695
3rd Qu.	1419
Max.	548029

Frames 1503 al 1896	
Min.	6
1st Qu.	146
Median	820
Mean	30020
3rd Qu.	2022
Max.	511641

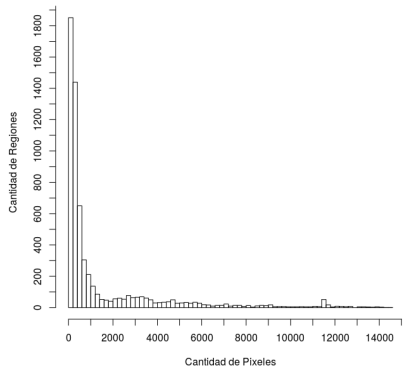
Estadísticas de tamaño de las regiones por segmento

Si bien difieren los valores estadísticos, el comportamiento es similar.

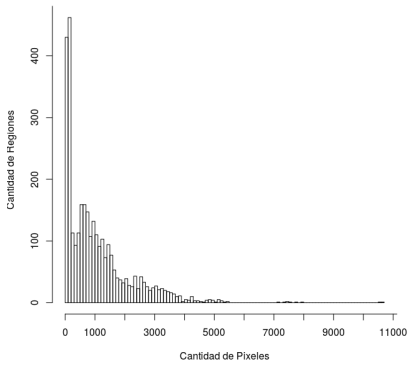
Se puede intuir que las regiones grandes corresponden a la tribuna, por eso son pocas y outliers. Por lo tanto, todo jugador se encontraría en regiones más pequeñas , por ejemplo, las inferiores al promedio (mean).

También, es claro que la regiones cercanas al mínimo no serían jugadores.

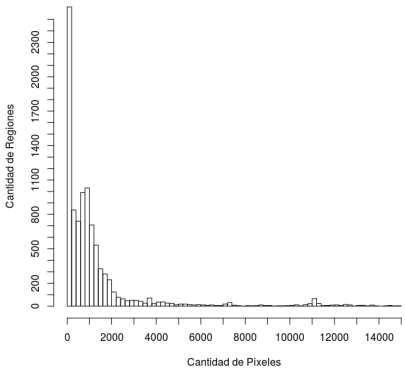
Se observa en detalle la distribución de las regiones por tamaño acotado a 15.000 píxeles:



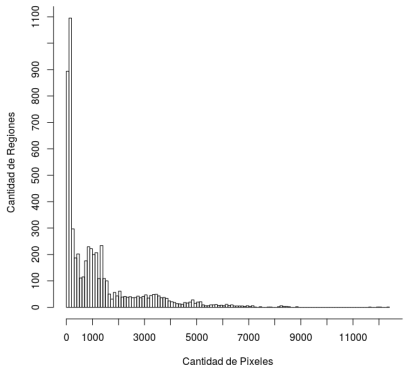
Píxeles por regiones en rango de frame 32 a 332



Píxeles por regiones en rango de frame 383 a 566



Píxeles por regiones en rango de frame 1028 a 1494



Píxeles por regiones en rango de frame 1503 a 1896

Claramente, hay un exceso de regiones de pocos píxeles que podrían ser descartadas.

Se utilizó el rango de píxeles 200-13000. Se incluyen los resultados cuantitativos de la experimentación a continuación:

Lote	Rango Frames	Aceptables	Totales	% Efectividad
1	32 a 332	56	299	18.73%
2	383 a 566	177	184	96.20%
3	1028 a 1494	314	467	67.24%
4	1503 a 1896	298	394	75.63%

Resultados cualitativos por rango de frames

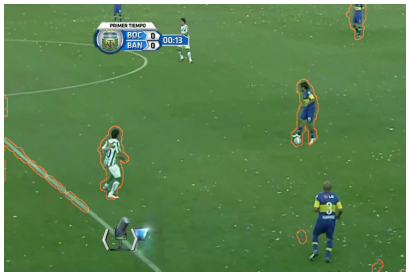
Vale aclarar que se toma como aceptable un frame que detecte los jugadores con la posibilidad de no detectar aquellos que se encuentren cerca de la tribuna, dado que son filtrados junto a ella. Por ejemplo, el arquero. Encontrar una forma de detectar estos jugadores también podría ser tema de un próximo trabajo.

Para entender un poco más estos resultados se presenta un análisis cualitativos sobre los lotes.

El primer lote, que posee un muy mal desempeño, tiene en la mayoría de los frames animaciones del mismo tamaño que los jugadores:

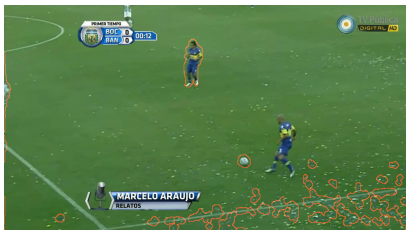
Frame mal detectado, molesta los papelitos

Por otro lado, algunos de los frames muestran las líneas del campo de juego, que también son tomados como jugadores:



Frame mal detectado, la linea confunde al jugador

Y por último, los papelitos no fueron eliminados al eliminar el cesped:



Frame mal detectado, molestan los papelitos

El segundo lote, el desempeño es muy superior. Si se observa las imágenes resultantes tienen los jugadores claramente marcados:

Aceptable. Los jugadores que no se marcan, se confunden con la hinchada.

5. Conclusiones

La variedad de videos, la cantidad de objetos que aparecen, la gama de los colores, el poco tiempo entre los frames hace que la detección de jugadores en una trasmisión de fútbol sea un problema desafiante del procesamiento de imágenes en tiempo real.

Obtener un proceso infalible para el 100 % de los frames y de los videos, es realmente muy complejo.

Los procesos implementados tienen una efectividad oscilante que depende del video analizado. El procesamiento especificado no es suficiente para que los jugadores puedan ser filtrados únicamente en función al tamaño de la región.

Sin embargo, cabe destacar que el filtrado por cantidad de regiones en imágenes que no corresponden al campo de juego obtuvo un buen resultado.

Si bien los resultados no son los deseados para el total de los frames, queda por sentado una plataforma para futuros trabajos de análisis de videos de fútbol y métricas que pueden de forma comparativa ser utilizadas para seleccionar los mejores.

Se proponen a continuación líneas futuras de trabajo.

6. Propuesta próximo trabajo

- Implementación de otros ordenes de aplicación de filtros, más procesos para compararlos y mejorar la técnica.
- Implementar el filtro de eliminación de lineas con la tranformación de Hough, erosión, apertura, cerrado, otros.
- Agrupar el mejor proceso elaborado dentro de un único programa ensamblador para acelarar los tiempos de procesamiento.
- Detectar jugadores en los bordes.
- Detección automática de parámetros, ejemplo offset de color de pasto, rango de píxeles en los que esten los jugadores, etc.
- Refinamiento varios, por ejemplo, deteción exacta de los bordes combinando la imagen original sin dilatar y sin eliminar del pasto con detección de bordes restringido a la ubicación detectada del jugador.

7. Apéndice

7.1. Ejecución del programa

La ejecución se hace mediante el binario: `./fplayers` . Para obtener la lista de parámetros disponibles: `./fplayers -h` y mostrará:

```
TP Final Organización del Computador II (A. Furfaro)
Soccer Players Finder by Leticia L. Rodrguez
Run: ./fplayers -f <filename> [-f <filename>] [-v] [-d] [-i]
[-t] [-s <step>] [-h] [-r <process>] [-g <offset>] [-p <frame_from>
<frame_to>]
Command line parameters:
-f <filename> Specifies video filename to process
-v Verbose. Prints realtime information
-d Dual. Shows two windows of the video.
-i Add detail information for each filter
-p Start the video in pause
-s <step> Start Step (default 1)
-r <process> Filter rule (process), default 1
-g <offset> Color number offset to adjust in delete grass, default
0
-c <fr_from> <fr_to> Cut. Plays video only from fr_from to fr_to
-t Test process
-q Quick. Fast. Doesn't show video, no key waiting.
-h Show this help
EOM
```

Ejemplos:

`./fplayers -f boca_banfield.avi -g 31`
Reproduce el video boca_banfield.avi en 2 ventanas: primera video original y la 2da. el mismo video pero procesado por el programa de detección de jugadores en el paso 1: eliminación de pasto.

`./fplayers -f boca_banfield.avi -v -i -g 31`
Igual al anterior, pero imprime información sobre el video en ejecución, que filtros es procesando, parámetros e información de logueo.

`./fplayers -f boca_banfield.avi -s 8 -r 1 -g 31`
Reproduce el video original y el de detección de jugadores en el paso 8 indicado por (-s 8) y utilizando el proceso 1 (-r 1).

`./fplayers -f boca_banfield.avi -s 8 -r 1 -g 31 -v -i`
Idem anterior pero imprimiendo información de logueo.

`./fplayers -f boca_banfield.avi -p -g 31`
Inicia el video en pausa.

`./fplayers -f boca_banfield.avi -c 10 50 -g 31`
Reproduce el video original y el de detección de jugadores, sólo los frames desde el número 10 al número 50.

`./fplayers -f boca_banfield.avi -t -g 31`
Reproduce el video original y el de detección de jugadores en modo

testeo.

La ventana de detección de jugadores es interactiva. Las teclas válidas son:

Modo Normal:

- ESC Sale de la pantalla
- 0-9 Setea el paso número
- s Toma una captura de pantalla blanco y negro, guarda en screenshot.bmp
- c Toma una captura de pantalla color, guarda en screenshot-color.bmp
- BarraEspaciadora Pausea o reanuda el video

Modo Testeo: Los frames pasan al presionar una tecla y se guarda un archivo test.log con la tecla que se presionó en cada frame.

8. Bibliografia

M. M. Naushad Ali, M. Abdullah-Al-Wadud and Seok-Lyong Lee, **An Efficient Algorithm for Detection of Soccer Ball and Players**

Stanford University Course EE368/CS232, **Digital Image Processing.**
http://web.stanford.edu/class/ee368handouts.html

Universidad Tecnológica Nacional, **Curso Datascientist con R**
http://elearning.sceu.frba.utn.edu.ar/e-learning/cursos-a-distancia/Informatica-y-Sistemas/Data-Scientist-con-R/temario.html

Rafael C. Gonzalez, Richard E. Woods, **Digital Image Processing**, 2da. Ed, Erosion-Dilatation pp 523-527

Junqing Yu, Yang Tang, Zhifang Wang, Lejiang Shi, **Playfield and Ball Detection in Soccer Video**, Advances in Visual Computing Volume 4842 of the series Lecture Notes in Computer Science pp 387-396

Yu Huang, Joan Llach, Sitaram Bhagavathy, **Players and Ball Detection in Soccer Videos Based on Color Segmentation and Shape Analysis**, Multimedia Content Analysis and Mining Volume 4577 of the series Lecture Notes in Computer Science pp 416-425

Samuel F. de Sousa Júnior and Arnaldo de A. Araújo, David Menotti, **An Overview of Automatic Event Detection in Soccer Matches**

Wikipedia, **Sobel Operator**,
https://en.wikipedia.org/wiki/Sobel_operator

Wikipedia, **Connected Component Labeling**,
https://en.wikipedia.org/wiki/Connected-component_labeling

Faculty of Science, Utrecht University, Mathematical morphology pp 127
http://www.cs.uu.nl/docs/vakken/ibv/reader/chapter6.pdf

Intel 64 and IA-32 Architectures Software Developer Manuals
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html