Proyecto Fin de Máster

Máster Universitario en Ingeniería en Electrónica, Robótica y Automática

Implementación software y mejora de un filtro de flujo óptico para cámara por eventos

Autor: Carlos Martín Cañal

Tutor: Hipólito Guzman Miranda

Cotutor: Alejandro Linares Barranco

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Proyecto Fin de Máster
Máster Universitario en Ingeniería en Electrónica, Robótica y Automática

Implementación software y mejora de un filtro de flujo óptico para cámara por eventos

Autor:
Carlos Martín Cañal

Tutor:
Hipólito Guzmán Miranda
Profesor Contratado Doctor
Cotutor:
Alejandro Linares Barranco
Profesor Titular de Universidad

Dpto. de Tecnología Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Proyecto Fin de Máster: Implementación software y mejora de un filtro de flujo óptico para cámara por eventos

Autor: Carlos Martín Cañal

Tutor: Hipólito Guzman Miranda

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal

A mi familia
A mis profesores

Agradecimientos

A mis profesores por su inestimable ayuda, y comprensión y a mi familia, que, gracias a su apoyo y cariño, he podido culminar con éxito este trabajo.

Resumen

En el presente trabajo se ha planteado un novedoso algoritmo, basado en la tecnología de Adress-Event, que extrae el flujo óptico desde eventos. Para realizar dicho proceso, se ha realizado un estudio de los distintos algoritmos clásicos de flujo ópticos, como el Lucas-Kanade, block-matching, Horn-Shuncky Elementary Motion Detector. Este último es el algoritmo que se ha estudiado con más profundidad al ser un candidato idóneo para ser implementado en hardware. Se ha desarrollado una versión mejorada de este filtro, extendiendolo a todas las posibles combinaciones de eventos. Para finalizar, este trabajo se ha culminado con la implementación de una versión en software de dicho filtro usando el *framework* de Java, jAER.

Una vez que se ha llevado a cabo la implementación, se ha realizado una comparación con los diversos tipos de filtros que ya han sido implementados en jAER para extraer métricas de velocidad de procesamiento y métricas de linealidad y, de esta manera, poder comparar entre los distintos algoritmos de flujo ópticos considerados.

Abstract

In this work, it has been studied the optic flow in adress-event *framework*. As principal algorithm, the "Elementary Motion Detector" has been studied. This algorithm has been extended to accept all possible combination of events. The final algorithm has been implemented in Java through jAER *frame*work, and it has been obtained the results for distinct type of optic flow filters.

Contenido

Índice de Tablas

Índice de Figuras

1 OBJETIVOS DE ESTE TRABAJO

El flujo óptico es un aspecto muy importante para disciplinas como la Robótica, donde ayuda a la navegación y la estimación de trayectoria, junto con otros sensores. Con la llegada de la cámara de eventos, se ha mejorado la adquisición de vídeo, ya que sólo se transmiten las partes de la escena que cambian en el tiempo. Con este principio, junto al algoritmo propuesto en este trabajo, se ha intentado mejorar la forma en la que se obtiene dicho flujo óptico, proponiendo un novedoso método basado en el sistema visual de las moscas Drosophila.

Este trabajo se ha culminado con la implementación en el framework jAER de dicho algoritmo y comparandolo con otros algoritmos de flujo óptico.

Para la realización de este TFM se han desarrollado los siguientes aspectos:

- Estudio de los distintos algoritmos de flujo óptico.
- Estudiar el algoritmo de "Elementary Motion Detector"(EMD) por ser un posible candidato a la implementación.
- Mejora del filtro EMD extendiéndolo a todas las posibles combinaciones de eventos de distintas polaridades.
- Implementación del filtro mejorado en jAER.
- Comparación, mediante algunas métricas, de los distintos algoritmos de flujo óptico.

2 DESCRIPCIÓN DE LOS EVENTOS

2.1 Introducción

Actualmente, las cámaras están basadas en *frame*s(fotogramas), es decir, que muestrean en intervalos de tiempo regulares toda la imagen. Este método ha sido el predominante hasta ahora, debido a que se ha investigado durante muchos años y la tecnología está muy madura. Existen distintas tecnologías, como la CCD o la CMOS [1] para los sensores de imágenes.
En relación con el ancho de banda, dichos dispositivos precisaran de un elevado rango de este para que la imagen pueda ser transmitida en tiempo real. Esto es importante puesto que, si se trata de una imagen estática, o una gran parte de ella lo es, dará como resultado una importante cantidad de información redundante e innecesaria con la consecuente saturación del canal.
Algo parecido sucede con los sistemas de procesamiento de imagen, donde la situación se complica. En ellos la escena o parte de ella no cambia y requiere un gran coste computacional superfluo. Por todo ello y en vista de la necesidad de solucionar dichas contingencias se ha propuesto un sensor que resuelve estos problemas basado en las retinas biológicas. [1]
En el sensor descrito en [1], los píxeles responden a cambios de intensidad lumínica. La salida del sensor es un flujo asíncrono de píxeles que codifican los cambios en la reflectancia en la escena y también su rectificación en eventos positivos y negativos.
Por lo tanto, para la correcta transmisión y posterior procesamiento de eventos se precisa de un *filtro de eventos*. En primer lugar, se ha de diseñar un sistema que además de representarlos, los defina y temporalice.
A continuación, se explicará dicha representación y se realizará una comparativa con el equivalente basado en *frame*.

2.2 Representación por eventos

Por la peculiaridad de las cámaras de eventos, su comunicación es **asíncrona,** es decir que solo se transmite un evento si ha ocurrido un cambio de luminosidad en ese píxel. Sinembargo, el sistema de barrido propio de las cámaras basadas en *frame*, en la que el sensor es leído entero y después es transmitido a través de algún canal de comunicación, no se puede usar, ya que este sistema es del tipo **síncrono**. En él la imagen es escaneada, leyendo cada píxel en intervalos regulares y conocidos, para ser reproducida mediante señales de control que sincronicen cada píxel de la pantalla con el píxel correspondiente de la cámara.
La idea de la comunicación asíncrona es la siguiente: Cada píxel del sensor mide la intensidad lumínica en su entorno. Cuando ocurre un evento, que en el caso de las cámaras de eventos es cuando se produce un cambio de intensidad lumínica, entonces la circuitería del píxel se encarga de procesar dicho cambio. Si pasa de un determinado valor, se produce un evento, este es recogido por la circuitería de la cámara y procesado para ser transmitido hacía el exterior.

A continuación, se describe la circuitería del píxel y su funcionamiento mediante gráficas. El esquema que usa cada píxel del sensor se muestra en la Ilustración 1.

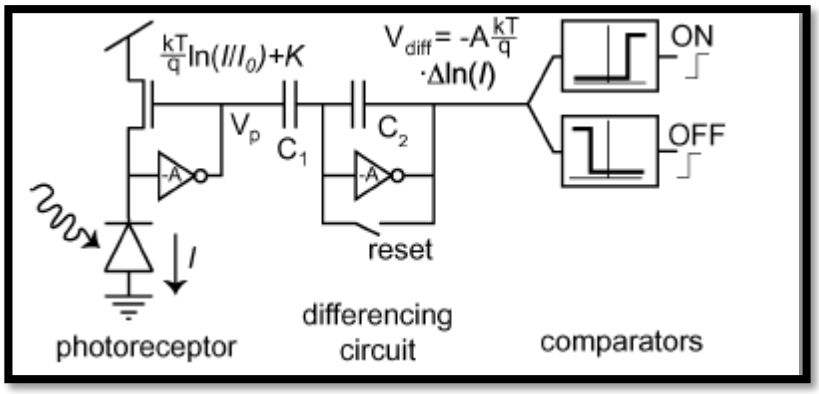

Ilustración 1: Esquema de cada píxel [1]

La primera parte del circuito es un amplificador y una fuente de corriente para polarizar el fotorreceptor en un punto óptimo de operación. El siguiente paso es un condensador que actúa como diferenciador, derivando la señal y obteniendo así la velocidad de cambio de la señal de luminosidad dada por la parte anterior del circuito. Esta señal pasa por un circuito integradorque integra esta señal y actúa como contador. Cuando se haya alcanzado un cierto umbral, se produce un evento, y se reinicia el integrador, volviendo al estado inicial. Dependiendo de la naturaleza del evento, tendremos eventos de tipo ON, si el crecimiento de la luminosidad es positivo y, por el contrario, OFF si el crecimiento es negativo. Para ejemplificar este principio de funcionamiento la ilustración 2 muestra dos gráficas de voltaje-tiempo que describen la dinámica del píxel.

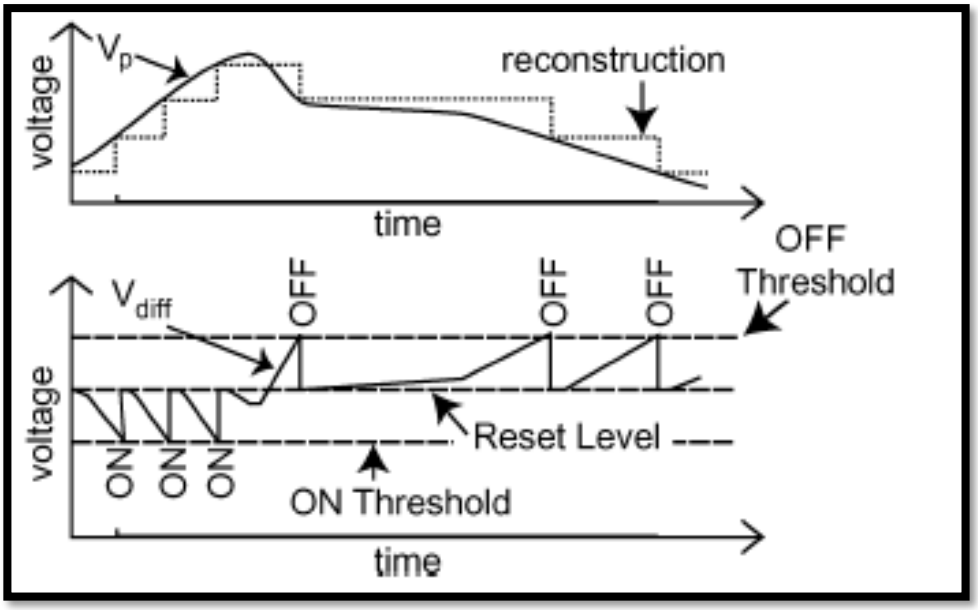

Ilustración 2: Gráficas de funcionamiento de un píxel. [2]

En esta gráfica se observa el funcionamiento descrito en el anterior párrafo: cuando ocurre un cambio en la luminosidad el voltaje en el integrador va subiendo hasta que alcanza un cierto umbral, y lo mismo ocurre en el caso de que el cambio se produzca en la dirección contraria. También se ejemplifica que, dependiendo de la razón de cambio de la luminosidad, tendremos diferentes frecuencias de eventos, por lo que dependiendo de cómo se mueva un objeto en la escena podemos estimar el movimiento o la velocidad y algunas características de la imagen.

2.3 Interfaz Adress-Event y arquitectura de la cámara.

Después de que un evento se produzca, éste debe de ser adecuadamente recogido para que pueda ser trasmitido y posteriormente procesado. Cada píxelforma parte de una matriz y se comunica con los circuitos periféricos. Cada uno de ellos comunican su posición (x, y) y su polaridad (ON, OFF). Esta

comunicación se dice que está *arbitrada*, ya que cada píxel, donde se ha producido un evento, espera a que el bus esté libre para transmitir el evento. También la cámara dispone de un sistema que impide que pueda producirse otro evento en un píxel en el que previamente el arbitrador ha concedido servicio, de esta forma se evita el envío por duplicado. A continuación, se muestra un esquema simplificado de la arquitectura de la cámara:

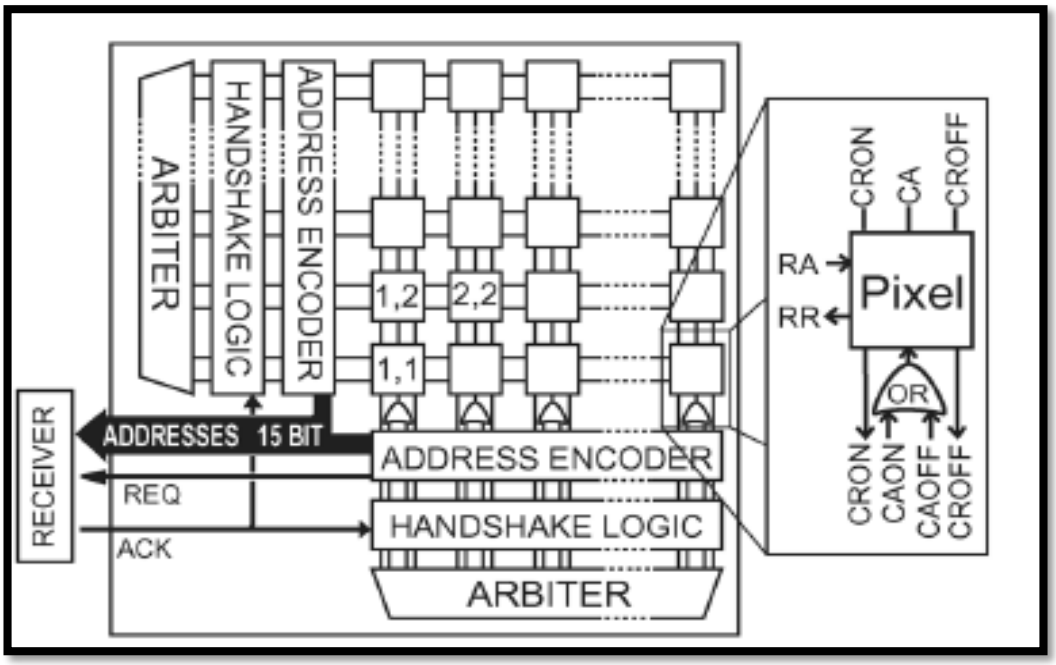

Ilustración 3: Arquitectura de la cámara DVS128[1]

Como se puede observar en la figura anterior, cada píxel forma una unidad mediante la cual se comunica con el exteriorcon unas series de señales que se describen a continuación:
A cada píxel le corresponden unas señales de reconocimiento (ACK) de columnas y filas que son CAON para los eventos de polaridad ON y CAOFF para los eventos de polaridad de tipo OFF. Para la elección de la fila tenemos la señal RA. De hecho, las señales CA como la RA son compartidas debido a que un píxel no puede tener 2 polarizaciones distintas al mismo tiempo [1]. Cuando se ha producido un evento de tipo ON, y además se activan las líneas RA y CAON al mismo tiempo, se produce la activación de la línea CRON, que comunica al arbitrador que se ha producido un evento de tipo ON en dicho píxel. Lo mismo ocurre con un evento de tipo OFF.

2.4 Interfaz con dispositivos externos

El sensor de visión puede ser conectado a otros dispositivos mediante el bus AERque usa el mismo protocolo de transmisión con palabra paralela, como por ejemplo CAVIAR [1] multichip AER, que se puede conectar directamente, u otro sistema que tenga el protocolo AER.
Para poder diseñar filtros para procesar los eventos en bruto se desarrolló un sistema de comunicación que mediante USB transmite un flujo de eventos a un PC. El problema es que los eventos son de naturaleza asíncrona, así que para procesar en tiempo real los eventos en el PC se deben usar algunas técnicas de acumulación de eventos para que sean procesados de manera secuencial en un sólo hilo de ejecución. La arquitectura del sistema se muestra en lasiguiente página:

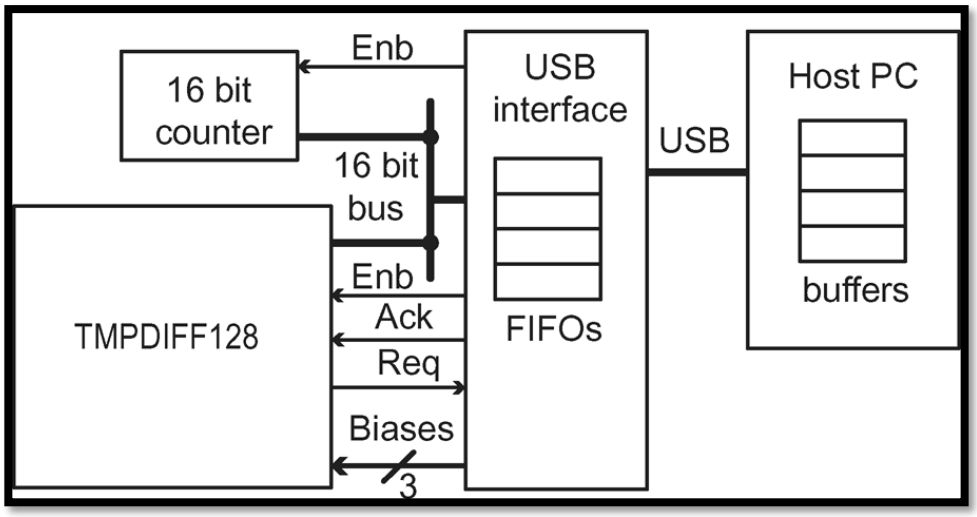

Ilustración 4:Arquitectura de comunicación con la PC [1]

El sensor de visión envía los eventos, y un contador de 16 bits corriendo a 100KHz se encarga de capturar el *timestamp*de cada evento. Como se ve en la figura, el contador comparte el bus de 16 bit con el sensor, y éstese comunica con la interfaz del USB que es básicamente una FIFO y funciona como buffer. Posteriormente son enviados por vía USB hacia el PC, donde será adecuadamente separado el *timestamp* del evento, de modo que esta información servirá como entrada a otro hilo de ejecución para alguna aplicación específica de procesamiento por eventos.
Para tal tarea se han desarrollado una serie de clases en Java que recogen los eventos recibidos del sensor y los procesa para alguna aplicación en concreto. Dicha herramienta se llama jAER [2], que además tiene un monitor de eventos y la posibilidad de crear salidas "a medida" para cada aplicación. Contiene una gran cantidad de ejemplos. Esta herramienta será desarrollada más detenidamente en el capítulo3: **Software jAER**, donde se hará una breve introducción a la herramienta y se procederá a describir el filtro realizado.

2.5 Datos principales del sensor de visión

En este apartado se describirán las características más importantes que tienen el sensor de visión.
Esta cámara tiene el siguiente aspecto:

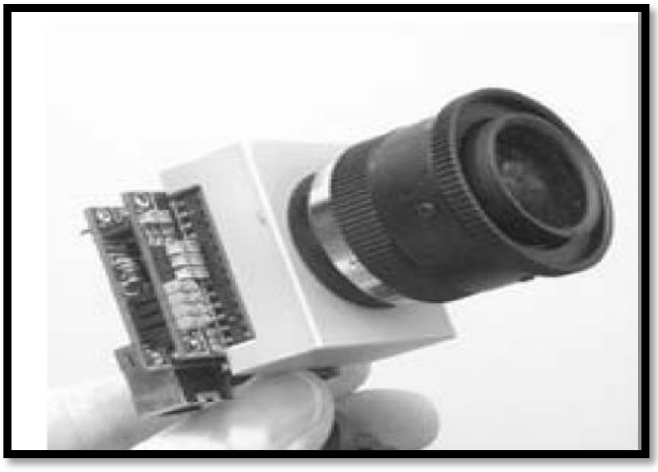

Ilustración 5: DVS128[1]

Algunas de las características más importantes del sensor son:

Características de la cámara DVS128	
Funcionamiento	Contraste temporal asíncrono
Interface	15 bits de palabra AER
Software	jAER y Caer
Dimensiones	H50 xW50xD25
Potencia	30-60mA @5V
Complejidad del píxel	26 transistores 3 condensadores
Rango dinámico	120dB 2lux a 100klux
Ancho de banda	1M Event/s
Voltaje	3.3V

2.6 Descripción del hardware del receptor de evento del sensor

Para realizar el procesamiento de los eventos en la aplicación final y comunicar con el exterior, se dispone de una última etapa, implementada mediante un CPLD o un microcontrolador, el cual se encarga de:

- Establecer la comunicación con la retina.
- Recoger un nuevo evento.
- Validar dicho evento mediante un bit de paridad.
- Guardar dicho evento en la pila FIFO.
- Generar el *timestamp* para dicho evento.
- Guarda el *timestamp* en la FIFO.
- En este último punto tenemos dos opciones:
    - Procesar el evento nativamente.
    - Enviar el evento completo mediante algún protocolo de comunicación hacía el exterior para ser procesado.

Aparte de estas funciones, el CPLD también puede realizar la carga de los parámetros del bias de la retina y comunicarlos al exterior.

La FIFO tiene la misión de servir como buffer para que no se produzcan errores de comunicación, ya que, al ser una comunicación de carácter asíncrono, puede que el receptor no esté preparado para un flujo muy rápido de eventos, por lo que el propio emisor debe ir almacenando todos los eventos que no consiga transmitir, para luego enviarlos cuando el bus esté inactivo.

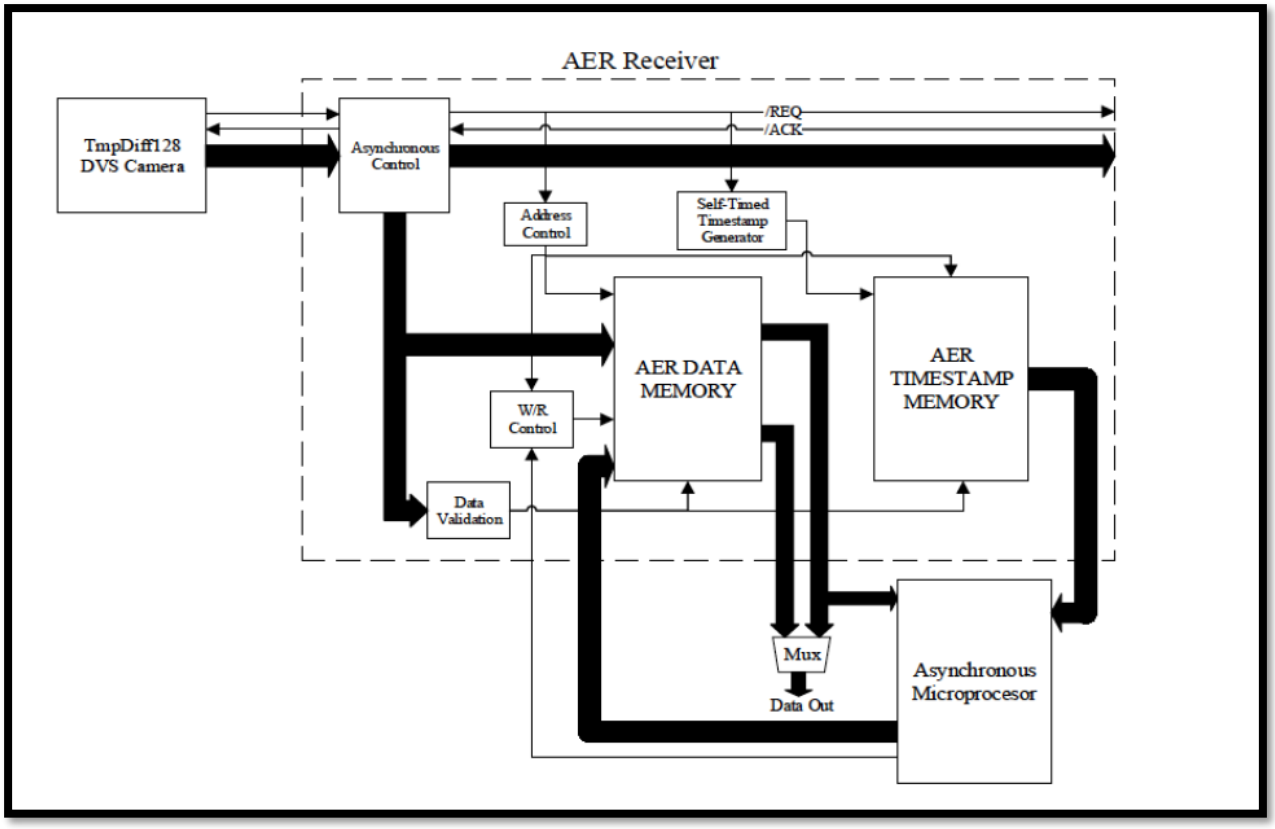

Ilustración 6: Diagrama simplificado del receptor de eventos. [23]

3 SOFTWARE JAER

3.1 Introducción

Se trata de un software libre de simulación, tratamiento y comunicación con cámaras de eventos [21]. Está escrito en Java. El proyecto nació en 2007 para dar soporte a los prototipos de las cámaras, visualización y tratamiento de eventos y sus diversos algoritmos.

3.2 Funcionamiento del programa

La precisión del *timestamp* de los eventos producidos por el sensor es de 1us. Son transmitidos vía USB al PC, donde el programa jAER se encarga de recibir el paquete con todos los eventos que se han producido en ese tiempo. Es entonces cuando el usuario se encarga de aplicar filtros o diversos tipos de visualizaciones para poder tratar y visualizar los eventos además de crear nuevas aplicaciones. Estas últimas permiten también guardar la secuencia de eventos y *timestamp* en disco en archivos de extensión aedat.

3.3 GUI de la aplicación

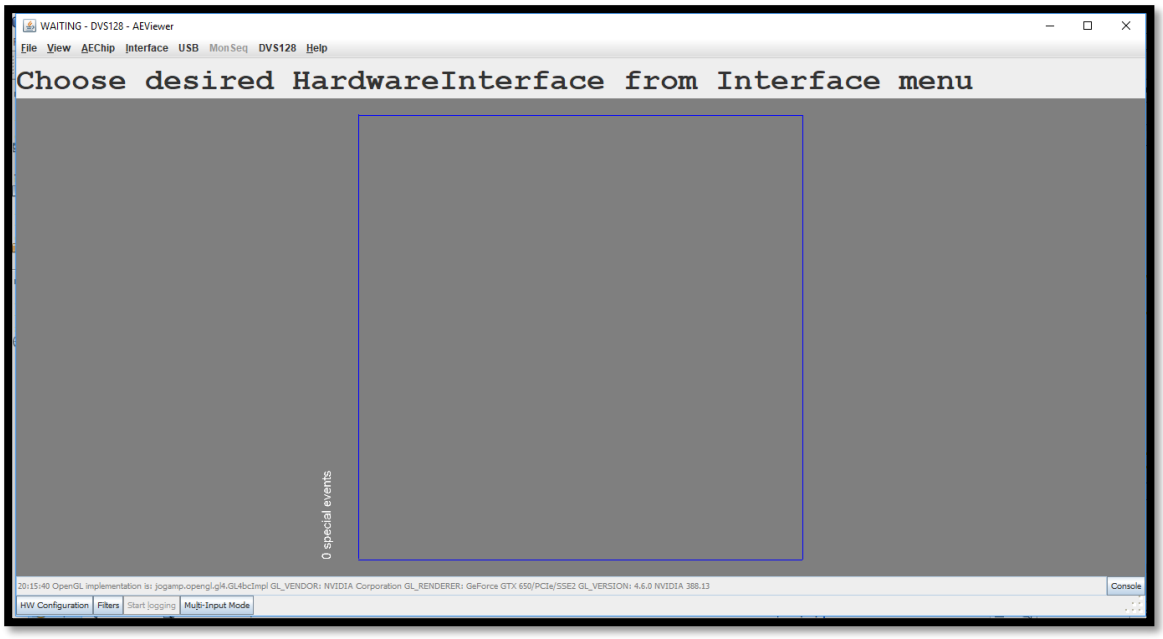

Ilustración 7: GUI de inicio del jAER

Para realizar la carga de algún archivo de eventos, jAER tiene una opción de abrir archivos de eventos, tal como se muestra en la siguiente figura:

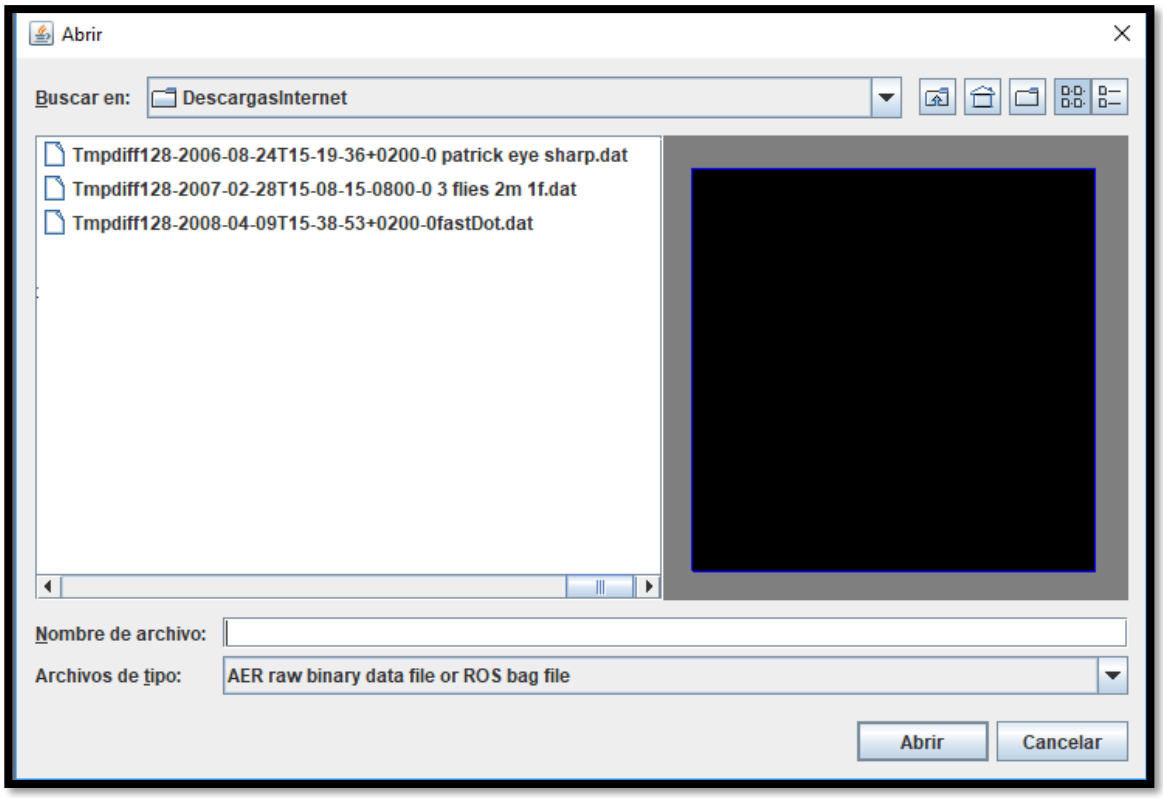

Ilustración 8: Ventana para cargar los archivos

Una captura del programa funcionando, donde se pueden observar diversas lecturas en el GUI referente a los eventos:

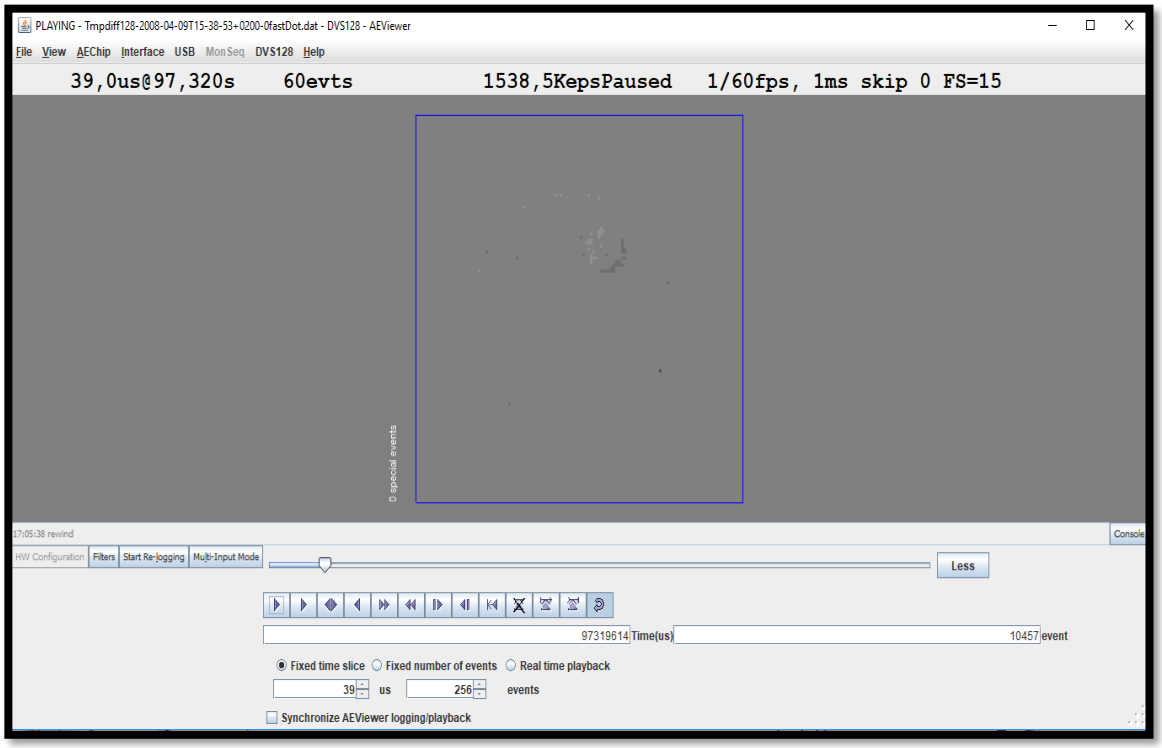

Ilustración 9: Visualización de los eventos.

Como se puede ver en la parte posteriorde la imagen,aparece unos datos referentes a los eventos:

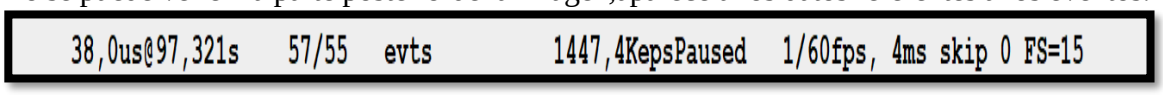

Ilustración 10: Barra de información

Ahora pasamos a describir cada una de las cifras aquí observadas:

Nombre	Explicación	Ejemplo
" Time slice"	Es el tiempo cubierto por el *frame* renderizado actual.	38,0us
" Tiempo absoluto"	Es el *timestamp* actual de la imagen.	97,321s
Eventos en bruto/filtrados	Número de eventos en brutos de cada *frame* y filtrados	57/55 evts
Tasa de eventos	Numero de eventos que se producen por segundo. Indica también si está pausado.	1447,4KepsPaused
Actual tasa de *frame*s /tasa deseada de *frame*s	Número de *frame*s actual y número de *frame*s deseados, definido por el usuario.	1/60fps
Retraso después del renderizado de un *frame*	Es el tiempo que está el procesador ocioso antes de que el bucle infinito del programa empiece de nuevo.	4ms
Escala de color	Renderizado de los eventos.	FS=15

3.4 Selección de filtros

Este programa puede cargar diversas clases en Java para el procesado de eventos. La ventana para cargar los diversos filtros es la siguiente:

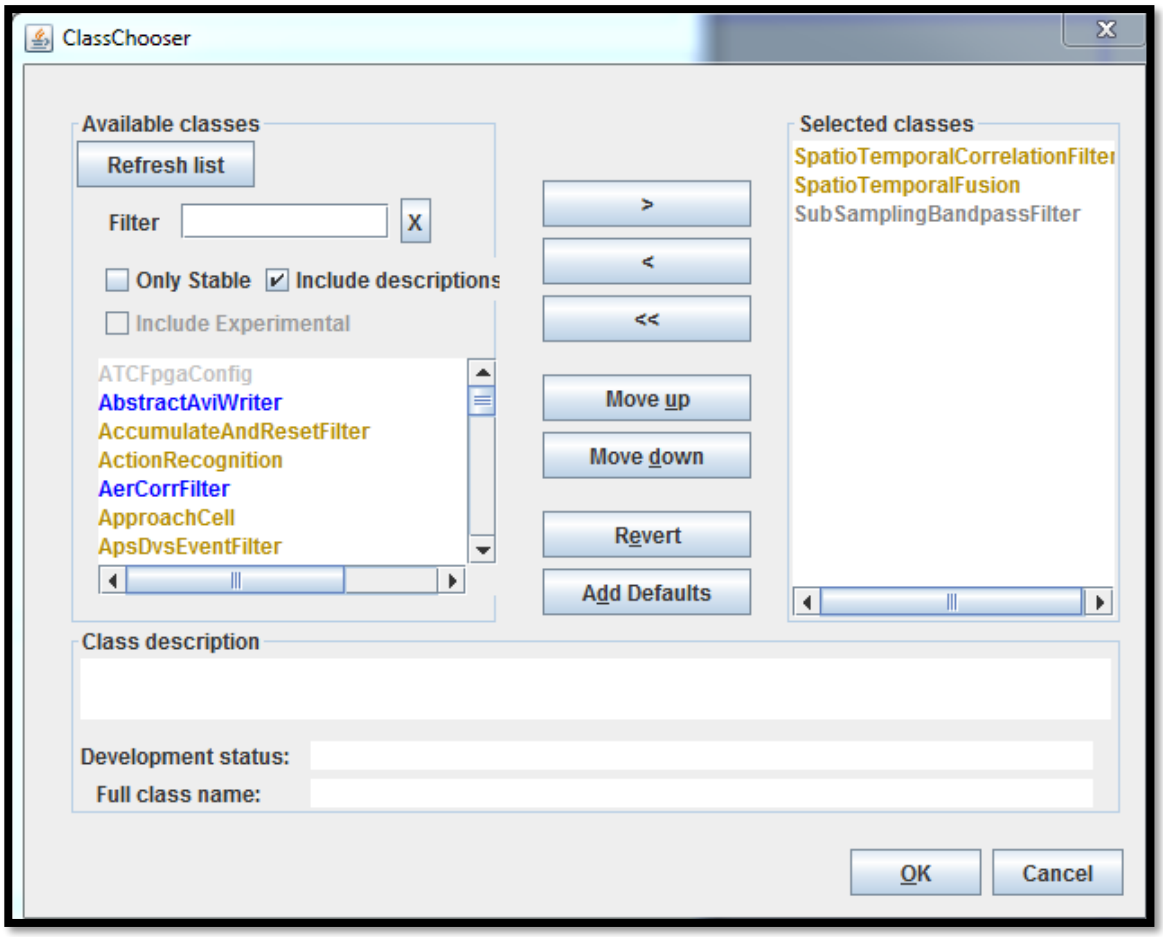

Ilustración 11: Ventana de selección de filtros.

Cada uno de estos filtros están situados en el proyecto como clases aisladas, que se puede modificar y escribir otros nuevos.

4 ALGORITMOS DE FLUJO ÓPTICO

4.1 Introducción

El flujo óptico es un patrón de movimiento aparente de un objeto, o esquinas dentro de una escena visual, resultado del movimiento relativo de los objetos de la escena y el observador. Se define como la velocidad de variación del patrón de brillo en una región de la imagen.
Este concepto fue introducido por el psicólogo James J. Gibson[22] ,durante los años 40, para describir el estímulo queacontece en el córtex visual del animal cuando sedesplaza a través del entorno, creando un mapa mental para llegar a su objetivo. Una representación la tenemos en la siguiente figura:

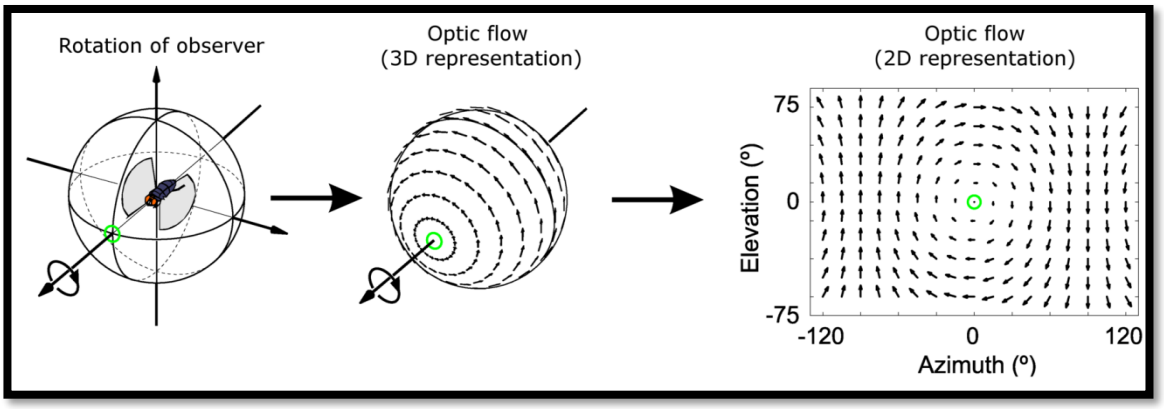

Ilustración 12: Flujo óptico [3].

4.1.1 Tipos de algoritmos

- *Correlación de fase:* Se trata de una técnica basada en la transformada de Fourier, que devuelve la frecuencia y la fase de la imagen. Como la traslación de un objeto en la imagen es un simple cambio de fase, puede ser extraído el movimiento facilmente.
- *Métodos de bloque*: Se basa en separar la imagen en múltiples bloques y encontrar dichos bloques en la imagen del instante siguiente, mediante una función de suma de error.
- *Métodos diferenciales*: Son los métodos más usados por su escasacomplejidad computacional. Se basa en el hecho de que la intensidad luminosa de un objeto o una escena no varían en el *frame* inmediatamente próximos.

4.1.2 Usos

Los algoritmos de flujos ópticos tienen múltiples usos. Uno de ellos, el más evidente, es servir a la Robótica y a los vehículos autónomos en la navegación y en Odometría. Otro uso está en el procesamiento de imagen, ya que algunos algoritmos de compresión de vídeo se basan en el flujo óptico para comprimirlos y eliminan la redundancia en la imagen. También es usado como herramienta para la visión estereoscópica.

4.2 Algoritmo de Lucas Kanade

Calcular los movimientos en una imagen es una actividad muy interesante para desarrollar tareas de tracking, cálculo de movimientos en sí y como base para la visión estereoscópica. El proceso de estimar el movimiento se define como la determinación de los vectores que describe la transformación de una imagen 2D en otra [3]. En otras palabras, estima la velocidad de las

estructuras de la imagen de un *frame* a otro en una secuencia temporal [3].

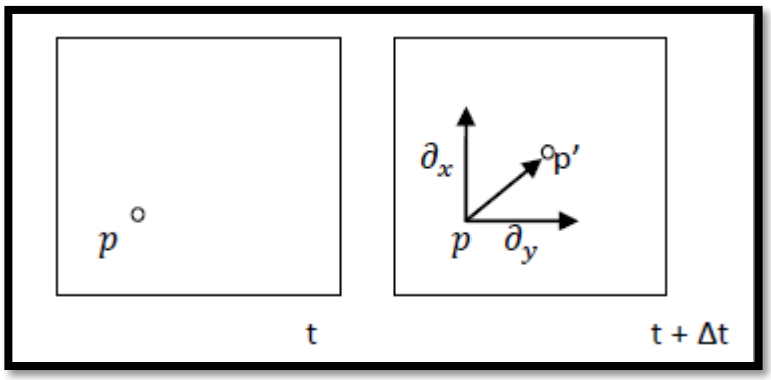

Ilustración 13: Concepto del cálculo del movimiento en una imagen [3].

En el primer *frame* se observa un píxel en (x, y), en el intervalo *t* de la secuencia de imágenes. En la segunda ese mismo píxel se ha desplazado un valor $(x + \partial x, y + \partial y)$.Esta transformación puede ser una traslación o una rotación.

4.3 Flujo óptico

El concepto de flujo óptico es un patrón de movimiento de objetos, esquinas o superficies en una escena visual causada por un movimiento relativo entre la cámara (o el observador) y los objetos que la componen. A continuación, se demostrará la fórmula del flujo óptico, partiendo de la definición de la intensidad *I(x,y,t)*de una secuencia de imágenes:

Partiendo de la premisa que, si hay un movimiento:

$$I(x, y, t) = I(x + \partial x, y + \partial y, t + \partial t).$$

Ahora asumiendo que las diferenciales no son grandes, ya que los movimientos considerados son pequeños, podemos tomar las primeras derivadas de una serie de Taylor de la función de la intensidad:

$$I(x + \partial x, y + \partial y, t + \partial t) = \frac{\partial I}{\partial x}\partial x + \frac{\partial I}{\partial y}\partial y + \frac{\partial I}{\partial t}\partial t + I(x, y, t)$$

De la primera igualdad y la segunda igualdad se extraen:

$$\frac{\partial I}{\partial x}\partial x + \frac{\partial I}{\partial y}\partial y + \frac{\partial I}{\partial t}\partial t = 0$$

Dividiendo todo por la diferencial del tiempo:

$$\frac{\partial I}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial I}{\partial y}\frac{\partial y}{\partial t} + \frac{\partial I}{\partial t}\frac{\partial t}{\partial t} = 0$$

y definiendo:

$$\frac{\partial x}{\partial t} = v_x$$
$$\frac{\partial y}{\partial t} = v_y$$

y las derivadas de la intensidad luminosa:

$$I_x = \frac{\partial I}{\partial x}$$
$$I_y = \frac{\partial I}{\partial y}$$
$$I_t = \frac{\partial I}{\partial t}$$

Finalmente, la expresión del flujo óptico que relaciona las velocidades con la intensidad lumínica

queda:

$$I_x * v_x + I_y * v_y + I_t = 0$$

Como se puede comprobar no basta sólo con una ecuación del flujo óptico ya que es un sistema indeterminado al tener una ecuación con dos incógnitas. Para solventar este problema se introduce el concepto de píxeles vecinos, donde se propone elegir un entorno de ellos, centrado en el píxel central para tener un número mayor de ecuaciones y así, mediante el método de mínimos cuadrados, obtener como resultado un sistema sobredimensionado, en las que pueden resolverse las ecuaciones de flujo óptico. Para cierta ventana de *nxn* píxeles, y *p1,..pn*, los píxeles de dicha ventana tendremos el siguiente sistema:

$$\begin{gathered} I_x(p1) * v_x + I_y(p1) * v_y = -I_t(p1) \\ I_x(p2) * v_x + I_y(p2) * v_y = -I_t(p2) \\ . \\ . \\ . \\ I_x(pn) * v_x + I_y(pn) * v_y = -I_t(pn) \end{gathered}$$

Que en forma matricial nos quedaría de la siguiente forma :$A = \begin{bmatrix} I_x(p1) & I_y(p1) \\ I_x(p2) & I_y(p2) \\ & . \\ I_x(pn) & I_y(pn) \end{bmatrix}$

$$v = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$

$$b = \begin{bmatrix} -I_t(p1) \\ -I_t(p2) \\ . \\ -I_t(pn) \end{bmatrix}$$

Mediante el principio de mínimos cuadrados, la expresión que calcula la velocidad en ese píxel es:

$$V = (A^T A)^{-1} A^T b$$

De una forma más general podríamos expresar esta ecuación de la forma siguiente:

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} \sum_i I_x(pn)^2 & \sum_i I_x(pn) I_y(pn) \\ \sum_i I_x(pn) I_y(pn) & \sum_i I_y(pn)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_x(pn) I_t(pn) \\ -\sum_i I_y(pn) I_t(pn) \end{bmatrix}$$

De esta forma queda desarrollado el algoritmo de Lucas-Kanade para la versión basada en *frame*.
Pero nuestro objetivo es la realización de un filtro basado en eventos, ya descrita en el primer capítulo, por lo que se debe modificar el algoritmo para adaptarlo y que mediante un flujo de eventos sea capaz de estimar el campo de velocidades.

4.4 Algoritmo de Lucas Kanade mediante flujo de eventos

Este algoritmo es muy parecido al descrito, pero con la peculiaridad de que las derivadas las calcula basándose en eventos [4]. Partiendo del principio del flujo óptico descrito en el anterior apartado, se realiza una modificación del concepto de derivada de intensidad al no existir escalas de grises en la imagen. Lo que se realiza es un conteo de los eventos en una ventana *NxN*en un tiempo Δt, por lo que las derivadas se pueden aproximar en este caso como:

$$\frac{\partial e(x,y,t)}{\partial x} \approx \sum_{t-\Delta t}^{t} e(x,y,t) - \sum_{t-\Delta t}^{t} e(x-1,y,t)$$

$$\frac{\partial e(x,y,t)}{\partial y} \approx \sum_{t-\Delta t}^{t} e(x,y,t) - \sum_{t-\Delta t}^{t} e(x,y-1,t)$$

La estimación del gradiente temporal es más exacta debido a la precisión de la DVS. Se puede escribir este gradiente como:

$$\frac{\partial e(x,y,t)}{\partial t} \approx \frac{\sum_{t-\Delta t}^{t1} e(x,y,t) - \sum_{t-\Delta t}^{t} e(x,y,t-1)}{t1-t}$$

Por lo que la ecuación del flujo óptico para el caso de eventos tiene la siguiente forma:

$$\left(\sum_{t-\Delta t}^{t} e(x,y,t) - \sum_{t-\Delta t}^{t} e(x-1,y,t)\right) v_x + \left(\sum_{t-\Delta t}^{t} e(x,y,t) - \sum_{t-\Delta t}^{t} e(x,y-1,t)\right) v_y = \frac{\sum_{t1}^{t} e(x,y,t)}{t1-t}$$

Siendo:

$$\sum_{t1}^{t} e(x,y,t) = \sum_{t-\Delta t}^{t1} e(x,y,t) - \sum_{t-\Delta t}^{t} e(x,y,t-1)$$

Como en la versión basada en *frame*, se debe calcular para una ventana NxN y posteriormente usar el método de mínimos cuadrados para el cálculo de la velocidad en la ventana. Un resumen del algoritmo seguido sería el siguiente:

Algoritmo de flujo óptico mediante eventos

Para cada evento e(x, y, t) que se recibe hay que:

1. Definir una ventana espacio temporal (N x N x $\Delta \mathbf{t}$)de píxeles vecinos al (x, y)
2. Calcular las derivadas:

$$\frac{\boldsymbol{\partial e(x,y,t)}}{\boldsymbol{\partial x}}$$

$$\frac{\boldsymbol{\partial e(x,y,t)}}{\boldsymbol{\partial y}}$$

$$\frac{\boldsymbol{\partial e(x,y,t)}}{\boldsymbol{\partial t}}$$

3. Resolver las ecuaciones del flujo óptico mediante mínimos cuadrados como lo expuesto en el primer apartado.

4.5 Algoritmo de Horn- Schunck

Este algoritmo está basado en el de Lucas-Kanade, pero con la diferencia de que es una versión iterativa, por lo que no se usa la resolución de ecuaciones mediante el método de mínimos cuadrados. Parte de las restricciones de que la función del brillo, denotada como *E(x,y,t)* en [7], es diferenciable. Más adelante cuando se extienda a imágenes creadas mediante la acumulación de eventos se verá que esta suposición sigue siendo válida.
Considerando que el brillo de un patrón particular es constante, esto es:

$$\frac{dE}{dt} = 0$$

Usando la regla de la cadena, se nota que:

$$\frac{\partial E}{\partial x}\frac{dx}{dt} + \frac{\partial E}{\partial x}\frac{dx}{dt} + \frac{\partial E}{\partial t} = 0$$

Definiendo ahora:

$$u = \frac{dx}{dt}$$

$$v = \frac{dy}{dt}$$

Por lo que se obtiene una ecuación de flujo óptico del algoritmo de Lucas Kanade:

$$E_x u + E_y v + E_t = 0$$

Pero hay dos incógnitas en esta ecuación que son desconocidas u y v, que son las componentes de velocidad del campo de velocidades en un punto de dicho campo. Para resolver esta ecuación hace falta añadir otra restricción más: la restricción de suavidad.
Supone que la velocidad en todo el campo de velocidades varía suavemente y sin discontinuidades, excepto en los bordes de los objetos, teniendo problemas en las esquinas. Una manera de expresar esta restricción es minimizar los gradientes del flujo óptico:

$$\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial x}\right)^2$$

y

$$\left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2$$

Si el observador se desplaza con el movimiento del objeto, los gradientes tienden a desaparecer y no hay movimiento, que es lo que se cabría esperar.

4.5.1 Estimación de las derivadas parciales del flujo óptico

Para el cálculo de las derivadas del flujo óptico, se deben averiguar las derivadas de la intensidad para las direcciones (x, y), y la temporal. Como sólo tenemos los valores de brillo de la imagen, que son valores discretos, entonces las derivadas se pueden aproximar con las siguientes fórmulas:

$$E_x \approx \frac{1}{4}\left(E_{i,j+1,k} - E_{i,j,k} + E_{i+1,j+1,k} - E_{i+1,j,k} + E_{i,j+1,k+1} - E_{i,j,k+1} + E_{i+1,j+1,k+1} - E_{i+1,j,k+1}\right)$$

Para las derivadas en y:

$$E_y \approx \frac{1}{4}\left(E_{i+1,j,k} - E_{i,j,k} + E_{i+1,j+1,k} - E_{i,j+1,k} + E_{i+1,j,k+1} - E_{i,j,k+1} + E_{i+1,j+1,k+1} - E_{i,j+1,k+1}\right)$$

Para la derivada temporal:

$$E_t \approx \frac{1}{4}\left(E_{i,j,k+1} - E_{i,j,k} + E_{i+1,j+1,k} - E_{i+1,j,k} + E_{i,j+1,k+1} - E_{i,j+1,k} + E_{i+1,j+1,k+1} - E_{i+1,j+1,k}\right)$$

4.5.2 Estimación del laplaciano del flujo óptico

También necesitamos la aproximación del laplaciano de *u* y *v*, ya que el algoritmo que se va a desarrollar es de tipo numérico. Una aproximación conveniente es la siguiente:

$$\nabla^2 u \approx k\left(\overline{u}_{i,j,k} - u_{i,j,k}\right)$$

y

$$\nabla^2 v \approx k\left(\overline{v}_{i,j,k} - v_{i,j,k}\right)$$

donde $\overline{u}$ *y* $\overline{v}$ se pueden expresar de la siguiente forma:

$$\overline{u}_{i,j,k} \approx \frac{1}{6}\left(u_{i-1,j,k} + u_{i,j+1,k} + u_{i+1,j,k} + u_{i,j-1,k}\right) + \frac{1}{12}\left(u_{i-1,j-1,k} + u_{i-1,j+1,k} + u_{i+1,j+1,k} + u_{i+1,j-1,k}\right)$$

y para la componente v, se tiene una expresión parecida:

$$\overline{v}_{i,j,k} \approx \frac{1}{6}\left(v_{i-1,j,k} + v_{i,j+1,k} + v_{i+1,j,k} + v_{i,j-1,k}\right) + \frac{1}{12}\left(v_{i-1,j-1,k} + v_{i-1,j+1,k} + v_{i+1,j+1,k} + v_{i+1,j-1,k}\right)$$

el factor k es computado como 1 para efectos prácticos del cálculo [7]. Para simplificar lo anterior se pueden formular las anteriores expresiones en una matriz que será el kernel de la convolución. Esta matriz tiene la siguiente forma:

$$\begin{bmatrix} \frac{1}{12} & \frac{1}{12} & \frac{1}{12} \\ \frac{1}{6} & -1 & \frac{1}{6} \\ \frac{1}{12} & \frac{1}{12} & \frac{1}{12} \end{bmatrix}$$

4.5.3 Minimización y obtención de las ecuaciones

El problema se resume en minimizar la expresión de abajo, ya que el brillo de un determinado patrón es constante y ello hace que la derivada de la intensidad luminosa es 0 como se ha demostrado en la sección anterior:

$$\delta_b = E_x u + E_y v + E_t$$

También se considera la medida de la suavidad del flujo óptico:

$$\delta_c = \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2$$

Para minimizar estos dos factores se debe considerar un factor debido a la cuantificación de la imagen y el ruido en el sensor, y en nuestro caso, como este algoritmo se va a usar para imágenes obtenidas de la acumulación de eventos, es necesario que se considere dicho factor para eliminar lo más posible efectos de ruido, y de cuantificación:

$$\delta^2 = \iint \left(\alpha^2 {\delta_c}^2 + {\delta_b}^2\right) dxdy$$

Para minimizar esta expresión, se deben encontrar los valores de *u* y *v* que hagan posible dicha condición. Esta funcional, mediante la teoría del cálculo de variaciones se llega a[7]:

$${E_x}^2 u + E_x E_y v = \alpha^2 \nabla^2 u - E_x E_t$$

y

$$E_x E_y u + {E_y}^2 v = \alpha^2 \nabla^2 v - E_y E_t$$

Y usando la aproximación Laplaciana introducida en el apartado anterior [7]:

$$\begin{cases} (\alpha^2 + {E_x}^2)u + E_x E_y v = \alpha^2 \overline{u} - E_x E_t \\ (\alpha^2 + {E_y}^2)v + E_x E_y u = \alpha^2 \overline{v} - E_y E_t \end{cases}$$

Resolviendo estas ecuaciones y sabiendo que el determinante de este sistema es: $\alpha^2(\alpha^2 + {E_x}^2 + {E_y}^2)$:

$$(\alpha^2 + {E_x}^2 + {E_y}^2)u = (\alpha^2 + {E_x}^2)\overline{u} - E_x E_y \overline{v} - E_x E_t$$

y para la dimensión v:

$$(\alpha^2 + {E_x}^2 + {E_y}^2)v = -E_x E_y \overline{v} - (\alpha^2 + {E_x}^2)\overline{v} - E_y E_t$$

Estas ecuaciones muestran el valor de las componentes *u* y *v* del flujo de velocidad en un punto que minimiza δ^2en la dirección hacia la línea de restricción entre una línea que intersecta con esta perpendicularmente.

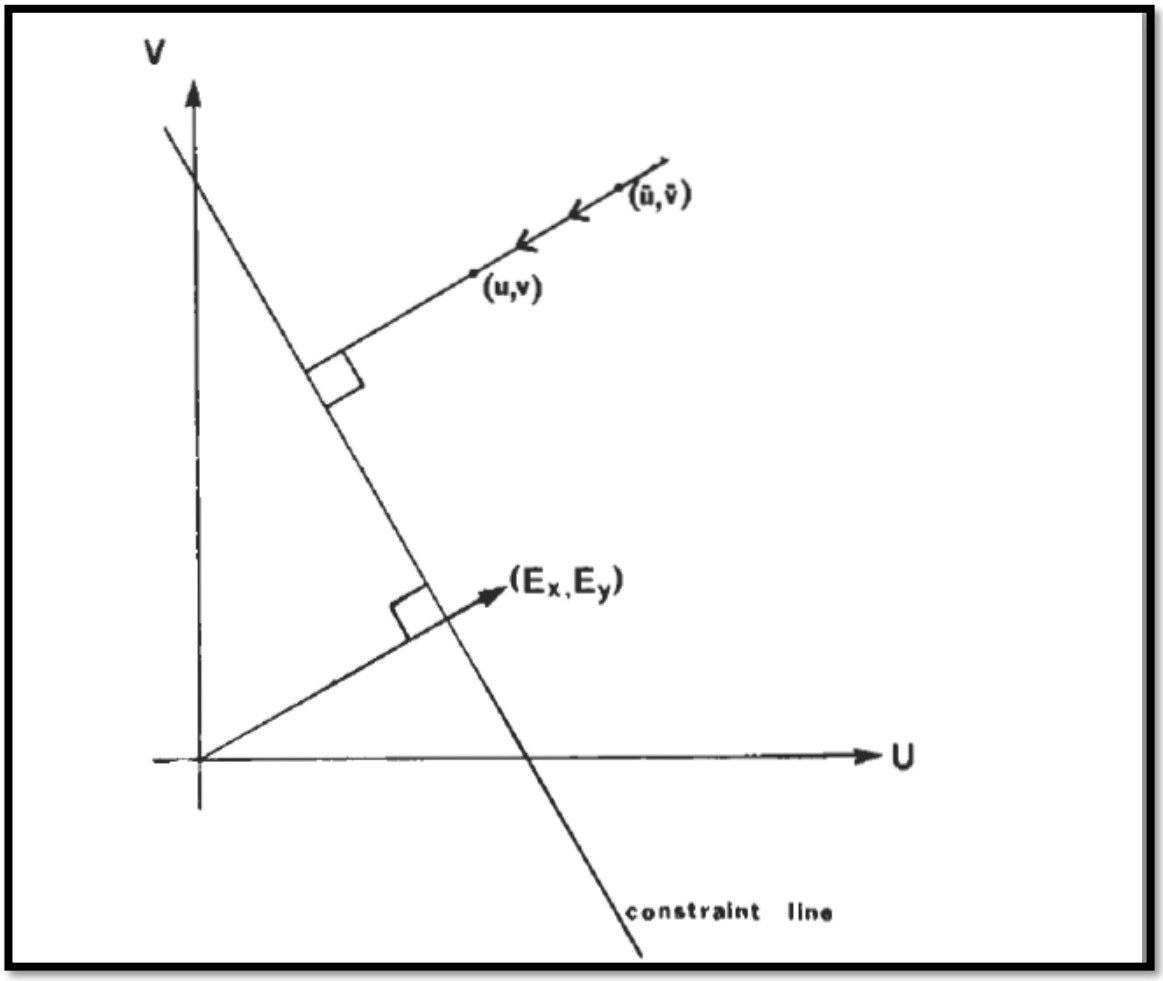

Ilustración 14: Figura que explica la relación geométrica de las derivadas con las componentes del flujo óptico. [7]

4.5.4 Solución de las ecuaciones

Para resolver las ecuaciones del flujo óptico se hace uso del método numérico iterativo de resolución de ecuaciones, como es el método Gauss-Seidel. De esta forma obtenemos las siguientes ecuaciones que solucionan el flujo óptico:

$$u^{n+1} = \overline{u}^n - \frac{E_x\left[E_x\overline{u}^n + E_y\overline{v}^n + E_t\right]}{(\alpha^2 + {E_x}^2 + {E_y}^2)}$$

y para *v*:

$$v^{n+1} = \overline{v}^n - \frac{E_y\left[E_x\overline{u}^n + E_y\overline{v}^n + E_t\right]}{(\alpha^2 + {E_x}^2 + {E_y}^2)}$$

La dependencia que tiene el factor α^2en el flujo óptico es cuando el gradiente de luminosidad es pequeño, y podría verse afectado por el ruido o por la cuantificación.

4.6 Algoritmo de Horn-Schunck para eventos

En este caso se puede usar este algoritmo para las imágenes obtenidas por la acumulación deeventos, ya sea recogiendo los eventos cada intervalo regular de tiempo o acumulando un número constante de eventos. Con esto se crea una imagen. Entonces, una vez obtenida dicha imagen, se elige un α particular y se ejecuta el algoritmo de flujo ópticoHorn-Schunck descrito en el apartado anterior.

4.7 Algoritmo Block Matching

Cada una de las imágenes pertenecientes a una secuencia de vídeo se divide en bloques rectangulares (generalmente cuadrados) denominados **macrobloques**. El método pretende detectar el movimiento entre imágenes, comparando los macrobloques de imágenes consecutivas en una secuencia de vídeo.

Los bloques del fotograma actual son cotejados con los del fotograma de destino o de referencia. Deslizando el macrobloque a lo largo de una región concreta de píxeles del fotograma de destino para encontrarlo en la siguiente imagen. [8]

Pueden existir dos tipos de cotejos: Hacia atrás(backward) y hacia delante(forward) [9]. En el cotejo hacía atrás, el macrobloque de referencia pertenece al *frame* actual y es cotejado con cada uno de los inmediatamente anteriores. En el cotejo "hacía delante", sucede lo contrario.

Una vez que se ha encontrado el macrobloque en el *frame* candidato, se puede estimar el vector de movimiento a partir de la posición del macrobloque de referencia. En la siguiente figura se muestra el proceso:

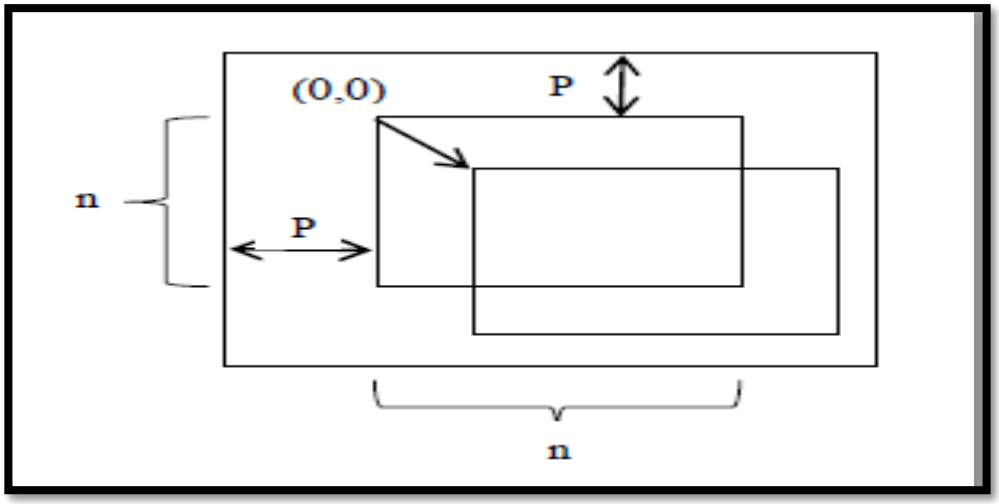

Ilustración 15: Estimación del vector del movimiento, desde el movimiento de un macrobloque [10].

A medida que el tamaño de los macrobloques aumenta, el número de estos disminuye, pero también el coste computacional de cotejarlos con otros aumentará. También el aumento de su tamaño incrementa la posibilidad de coincidencia, debido a que cada macrobloque contiene más información.

4.7.1 Criterio de cotejo en el block matching

El objetivo del block matching es la comparación entre imágenes o partes de ellas [10]. Para alcanzar tal objetivo se debe disponer de una medida de similitud de correlación que nos muestre la disparidad

entre las imágenes o bloques de cada una de ellas.Para realizar el cotejo y encontrar el macrobloque del *frame* de referencia en el *frame* candidato, se propone una serie de funciones de coste para realizar el cotejo, como el "Mean Square Error (MSE)", el " Mean Absolute Diferences(MAD)" y "el PeakSignals to Noise Ratio(PNSR)". Cabe decir que el PNSR es el más popular, porque determina el movimiento de la imagen y se calcula mediante el vector de movimiento de la imagen original. Seguidamente se detallan las fórmulas de cada uno de los criterios:

$$MSE = \frac{1}{n^2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\left(C_{ij} - R_{ij}\right)^2$$

Siendo C_{ij} y R_{ij} los macrobloques de las imágenes de referencia y candidata, y *n* el tamaño del bloque,

$$MAD = \frac{1}{n^2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\left|C_{ij} - R_{ij}\right|$$

Y para la medida del PNSR:

$$PNSR = 10Log_{10}\left[\frac{255^2}{MSE}\right]$$

4.7.2 Algoritmos del block matching

Para realizar la búsqueda de un macrobloque en la imagen objetivo, se han propuestos numerosos algoritmos, algunos de los cuales pasaremos a describirlos a continuación:

4.7.2.1 Búsqueda exhaustiva

Es el primer algoritmo que se nos ocurre cuando tratamos de resolver el problema de búsqueda. Se trata de buscar el macrobloque dentro de un espacio de búsqueda, centrado en la posición del *frame* de origen de donde procede el macrobloque. Este algoritmo requiere gran cantidad de cálculo, por lo que se ha intentado encontrar estrategias que minimicen el coste computacional, y que tengan un PSNR tan alto como este algoritmo.

4.7.2.2 Búsqueda en tres pasos

Este algoritmo fue uno de los intentos para mejorar el algoritmo de búsqueda exhaustiva. Surgió a mediados de los 80. La idea se representa en el siguiente esquema:

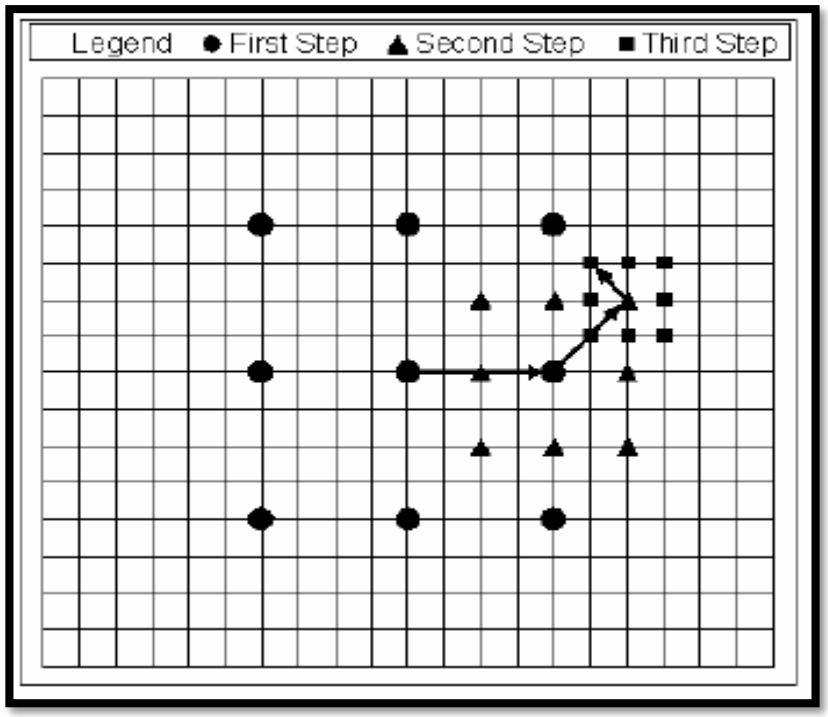

Ilustración 16: Esquema de funcionamiento del algoritmo "Threestepssearch" [11]

El funcionamiento de este algoritmo es el siguiente:
Se comienza con la búsqueda del macrobloque candidato en el centro y después se realiza un salto con un tamaño grande de S=4(siendo S el número de píxeles de dicho salto), para un tamaño usual de macrobloque se toma S=7 [11]. Entonces se procede con la búsqueda en las 8 direcciones en +/- *S* píxeles de distancia del centro que se tome de referencia. Una vez que se ha realizado dicha búsqueda, en las 8 direcciones se toma la que tiene menos coste, esto es, la que es más similar al macrobloque del *frame* de origen, y se pone como origen de la nueva búsqueda. Se hace esta vez a *S/2*píxeles y se toman, como en el caso anterior, las 8 direcciones. Esto se repite, hasta que tengamos *S=1*, y al final nos da la dirección del movimiento en ese instante. De esta forma se produce una reducción del coste de computación, por un factor de 9.
La idea detrás de la búsqueda en tres pasos es que la superficie de error debida al movimiento en cada macrobloque es unimodal (sólo tiene un mínimo). [11]Una superficie unimodal es aquella en forma de bol, que es monótonamente creciente a partir del mínimo global. Un ejemplo de esta superficie sería la siguiente:

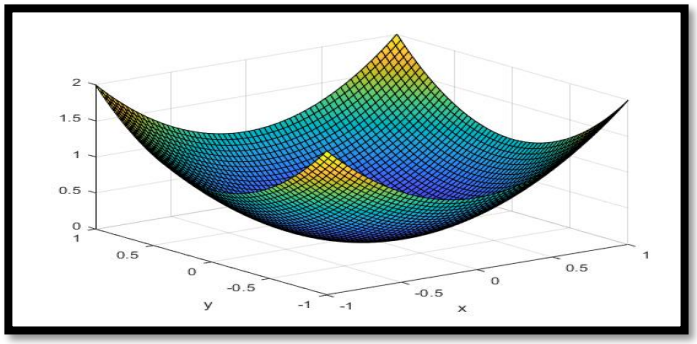

Ilustración 17: Ejemplo de superficie unimodal [12]

4.7.2.3 Nuevo algoritmo de búsqueda en tres pasos (NTSS)

Es un algoritmo que mejora el anterior TSS, y que proporciona un esquema "de influencia central" y tiene previsiones "de paro a medio camino"[11]. En la siguiente figura se ilustra el funcionamiento de este algoritmo:

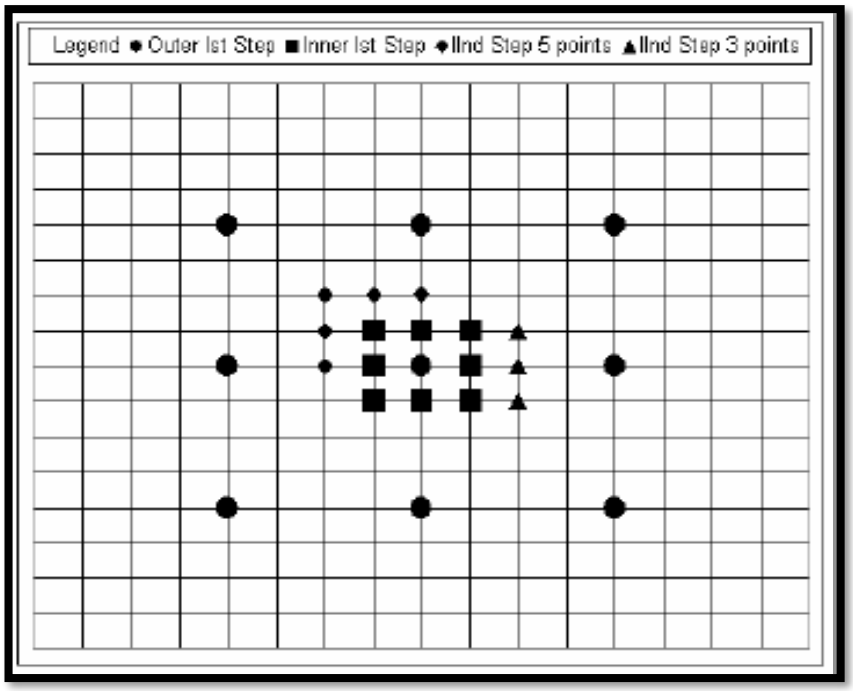

Ilustración 18: Figura que ilustra el funcionamiento del algoritmo NTSS [11].

En el primer paso se realiza una búsqueda exhaustiva de 16 puntos, centrada en el origen de la búsqueda. De esas 16 búsquedas, 8 son a S=4 y las otras 8 son a S=1, donde S es la distancia en píxeles al origen. Si se encuentra el macrobloque candidato en la misma posición que el de referencia, es decir, no se ha producido movimiento, entonces no se realiza búsqueda alguna [11]. Pero si el macrobloque se encuentra en uno de los 8 puntos en S=1, entonces ponemos el origen en ese punto y buscamos en ese punto.
Dependiendo de la localización, tendremos de 3 a 5 búsquedas. De otra manera, si encontramos el mínimo en una de las 8 localizaciones en S=4, entonces seguiremos con el algoritmo TSS normal. Este algoritmo podría necesitar un mínimo de 17 comparaciones y en el peor de los casos unas 33 comparaciones.

4.7.2.4 Búsqueda simple y eficiente (SES)

Este algoritmo es una extensión del TSS y explora la suposición de que la superficie de error tiene un sólo mínimo, y por lo tanto el patrón de 8 direcciones del TSS puede ser cambiado, reduciendo el coste computacional [11]. Este algoritmo aun cuenta con tres pasos como en el TSS, pero la diferencia con respecto al TSS es que cada paso tiene dos fases. La búsqueda se divide en cuatro cuadrantes y el algoritmo chequea tres puntos A, B y C [11]. En la siguiente figura se muestra gráficamente el proceso:

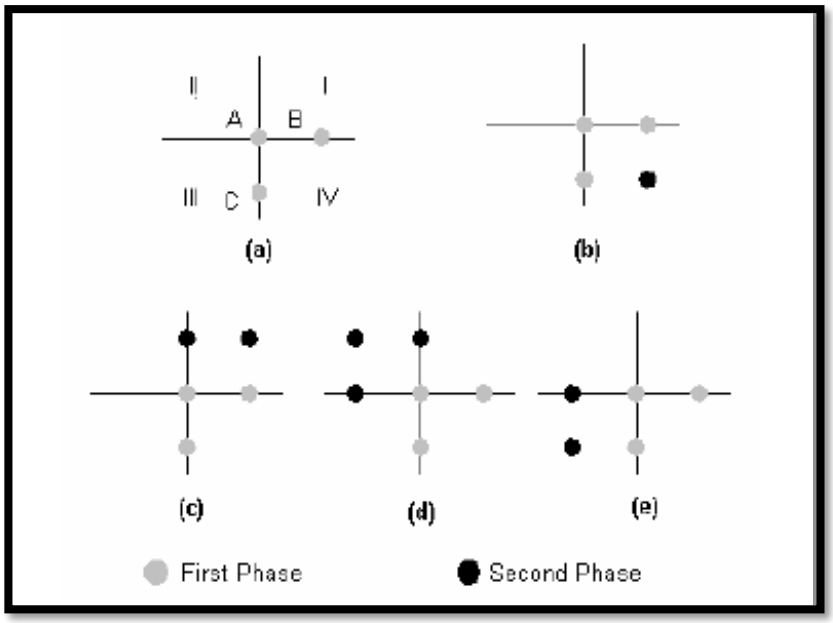

Ilustración 19: Cuatro cuadrantes y 2 fases de búsqueda en el algoritmo SES [11].

El punto A es el origen y los puntos B y C son puntos en S=4, en alguna dirección ortogonal a A. Dependiendo de la distribución de los valores de la función de coste entre los tres puntos, se determina qué cuadrante de búsqueda se selecciona. Las reglas que determinan la búsqueda en la segunda fase son las siguientes:

- Si MAD(A) ≥ MAD(B) y MAD(A) ≥ MAD(C), entonces selecciona b;
- Si MAD(A) ≥ MAD(B) y MAD(A) ≤ MAD(C), entonces selecciona c;
- Si MAD(A) < MAD(B) y MAD(A) < MAD(C), entonces selecciona b;
- Si MAD(A) < MAD(B) y MAD(A) ≤ MAD(C), entonces selecciona e;

Una vez seleccionados los puntos, se chequean para encontrar una coincidencia en uno de esos puntos, y entonces ponemos de origen el punto encontrado. Después actuamos de manera similar al TSS y repetimos hasta que S=1, entonces la localización encontrada, será el movimiento de esa parte de la imagen. Este algoritmo reduce bastante el coste de computacional comparado con el TSS[11], pero no es ampliamente aceptado por dos razones fundamentales:

Primero, porque la superficie de error no es estrictamente unimodal, y por lo tanto el PSNR mejorado es pobre, comparado con el TSS. Segundo, hay otro algoritmo que entraremos a describir a continuación que fue publicado un año antes que presenta un coste computacional más bajo comparado con el TSS y da un PSNR significativamente mejor que el TSS.

4.7.2.5 Búsqueda en cuatro pasos

Este algoritmo es muy similar al NTSS, porque también emplea "búsqueda de influencia de centro" y tiene "parada provisional a medio camino". Este algoritmo, elige un patrón fijo de búsqueda con S=2[11] para el primer paso. En la figura de abajo se explica gráficamente el algoritmo:

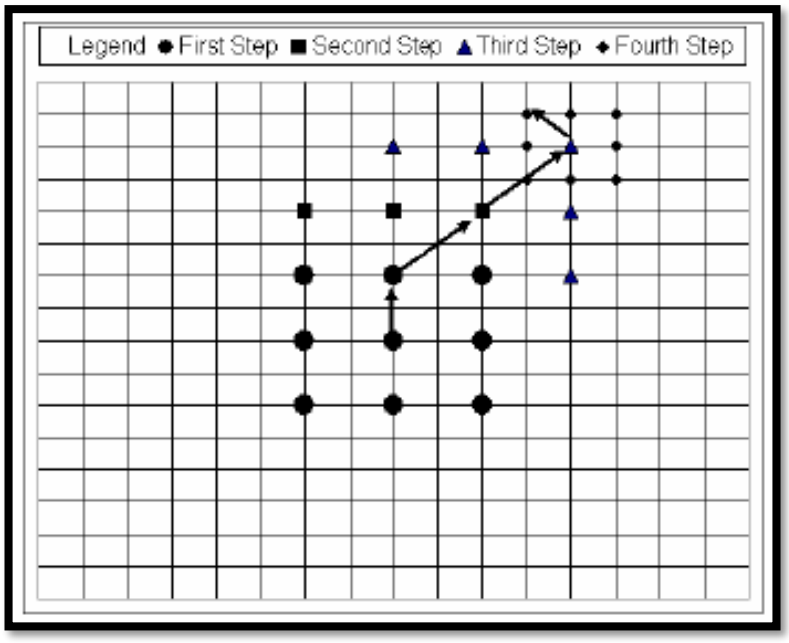

Ilustración 20: Algoritmo FSS [11]

Si hay coincidencia en el centro de la búsqueda, se devuelve el vector (0,0) y se finaliza la búsqueda. Si el mínimo es encontrado en una de las ocho posiciones, entonces hacemos que ese punto sea el origen del segundo paso. Dependiendo de la situación del punto nos encontramos las siguientes situaciones:

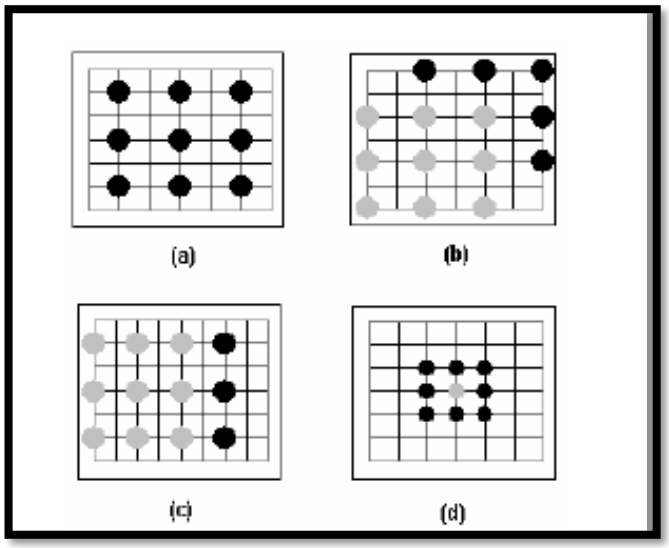

Ilustración 21: Patrones de búsqueda en cada uno de los pasos, a es el primer paso, b y c el segundo y el tercero y d es el cuarto paso. [11]

Como se puede observar en la figura, se tienen que chequear de 3 a 5 posiciones, dependiendo de la posición del centro en los pasos dos y tres.En el paso dos, si encuentra coincidencia en el centro pasa al cuarto paso, pero si lo encuentra en una de las ocho posiciones, entonces pasa al siguiente paso, el tercero, y en el tercero repetimos lo mismo, hasta llegar al cuarto paso, cuya ventana es de S=1,y se busca el punto donde sea mínima la función de coste, y se obtiene el vector de movimiento. En el mejor de los casos, este algoritmo, chequea 17 puntos hasta llegar a la solución; y en el peor de los casos chequea unos 27 casos.

4.7.2.6 Búsqueda diamante

El algoritmo de búsqueda diamante es muy similar al algoritmo de cuatro pasos, pero el patrón de búsqueda cambia de un cuadrado a un diamante, y no hay límites en el número de pasos que puede dar el algoritmo. Usa dos tipos de patrón de búsquedas: el primero es el llamado "gran patrón diamante de búsqueda"(LDSP) y el otro es el pequeño patrón diamante de búsqueda (SDPS). En la figura de abajo se muestra el funcionamiento del algoritmo con dichos patrones de búsquedas:

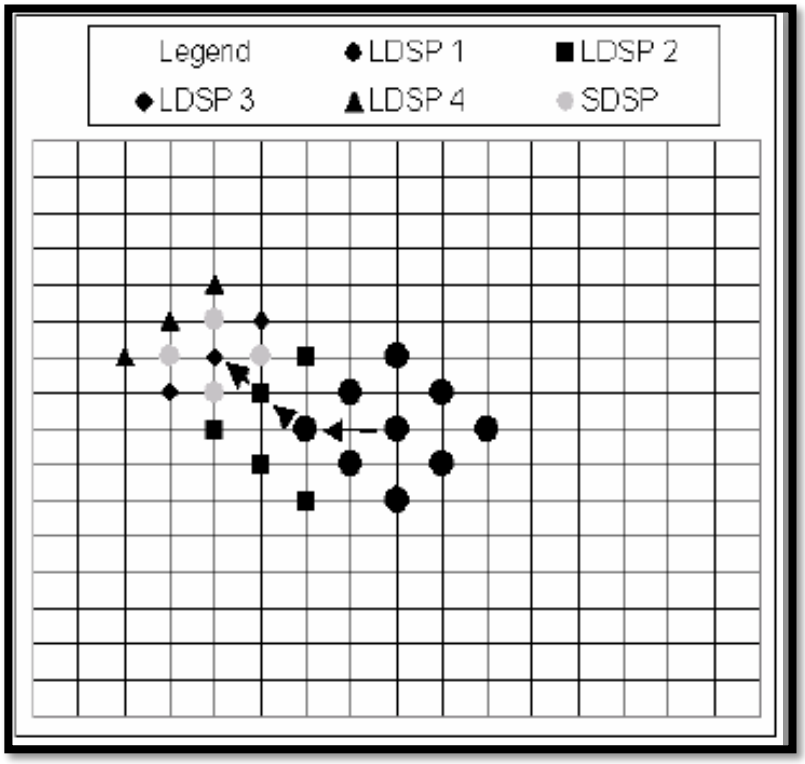

Ilustración 22: Patrón de búsqueda diamante [11]

El primer paso es como la "búsqueda de cuatro pasos", si hay coincidencia en el centro pasa al cuarto paso y termina la búsqueda. El resto de los pasos es muy similar excepto en el último que se usa el algoritmo LDSP. En el último paso usa SDSP en conjunto con el algoritmo de LDSP para mejorar el coste computacional. Como la búsqueda que se realiza es muy pequeña tiene poco coste computacional y el algoritmo tiene mucha precisión. Su PSNR es muy similar a la búsqueda exhaustiva.

4.7.2.7 Búsqueda de patrón cruz adaptativa

Este algoritmo usa el hecho de que el movimiento es generalmente predecible.[11] Esto se puede explicar de la siguiente manera: si un macrobloque actual tiene un determinado vector de movimiento en el instante actual, en el *frame* posterior habrá una alta probabilidad de tener un vector muy parecido al actual. Este algoritmo usa el vector de movimiento del macrobloque inmediatamente izquierdo. Un ejemplo del algoritmo se tiene en la siguiente figura:

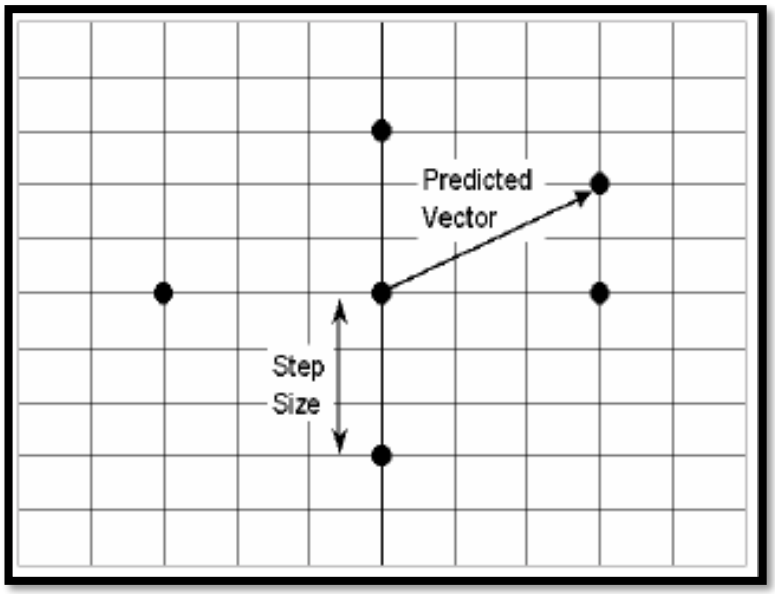

Ilustración 23: Ejemplo del funcionamiento. En la figura se observa el vector predicho y el patrón de búsqueda [11]

El paso S se calcula como S=$max(|X|, |Y|)$, donde x e y son las coordenadas del vector predicho de movimiento. Este patrón expuesto es siempre el primer paso, después el algoritmo se basa en este vector de movimiento para buscar en un área donde es más probable de encontrar una coincidencia. Una vez que se ha encontrado, entonces para afinar se pasa al SDSP, descrito anteriormente y se procede la búsqueda hasta que se encuentra en la ubicación precisa del nuevo vector de movimiento. Si la coincidencia estuviera en el centro, se pararía el algoritmo y devolvería las coordenadas del centro.

Unas de las ventajas de este algoritmo, con respecto a la búsqueda mediante el patrón diamante, es que directamente empieza buscar con el patrón SDSP, ya que se ha predicho, mediante el movimiento, la ubicación del nuevo vector de movimiento, por lo que el coste computacional de este nuevo algoritmo es menor que incluso el DS, ya que se salta el paso de usar LDSP.

4.8 Algoritmo de Block Matching basado en eventos

4.8.1 Introducción

Hasta ahora se han descrito algunos de los algoritmos más importantes del Block Matching clásico. Ahora se va a describir un algoritmo de block-matching basado en eventos, descrito en [13]. Este algoritmo parte de la premisa de que los eventos de la cámara DVS, que se producen por la variación de la luminosidad, son acumulados en porciones de RAM, en forma de imagen binaria. Al no considerarse la polaridad, cada píxel de dicha imagen binaria será producido por la aparición de un evento. El bloque que se considera como referencia es aquél que está situado en torno a la localización del evento actual. Para realizar el cotejo con el *frame* candidato, o en este caso con el "slide" candidato, la función de coste es la distancia métrica entre los dos bloques. En este caso se propone la "distancia de Hamming". Esta distancia se basa en contar cuantos bits han cambiado de un bloque a otro. Esta medida es exactamente la misma que en el caso de imágenes, la "Sum of Absolute Difference" (SAD).

4.8.2 Algoritmo

Cuando un evento llega, el bloque de referencia, que en nuestro caso es el "slide t-d", quiere decir

que está a un intervalo de tiempo del evento actual. Cada bloque contiene 9x9 eventos. El algoritmo coteja el bloque centrado en la posición actual de t-d con los 8 bloques adyacentes en t-2d, para ver si hay coincidencias.
La distancia de Hamming está implementada mediante puertas XOR, que calcula la distancia de Hamming entre los 8 bloques. En la figura de abajo se muestra el esquema del cálculo de HD:

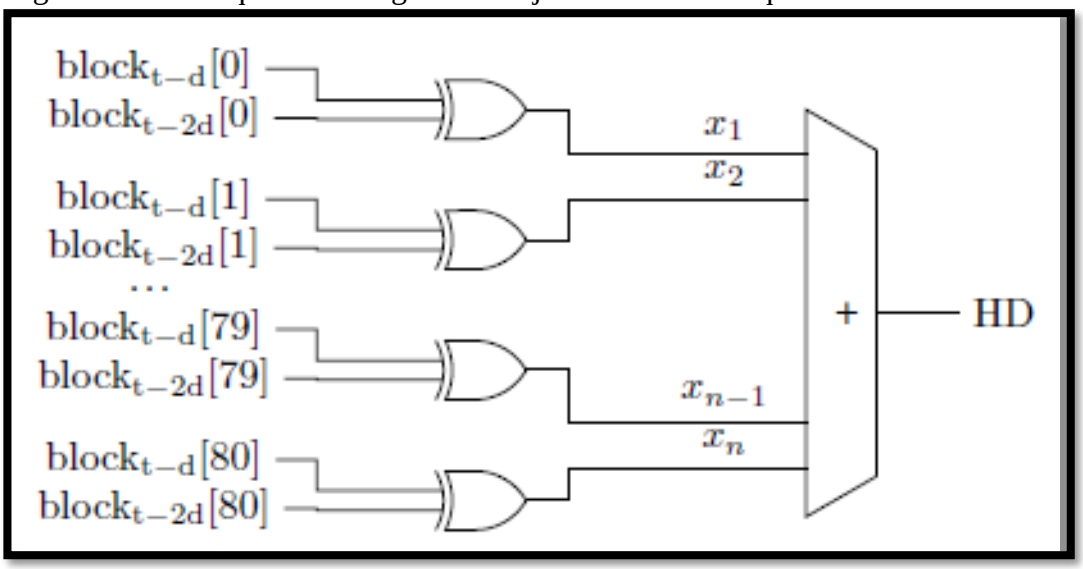

Ilustración 24: Estructura de cálculo de HD mediante circuitos combinacionales [13].

Para calcular cúal es la distancia mínima entre los 8 bloques, para cada una de las ocho direcciones, se ha propuesto también una arquitectura en [13], que se basa en la realización en paralelo de las operaciones de comparación. En la figura de abajo se muestra la arquitectura de ejemplo del concepto que se ha propuesto:

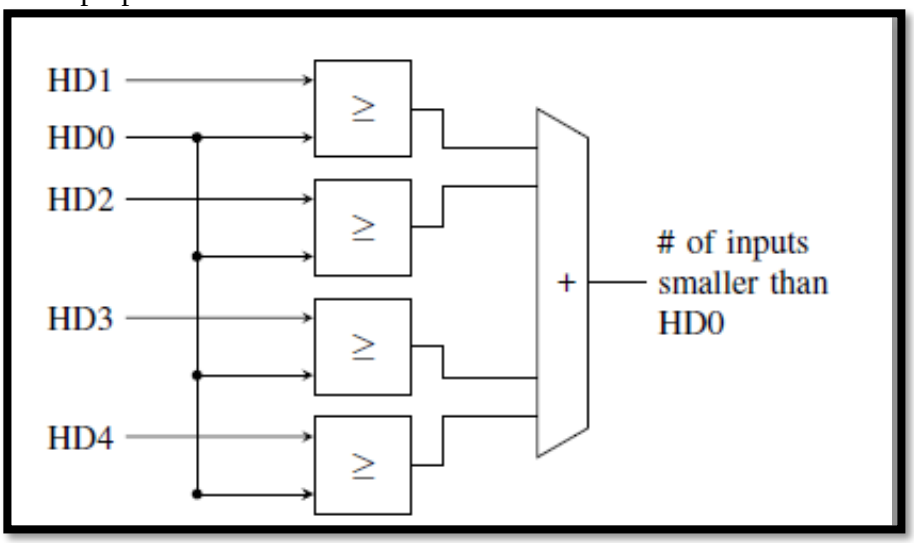

Ilustración 25: Arquitectura para calcular la distancia mínima de 5 distancias

5 DETECTOR ELEMENTAL DE MOVIMIENTO

5.1 Introducción

En los estudios de Egelhaaf y Reichardt, se describieron estructuras que eran sensibles al movimiento en dos direcciones del eje de acción.
Estas investigaciones fueron llevadas a cabo en moscas [14],[15], y en otros insectos. De los estudios fisiológicos del sistema nervioso de las moscas surgió la idea de que el mecanismo detector de movimiento ha de satisfacer tres requerimientos:
(1) Debe tener al menos dos canales; (2) estrictamente hablando, debe tener naturaleza no lineal, ya que no detectaría la dirección debido a que si fuera lineal no distinguiría la dirección; y (3) el detector debe ser asimétrico, ya que, si fuera simétrico en los dos canales, no podría detectar movimiento alguno.

Solo queda una cuestión: la naturaleza no lineal de la interacción entre canales. Las evidencias apuntan a que esta naturaleza es de carácter correlacional, esto es, que el movimiento es estimado mediante un tipo de correlación espaciotemporal.

Esta correlación es en esencia una multiplicación de los dos canales, por lo que el detector adquiere una naturaleza no lineal [14].
Este detector puede incluso, debido a su carácter no lineal, distinguir la dirección del movimiento. Este esquema fue descubierto en el sistema nervioso de una mosca, donde además se vio que esta estructura se replica, en una matriz espacial y puede llegar a medir la velocidad, en un cierto rango dinámico [14]. El detector encuentra el movimiento, cuando el objeto o el patrón en este caso, se mueva a partir de una cierta velocidad, y llegado a un cierto límite, se provoca aliasing en la medida de la velocidad [14],[15].

5.2 Análisis del detector

El detector de movimiento planteado se muestra en la siguiente figura:

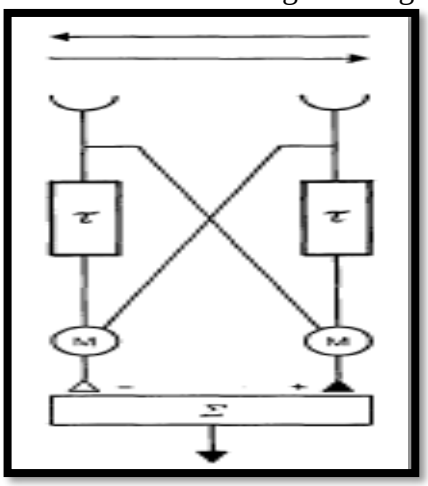

Ilustración 26: Detector elemental de movimiento

Como se puede observar en esta figura, este detector tiene dos canales separados por una distancia d, llamada en [14] "muestreo de base". Se puede observar en su estructura que consiste en dos subunidades, en la que un canal es apropiadamente retardado, mediante un filtro lineal [14]. Esta señal retardada, junto con la señal original del canal opuesto, son multiplicadas. Finalmente, la salida

será el resultado de dicha operación sobre estas dos señales, correspondiente a cada canal. Esto queda reflejado en la siguiente expresión:

$$R(t) = A_2(t)B_1(t) - A_1(t)B_2(t)$$

Según esta expresión, el detector es perfectamente asimétrico, como decíamos al principio. Debido a esto, es capaz de distinguir direcciones. Para que sea práctico y se pueda usar como detector de movimiento, se disponen en una matriz, en el que cada canal está a una distancia fija del siguiente canal. Estos detectores se disponen consecutivamente. La figura de abajo se muestra la estructura en una dimensión espacial:

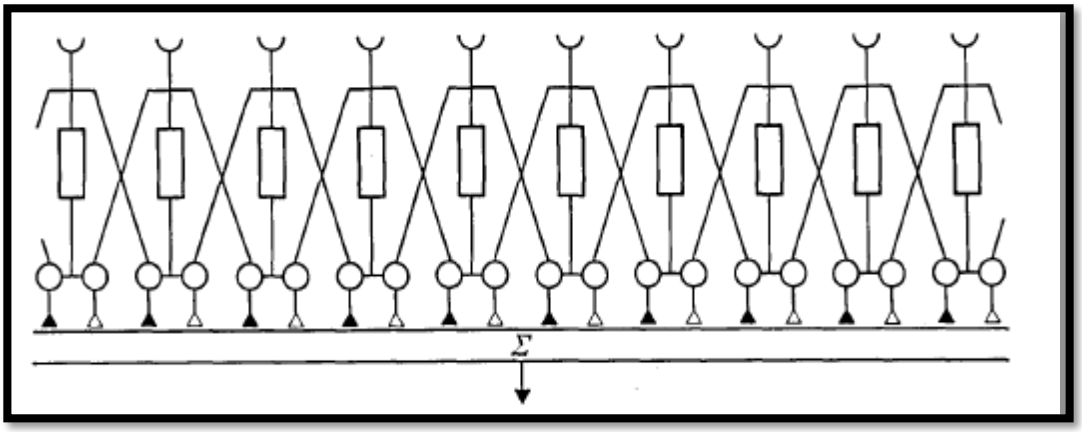

Ilustración 27: Estructuras de detectores de movimiento[14]

Como se puede observar en esta estructura, cada canal está conectado al siguiente, mostrando la estructura del detector elemental, y de esta manera el movimiento es medido en cada elemento de la imagen, en este caso, cada canal de entrada.

5.3 Detector de movimiento de cuatro cuadrantes

Este detector tiene un rango de detección de la velocidad limitado, porque se basa en un retraso lineal, y por lo tanto todas las correlaciones entre eventos que no se den en el tiempo de respuesta del filtro quedan descartadas.

Además del problema del rango dinámico, el detector tiene otro problema importante: sólo considera eventos del mismo tipo y omite los demás.

En un estudio [16] se muestra como en la retina de las moscas se producen eventos de tipo ON y OFF dependiendo de la variación de la luminancia y del signo de dicha variación. Las células de la retina se despolarizan o se polarizan según el tipo de evento.

La consecuencia inmediata de este hecho es que el detector de movimiento se debe modificar para que pueda recibir todas las polaridades posibles de eventos, esto es, ON-ON, ON-OFF, OFF-ON y OFF-OFF.

Como resultado se tiene una arquitectura como la mostrada a continuación:

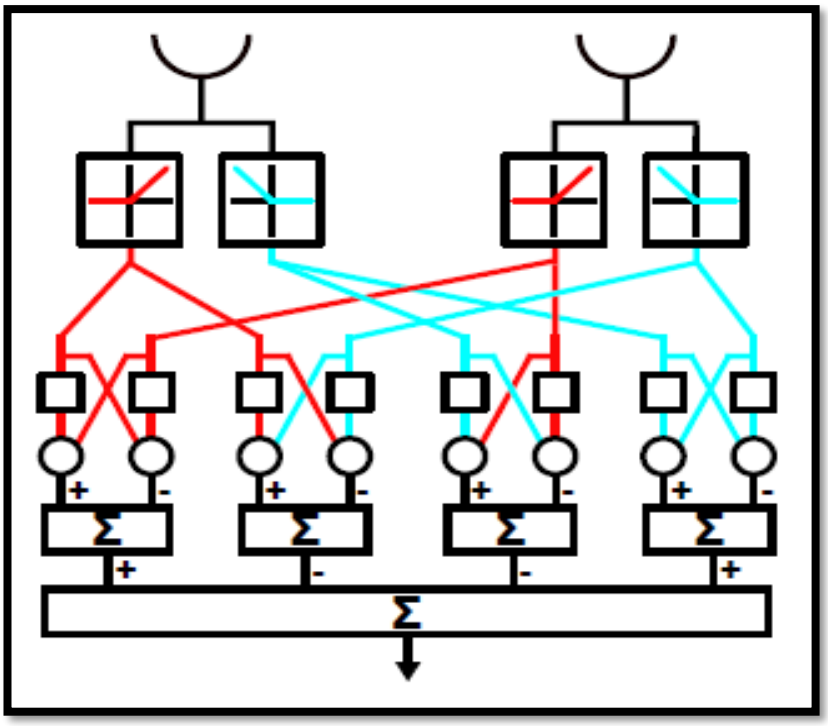

Ilustración 28: Detector de movimiento de cuatro cuadrantes [16].

Como se puede observar en la imagen, este detector está compuesto de cuatro detectores, en el que cada uno es sensible a un par de tipo de eventos (ON ó OFF). El último elemento es un combinador lineal, cuya función en este detector es computar una función lineal, cuyas entradas son las salidas de cada detector. Tiene la expresión siguiente:

$$salida = f(en1, en2, en3, en4) = c1 * en1 + c2 * en2 + c3 * en3 + c4 * en4$$

Los coeficientes c1, c2, c3, c4, deben de ser obtenidos por algún método para ajustar que la respuesta de la salida sea lo más lineal posible. Una representación gráfica de este proceso lo tenemos en [17]:

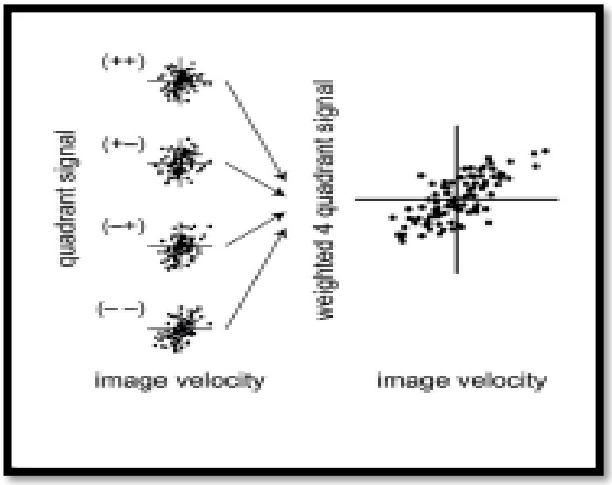

Ilustración 29: Representación gráfica del ajuste de pesos en el combinador lineal [17].

6 MODELO DE NEURONAS PULSANTES

Para la realización de un modelo más exacto del detector de movimiento, la multiplicación ha de sustituirse por una neurona que, en este caso, calcula cuanto tiempo transcurre desde que un canal recoge un evento, y el canal consecutivo recoge el mismo evento. De hecho, este nuevo detector basado en [19], tiene más rango dinámico, debido a que ya no realiza la correlación mediante la multiplicación sino con una neurona que es sensible a la diferencia temporal de los dos pulsos.
A continuación, se van a describir tres modelos principales de neuronas pulsantes, ya que son la base de este nuevo modelo.

6.1 Modelo Integrate&Fire.

Este modelo se basa en la capacidad que tiene la membrana para cambiar la tensión cuando le llega una señal de otra neurona vecina. Este modelo se representa con en el siguiente esquema:

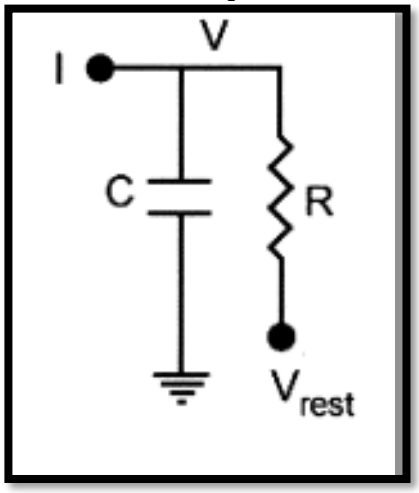

Ilustración 30: Modelo eléctrico de una neurona integrate and fire

Como se puede observar en la figura de arriba, este modelo está formado por un condensador y una resistencia y un potencial V_{REST} que cuando la tensión de la membrana umbral llega a un determinado valor V_{th}, entonces la tensión cae, reiniciando la tensión de la membrana. Su ecuación es la siguiente:

$$C_m \frac{dV}{dt} = I$$

Si V>V_{th} , entonces V_{REST}=0 y reinicia la membrana.

6.2 Modelo Leakyintegrate and fire

Este modelo es un poco diferente al anterior, debido a que también se considera el efecto del olvido de la neurona. Este efecto se modela mediante una resistencia en serie con el condensador que representa a la membrana. El circuito es el siguiente:

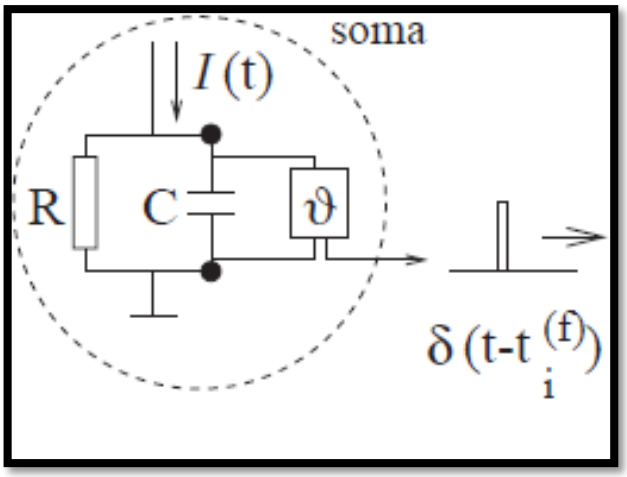

Ilustración 31: Circuito equivalente de la neurona leakyintegrate and fire [20].

Este modelo tiene la siguiente fórmula:

$$\tau_m \frac{du}{dt} = R_m I - u$$

Siendo τ_m la constante de tiempo del "leaky integrator" y $R_m$la resistencia de la membrana. El disparo en este tipo de modelo no está descrito explícitamente. En su lugar se usa el "tiempo de disparo, que se extrae de la expresión:

$$t^{(f)}: u\left(t^{(f)}\right) = \vartheta$$

Siendo ϑ el umbral de disparo de la neurona. Este $t^{(f)}$ se usa en una delta de Dirac para expresar el pulso y después se pasa por un filtro de paso bajo en la sinapsis.

6.3 Modelo de sinapsis

Para interconectar cada neurona con otras neuronas vecinas se hace uso de un modelo simplificado de sinapsis, que transforma los pulsos de las neuronas de salida en impulsos de corrientes que servirán para estimular otras neuronas [20].
Esto se puede expresar de la siguiente manera:

$$I_i = \sum_j \omega_{ij} \sum_f \alpha\left(t - t_j^{(f)}\right)$$

siendo α ,

$$\alpha\left(t - t_j^{(f)}\right) = -g\left(t - t_j^{(f)}\right)\left[u_i(t) - E_{syn}\right]$$

7 PROPUESTA DE SOLUCIÓN

7.1 Idea principal

El presente proyecto tiene como objetivo desarrollar un algoritmo de flujo óptico que pueda ser implementado de una forma óptima en una FPGA o en otro soporte hardware. Se han descrito numerosos algoritmos para realizar una comparativa con el algoritmo propuesto. Este algoritmo está basado en las ideas de Hassentstein y Reichardt (1956), que propusieron un modelo de detector de movimiento, basado en los circuitos neuronales de las moscas Drophile.
También se consideranotras ideas de [19] donde se implementan modelos neuronales, generalizando también el modelo de detector de movimiento. Para la realización de este trabajo se usarán los siguientes elementos:

7.1.1 Cámara DVS

Es el sensor donde se originan los eventos. Èstos informan de los cambios de brillo detectados por el sensor y los transmiten al procesador para nuestra aplicación. Los eventos son de dos tipos, ya descrito anteriormente, ON en caso de un cambio positivo y OFF en el caso de un cambio negativo. Cada evento se da en forma de un vector que tiene información sobre sus coordenadas x e y, *timestamp* y polaridad. Estos eventos se pueden usar para extraer el movimiento de la imagen.

7.1.2 Detector de movimiento

Es la idea principal del proyecto. Se basa en la idea de que el movimiento produce eventos con un desfase, tanto espacial como temporal y podemos calcular la velocidad por métodos correlacionales. De esta forma, al existir un desfase entre los eventos que se producen, la correlación se lleva a cabo, primero pasando por un filtro de paso bajo que retrasa el evento y una multiplicación entre los 2 canales del detector lleva a cabo el último paso de la correlación.
En este trabajo, al tener un sensor en el que los eventos pueden tener dos polarizaciones, ON y OFF, debemos generalizar el detector para que la estimación de velocidad se beneficie de este hecho e intentar que la velocidad sea más exacta. Existen referencias en [16], sobre que los circuitos neuronales aceptan diferentes eventos, que los hacen más precisos a la hora de estimar la velocidad. En este proyecto se ha realizado una implementación de dicho tipo de detector, basado también en la idea de [17], de que el sumador final es un combinador lineal, que internamente tiene unos pesos, que hay que ajustar para conseguir una respuesta de velocidad lo más lineal posible.

7.1.3 Neuronas de pulsos

Este tipo de neuronas también es utilizado en este proyecto para llevar a cabo la correlación como en [19], ya que la neurona mide el tiempo de vuelo de cada evento, y obtenemos la diferencia de tiempos entre los dos eventos del objeto en movimiento. Para tal fin se ha decidido usar un modelo sencillo adaptado al marco basado en eventos, en el que la neurona sólo hace una resta entre dos *timestamp* de dos eventos separados una distancia d.
Tenemos entonces dos tipos de neuronas, sensibles a cada una de las direcciones en el eje considerado. De este modo obtenemos un "tiempo de vuelo" en cada uno de los detectores, sensible a cada par de eventos y al final se obtienen cuatro tiempos de vuelos de cada uno de los detectores.

7.1.4 Combinador lineal

Como se explicó anteriormente, como resultado de obtener cuatro tiempos de vuelo de los cuatro detectores, en el que cada uno de los tiempos corresponden a una medida de la velocidad de cada uno de los detectores expresada en unidades de tiempo. Pero la medida de cada uno de los detectores no es precisa ni tampoco es lineal, por lo que, para solucionar dicho problema, se combinan los diversos tiempos, cada tiempo de muestreo, para que el combinador obtenga los tiempos de cada uno de los

detectores y así obtener una medida precisa.
Una representación de este hecho se tiene en la siguiente imagen:

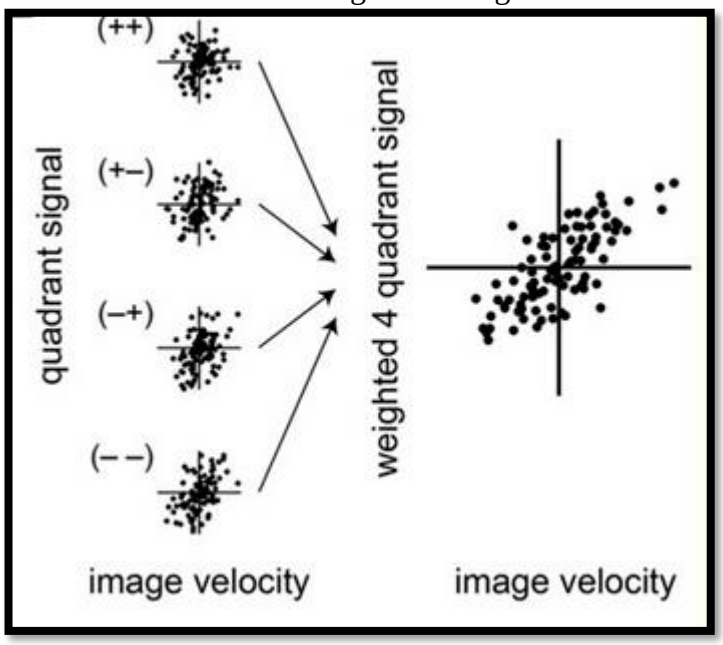

Ilustración 32: Representación de la operación del combinador lineal y como mejora la estimación de velocidad [17].

7.1.5 Funcionamiento del sistema

El sistema está pensado para funcionar a un tiempo de muestreo mayor que la tasa de muestreo de la cámara a fin de obtener los cuatro tiempos de vuelo de los detectores y realizar así la combinación de tiempos además de calcular la velocidad con más precisión. También se ha implementado un sistema de acumulación de eventos para mejorar el cálculo del flujo óptico, ya que al tener más eventos por *timestamp*, el algoritmo se comporta mejor y es más preciso.

8 IMPLEMENTACIÓN DE LA SOLUCIÓN

Como en todo proyecto de investigación, se ha desarrollado una simulación de las ideas propuestas para verificar si es posible la implementación hardware y, en caso afirmativo, poder comparar con algunas de las tecnologías existentes para poder sacar conclusiones para futuras aplicaciones. En este caso se va a describir el proceso de la implementación con la descripción del código de la solución que finalmente se ha adoptado. Se empezará con el primer modelo, explicando cómo se implementó el EMD, porque será el punto de partida de versiones posteriores.

8.1 EMD simple

En este modelo se usará la matriz de "eventTimes" que es una matriz de longitud 128x128 que registra el *timestamp* de cada evento. Esta es la base del funcionamiento del detector porque es una forma de registrar los eventos para su posterior procesamiento. El elemento fundamental aquí es el detector de movimiento en cada eje, cuyo flujo de programa es como sigue:

1. Recibir el paquete de eventos, y por cada evento:
2. Comparar el *timestamp* del paquete actual con eventTimes[x+d][y]
    1. Si es positivo, entonces calcular la diferencia en el sentido positivo del movimiento. Se compara con limites de tiempos y si entra dentro del rango se obtiene el tiempo, si no es así se obtiene un 0.
    2. Si es negativo o menor que una cierta cantidad, entonces el tiempo de vuelo es 0.
3. Comparar el *timestamp* del paquete actual, y también con eventTimes[x-d][y]
    1. Si es positivo, entonces calcular la diferencia en el sentido negativo del movimiento. Se compara con limites de tiempos y si entra dentro del rango se obtiene el tiempo, si no es así se obtiene un 0.
    2. Si es negativo o menor que una cierta cantidad, entonces el tiempo de vuelo es 0.

Este algoritmo es el mismo para ambos ejes, obteniéndose la velocidad en forma de tiempo tardado entre dos eventos de píxeles vecinos, pero dentro de unos límites para evitar tiempos de vuelos absurdos, en caso de eventos espurios.
Para solventar el problema, se propone un modelo que imita a las neuronas del córtex visual, muy parecido a este último, pero con la diferencia que la correlación se realiza en una neurona que también es sensible a la dirección del movimiento.

8.2 EMD con neuronas pulsantes como correlador

En este caso tenemos un algoritmo muy parecido al anterior, pero está simplificado, al no haber límites de rango dinámico, por lo que se aprovecha todo el potencial de los eventos. En esta implementación se ha elegido el modelo de "Exponencial LIF". Este modelo de neurona comparte las características del olvido de las "LIF", junto con el hecho de que si se produce un evento dentro del tiempo en que la neurona todavía recuerda el evento anterior, produce spikes a la salida dependiendo de la diferencia temporal entre ambos. Este tiempo tiene un máximo, y depende del olvido de la neurona.

En la siguiente imagen se representa el modelo de neurona pulsante:

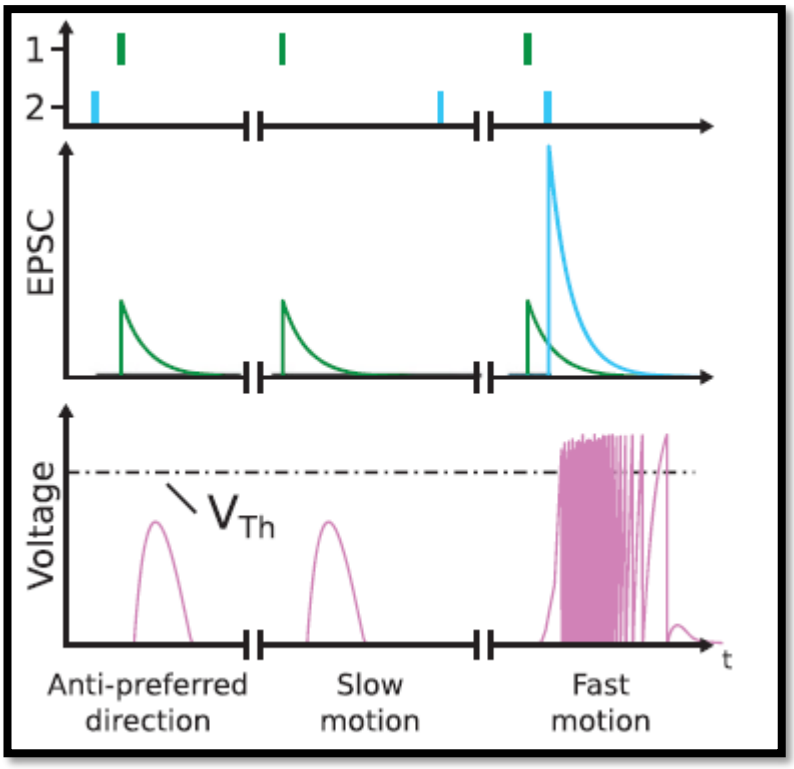

Ilustración 33: Gráfica del modelo de la neurona pulsante [19]

Como se puede ver en la imagen, en el caso de movimientos lentos y de movimientos en la dirección contraria al movimiento al que es sensible la neurona, no se produce spike de salida. Sin embargo, en el último caso en el que tenemos el evento 2 está a una distancia temporal en que la neurona puede disparar. Este disparo depende de la distancia temporal del evento, representado en la siguiente gráfica:

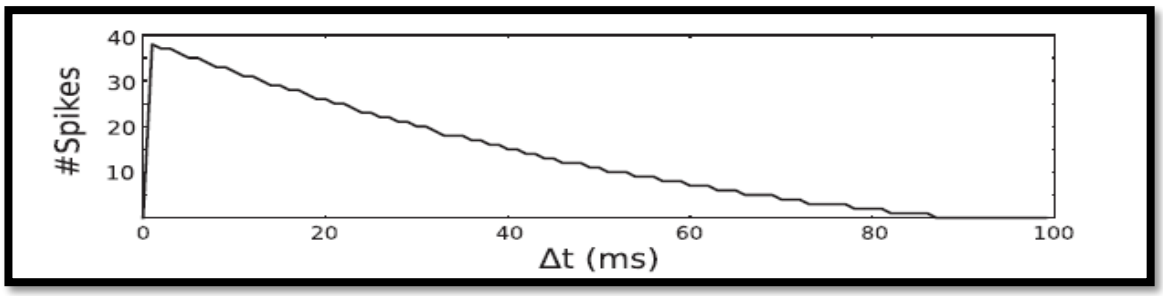

Ilustración 34: Gráfica de la frecuencia de disparo de la neurona [19].

Esta gráfica representa como la neurona dispara según el tiempo de vuelo de ambos eventos, por lo que se demuestra que la neurona correlaciona los dos eventos y puede llegar a obtener la diferencia temporal.
En el caso basado por eventos es más sencillo, ya que este tiempo es la diferencia entre *timestamp* de dos posiciones vecinas, y después se comprueba que está dentro del rango medible. Para ello se comprueba que es mayor de un cierto valor mínimo y es menor que un máximo, que será el tiempo de olvido de la neurona, por la que no podrá correlacionar los dos eventos.

8.3 EMD de cuatro cuadrantes

En este apartado se describirá cómo se ha planteado la versión del código del detector con los cuatro cuadrantes. Como se ha explicado anteriormente, este tipo de detectores pueden llegar a ser más efectivos que los detectores simples, debido a que son sensibles a todos los conjuntos de eventos

posibles. Este tipo de detector tiene la siguiente arquitectura:

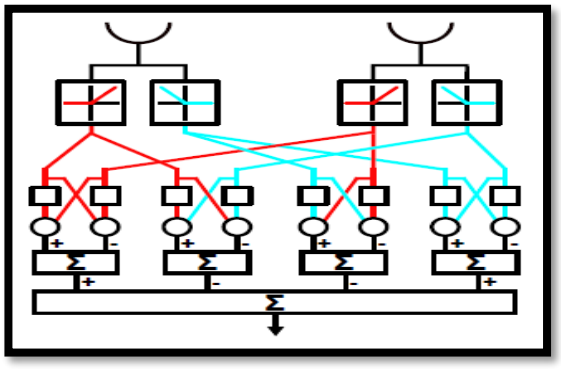

Ilustración 35:Arquitectura usada en esta versión del código. [16]

Cada detector se implementa por separado, mediante el algoritmo anterior, sensible a cada uno de pares de tipos de eventos. La diferencia en este caso estriba en el combinador lineal. Cada uno de los detectores calcula un tiempo de vuelo para cada par de tipo de eventos, que es la entrada del combinador. Internamente, el combinador realiza la siguiente operación:

$$salida = f(en1, en2, en3, en4) = c1 * en1 + c2 * en2 + c3 * en3 + c4 * en4$$

Los coeficientes c1, c2, c3, c4, deben ser obtenidos por algún método para ajustar que la respuesta de la salida sea lo más lineal posible, como se dijo anteriormente.

Por último, para implementar la acumulación de eventos, se ha procedido de la siguiente manera: a medida que llegan los eventos, se van guardando en la matriz "eventTimes", pero con un distinto *timestamp*. Este nuevo *timestamp* se calcula, teniendo en cuenta que cada paquete llega cada 1us. Lo que se hace en esta parte del algoritmo es recoger un número n de paquetes, pero guardando los eventos de cada bloque de paquetes considerando un mismo *timestamp*. De esta manera se puede realizar una acumulación de eventos y tener más eventos por *timestamp* para que el algoritmo funcione mejor.

9 IMPLEMENTACIÓN DEL FILTRO

9.1 Introducción

A continuación, se va a explicar en detalle cómo se ha implementado el filtro, y todas las soluciones que se han adoptado para implementar el código en Java en el *frame*work de jAER. Primero, se hará una introducción con algunas de las clases del código Java más importantes usadas en el código, para seguir con la implementación del código.

9.2 Librerías usadas en el trabajo

A continuación, se mostrará la relación de librerías que se ha usado en el jAER, explicando brevemente cada una de las funciones que se aportan en el código.

Definición de las clases utilizadas en el filtro	
Nombre de la clase	Definición
net.sf.jaer.chip.AEChip	Esta clase define la interfaz con los eventos y su renderizado en la GUI
net.sf.jaer.event.BasicEvent	Esta clase define la estructuras de los eventos
net.sf.jaer.event.EventPacket	Define la función de procesado de eventos en el filtro
net.sf.jaer.event.PolarityEvent	Define la estructura de datos de los eventos con polaridad
net.sf.jaer.eventprocessing.EventFilter2D	Función procesadora de eventos
com.jogamp.opengl.GL2	Clase derivada de la librería de OpenGL para ser usada en jAER
com.jogamp.opengl.GLAutoDrawable	Clase de OpenGL
net.sf.jaer.graphics.*Frame*Annotater	Esta clase contiene la función "annotate" para dibujar resultados del filtro
java.lang.Math	Clase nativa de Java, que contiene funciones matemáticas.
java.util.ArrayList	Clase que emula un vector de objetos, de dimensión dinámica.

9.3 Clases creadas en el proyecto

Para implementar el filtro, hace falta generar tipos de datos específicos para la aplicación. Es el caso de "Time_class". Esta clase contiene sólo dos miembros que son "time" y "*timestamp*". Sirve para registrar el tiempo de vuelo de cada uno de los detectores en cada uno de los *timestamp*s.
Para almacenar los resultados del filtro, se usa otra clase llamada "flecha" que contiene los siguientes atributos:

- float time[]: Tiempo de vuelo conseguido en cada detector.
- int x : Coordenada x

- int y: coordenada y
- int direction[]: vector de dirección para los dos ejes.
- int distance: variable que guarda la distancia de correlación.

En esta clase se encuentran los siguientes miembros:

- float[] get_time(): Obtiene el tiempo de vuelo en los dos ejes.
- int get_x(): Obtiene la coordenada x
- int get_y(): Obtiene la coordenada y.
- int[] get_direction(): Obtiene la dirección.
- int get_distance(): Obtiene la distancia.
- floatget_scale(): Obtiene la escala.
- set_time(float[] time_en): Carga la variable interna "time"
- set_x(int x_en):Carga la variable interna "x"
- set_y(int y_en):Carga la variable interna "y"
- set_direction(int[] direction_en)Carga la variable interna "direction"
- set_distance(int distance_en) Carga la variable interna "distance"
- set_scale(floatscale_en) Carga la variable interna "scale"

Esta clase ha sido desarrollada para que el método *filterPacket* pueda comunicar eficientementelos resultado del filtro al método *annotate*. Cuando el método del filtro "Eventfilter" procesa el paquete de eventos, los resultados del cálculo son guardados en un objeto de la clase "flecha", para que el método "annotate" pueda dibujar los resultados correctamente en la pantalla.

9.4 Estructura de un filtro en jAER

En jAER existe la posibilidad de implementar filtros propios y además usar todas sus librerías disponibles en el proyectopara las que, al ser de software libre, se tiene acceso a todo el código. Ahora se presentará la estructura de un filtro en jAER, y es la que se seguirá en apartados posteriores para describir la implementación realizada en este trabajo. La estructura de todo filtro en jAER es la siguiente:

- Importar la subclase *net.sf.jear.eventprocessing.EventFilter2D*
- *Como* se está extendiendo la clase *EventFilter2D* también se extenderá la clase EventFilter , por lo que hay que sobrescribir los siguientes métodos:
    - filterPacket(EventPacket)
    - resetFilter()
    - initFilter()
- Añadir propiedades que pueden ser mostradas en la GUI del filtro.
- Documentar las propiedades del filtro haciendo uso del método *setPropertyTooltip(StringgroupName, StringparameterName, Stringtip).*
- Para ejecutar el filtro desde el jAER, hacer losiguiente:*"View" -> "Event filtering" -> "Filters",* yelegir el nombre de nuestro filtro.

9.5 Renderizado de resultados

Como en el filtro que se desarrolla en este trabajo se usan gráficos de openGL para mostrar resultados, se va a explicar brevemente los métodos y las clases usadas para dicho fin.
Para este propósito, el método que tiene jAER es void annotate(). Para usar dicho método se debe llamar a la interfaz *FrameAnnotator.*

9.6 Añadir propiedades al filtro

Para la modificación de las propiedades del filtro en tiempo real, jAER tiene la posibilidad de crear una GUI , mediante javabeans, implementando los métodos get/set para cada campo de este GUI.

Se puede añadir un textbox y una explicación para éste. Un ejemplo del código se indica a continuación:

private int corrTimeMax = getPrefs().getInt("EMDMotionCorrelator.corrTimeMax", 1000);

Como se puede observar, en este ejemplo la variable "corrTimeMax " queda definida por el método de entrada getInt , ya que es de tipo int. Y los métodos set/get se definen como:

Código de los métodos set y get

```
public int getCorrTimeMax(){
return corrTimeMax;
}
Publicvoid setCorrTimeMax(int corrTime){
this.corrTimeMax = corrTime;
getPrefs().putInt("EMDMotionCorrelator.corrTimeMax", corrTime);
}
```

9.7 Explicación del código del filtro

A continuación, se van a explicar las partes que componen el código del filtro. El núcleo del filtro está implementado en el método "publicEventPacket<?>filterPacket(EventPacket<?> in)", que es el encargado de recibir los paquetes de eventos y por consiguiente, del procesado del filtro. También nos centraremos en el método "annotate(GLDraw)" que será el método encargado de pintar los resultados de los cálculos del filtro. También se va a comentar los diversos controles del GUI del filtro y que significan cada uno de ellos con diversos ejemplos que clarificaran su uso. Se empezará explicando el núcleo del filtro, que se tratará a continuación:

```
void checkArrays()
```

Este método está definido como:

Código de checkArrays()

```
Privatevoid checkArrays(){if(eventTimes ==null){
eventTimes      =newint[chip.getSizeX()][chip.getSizeY()][2];//      0.-
timestamp,1.-polarity
}
if(time_flechas==null){
time_flechas=newfloat[2];
}
if(sentido==null){
sentido=newint[2];
}
if(time_x_ON_ON==null){
```

```
time_x_ON_ON=new time_class[chip.getSizeX()][chip.getSizeY()];
}
if(time_x_OFF_OFF==null){
time_x_OFF_OFF=new time_class[chip.getSizeX()][chip.getSizeY()];
}
if(time_x_OFF_ON==null){
time_x_OFF_ON=new time_class[chip.getSizeX()][chip.getSizeY()];
}
if(time_x_ON_OFF==null){
time_x_ON_OFF=new time_class[chip.getSizeX()][chip.getSizeY()];
}
if(time_y_ON_ON==null){
time_y_ON_ON=new time_class[chip.getSizeX()][chip.getSizeY()];
}
if(time_y_OFF_OFF==null){
time_y_OFF_OFF=new time_class[chip.getSizeX()][chip.getSizeY()];
}
if(time_y_OFF_ON==null){
time_y_OFF_ON=new time_class[chip.getSizeX()][chip.getSizeY()];
}
if(time_y_ON_OFF==null){
time_y_ON_OFF=new time_class[chip.getSizeX()][chip.getSizeY()];
for(int i=0;i<chip.getSizeX();i++){
for(int j=0;j<chip.getSizeY();j++){
time_x_ON_ON[i][j]=new time_class();
time_y_ON_ON[i][j]=new time_class();
time_x_ON_OFF[i][j]=new time_class();
time_y_ON_OFF[i][j]=new time_class();
time_x_OFF_ON[i][j]=new time_class();
time_y_OFF_ON[i][j]=new time_class();
time_x_OFF_OFF[i][j]=new time_class();
time_y_OFF_OFF[i][j]=new time_class();
}
}
}
if(timestamp_total==null){
timestamp_total=new int[8];
}
}
    }
```

Como se puede observar en el código, esta función se usa para inicializar las estructuras de datos

usadas en el filtro, para después ser usadas más adelante para registrar los tiempos de vuelo y el *timestamp*, usado para registrar los eventos que van llegando de los paquetes de entrada.

En cuanto a la estructura que registra los tiempos de vuelos, tiene como estructura: time_[x|y]_[ON_ON|ON_OFF|OFF_ON|OFF_OFF]. Esto quiere decir que se distingue entre cada uno de los ejes y también en cada uno de los pares de tipo de eventos que se pueden originar.

A continuación, está el bucle que itera todos los eventos del paquete:

```
for(Object o : in){
```

Como se puede observar, este bucle for no tiene contador, en su lugar, itera cada elemento de una lista que se pasa al evento de tipo "eventfilter" e introduce cada evento del paquete en la variable "o" de tipo Object.

Se hace necesario entonces, cambiar a un tipo usable definido en jAER, esto aparece en la siguiente línea:

```
PolarityEvent e =(PolarityEvent) o;
```

Una vez que se ha convertido el tipo de datos de los eventos para ser usado en el filtro, se pasa ahora al siguiente paso del filtro:

```
if(e.x>10&& e.x<110){// Para limitar en x la ventana
if(e.y>10&& e.y<110){//Lo mismo para las y
```

Este fragmento de código limita la zona de los eventos. Esto es debido a que el algoritmo recoge el *timestamp* de los eventos vecinos en la matriz "eventTimes" y si se considera todo el cuadro, habría error de acceso de matriz, porque se accedería a índices inexistentes en la matriz.

```
int oldtsr_x = eventTimes[e.x + corrDistance][e.y][0];
int oldtsl_x = eventTimes[e.x - corrDistance][e.y][0];
int oldpol_r=eventTimes[e.x + corrDistance][e.y][1];
int oldpol_l=eventTimes[e.x - corrDistance][e.y][1];
```

Este fragmento de código se encarga de extraer tanto los *timestamp* de las coordenadas vecinas al evento actual, como la polaridad de dichos eventos. Estos valores serán procesados en etapas futura del filtro.

El cálculo del tiempo de vuelo se realiza en este fragmento:

```
int mr = timestamp - oldtsr_x;
int ml = timestamp - oldtsl_x;
```

Una vez obtenido los tiempos de vuelos en las dos direcciones, entramos en un bloque if-then para decidir si el movimiento va hacia la derecha o hacia la izquierda. Esto se consigue mediante el siguiente fragmento de código:

```
if(ml < corrTimeMin && mr > corrTimeMin){
```

y en el caso de que el movimiento vaya hacia la izquierda, se implementa otro bloque que se activa sólo en ese caso:

```
if(mr < corrTimeMin && ml > corrTimeMin){
```

que es el caso del movimiento hacia la izquierda.

Ahora se va a describir el bloque del código del detector completo. A continuación, se explicará para un caso en concreto y se generalizará para todos los pares de eventos y para todas las direcciones posibles y en los dos ejes. Empecemos explicando para el caso del detector para el par de eventos On-On:

Detector ON-ON

```
if(oldpol_r==1&& e.polarity==PolarityEvent.Polarity.On){
      if(mr<=corrTimeMax){

            y_p=e.y;
            x_p=e.x;
            time_x_ON_ON[e.x][e.y].time_res=mr;
            time_x_ON_ON[e.x][e.y].timestamp=timestamp;
            sentido[0]=0;
      }else{
            time_x_ON_ON[e.x][e.y].time_res=0;
            time_x_ON_ON[e.x][e.y].timestamp=timestamp;
      }

}
```

En este bloque de código se pueden apreciar varias partes:

- Bloque if para seleccionar el tipo de eventos a que es sensible este detector
- Bloque if que limita para un tiempo máximo de vuelo
- Bloque de asignación donde se asignan las variables a las estructuras de datos concretas.
- Bloque else, para el caso que se exceda el tiempo máximo de vuelo, esto va acorde al modelo de neurona.

Cada uno de los detectores se dirige a una estructura de datos que registra el tiempo de vuelo en cada detector, en cada eje. Esta estructura registra dos valores:

- El tiempo de vuelo en sí, como time_res
- El *timestamp* en que se ha producido el evento actual

Este *timestamp* es usado después para indicar al "combinador lineal" cuando debe de procesar los datos llegados.
Dependiendo del tipo de eventos que se adquiera, la línea:

```
if(oldpol_r==1&& e.polarity==PolarityEvent.Polarity.On){
```

cambiará de valores, siendo el 1 o -1 en el caso de On o Off, debido a la forma que se almacenan los eventos en la matriz de "eventTimes".

Según el sentido tenemos que considera mr o ml en el caso de las x, y en el caso de las y, md y mu, dependiendo de si el movimiento se dirige hacia abajo o hacia arriba.

Ahora va a tratarse un punto clave del filtro: el cambio del *timestamp* a otro intervalo de tiempo. Con esto nos aseguramos de que se acumule los eventos cada intervalo de tiempo definido por el usuario:

Código de acumulación de eventos

```
if(e.timestamp-oldtimestamp<0){
            timestamp=0;
            oldtimestamp=0;
      }
                  if(e.timestamp-oldtimestamp>=accumulate_time){
            oldtimestamp=e.timestamp;
            timestamp++;
      }
eventTimes[e.x][e.y][0]= timestamp;
```

En este código se observa primeramente que si el *timestamp* es más pequeño que el que había en la iteración anterior quiere decir que se ha rebobinado el archivo o que se ha reiniciado el contador de *timestamp* de la DVS128, y por lo tanto se reinicia el contador que controla el número de paquetes de

eventos.
En la segunda parte de este código se realiza la conversión de intervalo del *timestamp*, y por tanto de la acumulación de eventos en la matriz *eventTimes*, ya que cambió el *timestamp* de dichos eventos.
El último paso es la integración de los tiempos y el cálculo de vuelos además del sentido del movimiento, obtenidos en el combinador lineal. Este es implementado de la siguiente manera:

Código del combinador

```
timestamp_total[0]=timestamp-time_y_ON_ON[e.x][e.y].timestamp;
timestamp_total[1]=timestamp-time_y_OFF_ON[e.x][e.y].timestamp;
timestamp_total[2]=timestamp-time_y_ON_OFF[e.x][e.y].timestamp;
timestamp_total[3]=timestamp-time_y_OFF_OFF[e.x][e.y].timestamp;
timestamp_total[4]=timestamp-time_x_ON_ON[e.x][e.y].timestamp;
timestamp_total[5]=timestamp-time_x_OFF_ON[e.x][e.y].timestamp;
timestamp_total[6]=timestamp-time_x_ON_OFF[e.x][e.y].timestamp;
timestamp_total[7]=timestamp-time_x_OFF_OFF[e.x][e.y].timestamp;
int maxValue = timestamp_total[0];
for(int i=1;i < timestamp_total.length;i++){
      if(timestamp_total[i]> maxValue){
            maxValue = timestamp_total[i];
                  }
      }
if(maxValue>thresold_time){
time_y=w_ON_ON*(float)time_y_ON_ON[e.x][e.y].time_res+w_OFF_OFF*(float)time_y
_OFF_OFF[e.x][e.y].time_res+w_ON_OFF*(float)time_y_ON_OFF[e.x][e.y].time_res+
w_OFF_ON*(float)time_y_OFF_ON[e.x][e.y].time_res;
time_x=w_ON_ON*(float)time_x_ON_ON[e.x][e.y].time_res+w_OFF_OFF*(float)time_x
_OFF_OFF[e.x][e.y].time_res+w_ON_OFF*(float)time_x_ON_OFF[e.x][e.y].time_res+
w_OFF_ON*(float)time_x_OFF_ON[e.x][e.y].time_res;
time_flechas[0]=time_x;
time_flechas[1]=time_y;
flechas.add(new flecha(time_flechas,sentido,x_p,y_p,scaler,corrDistance));
}
```

Como se puede observar, se extraen los *timestamp* que ha registrado cada detector en cada eje, y se calcula cual ha sido su máximo valor. Este valor se usa en el siguiente código como comparación con el límite del intervalo de cálculo:

```
int maxValue = timestamp_total[0];
```

y después se introduce el resultado en objeto de la clase "flecha":

```
flechas.add(new flecha(time_flechas,sentido,x_p,y_p,scaler,corrDistance));
```

Por último, se introduce los *timestamp* nuevos en la matriz "eventtimes" junto a la polaridad:

Código que introduce la polaridad

```
if(e.polarity==PolarityEvent.Polarity.On){
          eventTimes[e.x][e.y][1]=1;
      }
      if(e.polarity==PolarityEvent.Polarity.Off){
            eventTimes[e.x][e.y][1]=-1;
      }
}
```

Ahora se va a proceder a describir el método annotate, que es el encargado de pintar en la pantalla los resultados del filtro.

```
GL2 gl=drawable.getGL().getGL2();
```

En esta parte inicializamos las variables temporales para el eje x y el eje y, y además, inicializamos

las estructura de OpenGL.

Código para extraer el tiempo de vuelo

```
if(flechas.get(ii).get_direction()[0]==0){
  time_tmp_x=flechas.get(ii).get_time()[0];
}
if(flechas.get(ii).get_direction()[0]==1){
       time_tmp_x=-flechas.get(ii).get_time()[0];
}
if(flechas.get(ii).get_direction()[1]==0){
      time_tmp_y=-flechas.get(ii).get_time()[1];
}
if(flechas.get(ii).get_direction()[1]==1){
      time_tmp_y= flechas.get(ii).get_time()[1];
}
```

Esta estructura if, realiza el cálculo del sentido. Es aplicable a las dos direcciones. Una vez descritaslas partes principales del filtro de flujo óptico, se va a proceder a las pruebas y a comparación con distintos algoritmos de flujo óptico ya implementados en jAER.

9.8 GUI del filtro y captura del filtro en funcionamiento

Ahora se mostrarán los distintos controles del filtro implementado y se explicarán brevemente, además de realizar una captura de pantalla con el filtro funcionando.

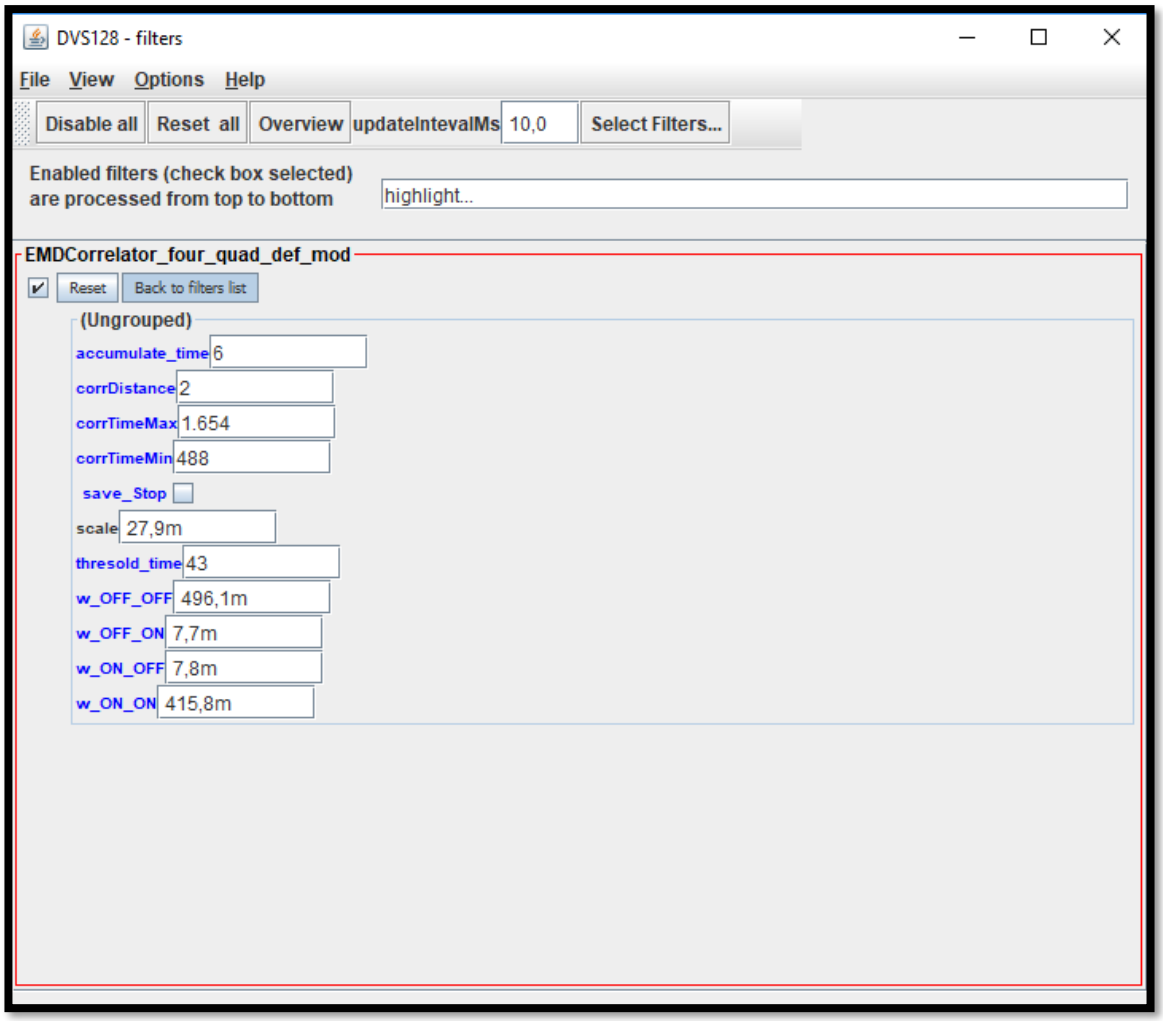

Ilustración 36: Ventana del filtro

Ahora se explicará cada uno de los controles:

- Accumulate time: Con este control se varía el tiempo de acumulación para un mejor rendimiento del filtro.
- corrDistance: Distancia de correlación. Es la distancia (en píxeles) que considera el detector para calcular la velocidad.
- corrTimeMax: El tiempo máximo de vuelo que detecta la neurona pulsante del detector.
- corrTimeMin: El tiempo mínimo de vuelo que detecta la neurona.
- Check Save-Stop: Este control sirve para grabar los datos en un fichero, con nombre predefinido en el código.
- Thresold_time: Es el intervalo de tiempo con que se ejecuta el combinador, ya que al ser tiempos de vuelos que les llegan alos distintos filtros, tiene que esperar hasta que la mayoría de los detectores hayan producido un resultado.
- w_OFF_OFF, w_OFF_ON, w_ON_OFF y w_ON_ON: Pesos del combinador.

Ahora, amodo de ejemplo se muestra una ventana del filtro:

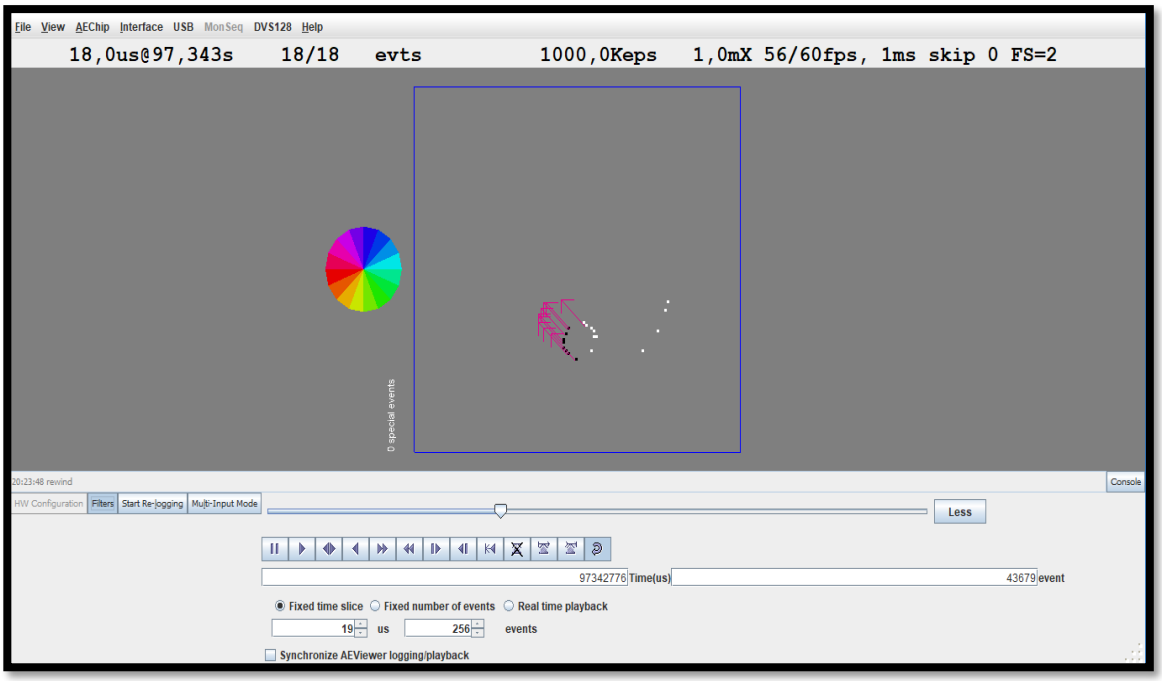

Ilustración 37: Ventana del filtro.

10 RESULTADOS DE LOS FILTROS

10.1 Introducción

Para medir los resultados de los filtros y tener una métrica para comparar entre distintos algoritmos disponibles, el cálculo del flujo óptico dispone de distintas métricas. Para desarrollar dicha métrica, antes nos hace falta elegir un dataset que tenga una serie de criterios para que puedan ser fácilmente evaluados, en torno al flujo óptico. Estos son:

- Que sea un movimiento fácilmente reconocible.
- Que la velocidad del objeto sea constante.
- Que el cálculo de las métricas requeridas no sea demasiado complejo.

Con estos requerimientos, sea elegido el dataset "fastdot", que se muestra en la siguiente figura:

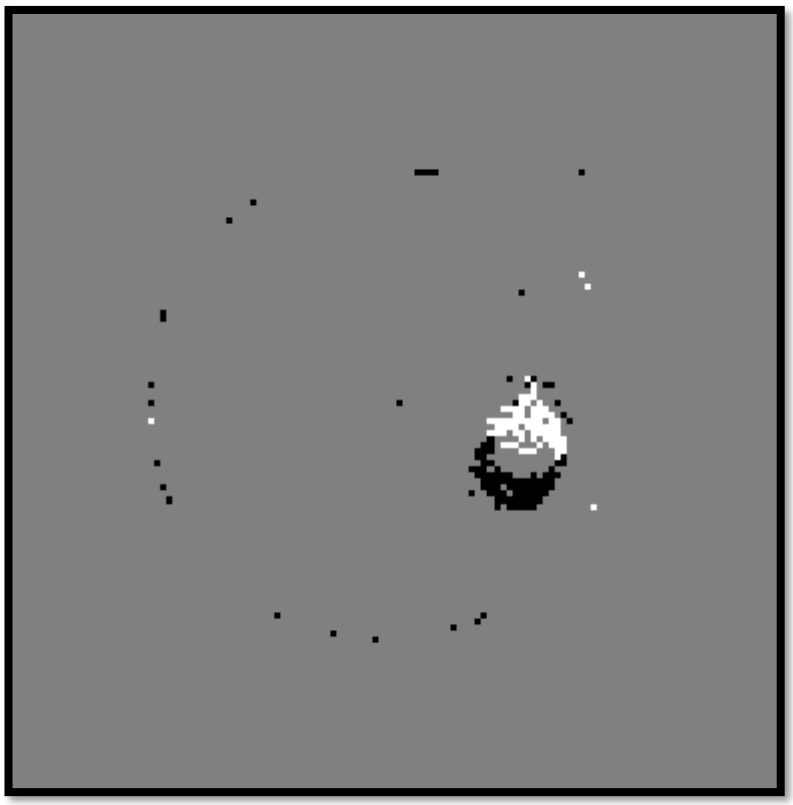

Ilustración 38: Captura de la pantalla de eventos del jAER con el fastDot.

Como se puede observar en la figura el movimiento es circular, además la velocidad del punto es conocida y constante porque esrealizado con un sistema controlando la velocidad angular:

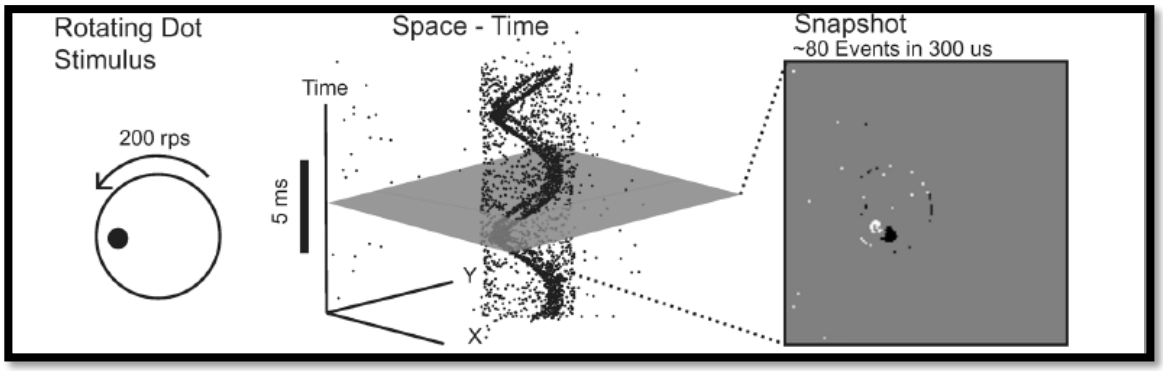

Ilustración 39:Dataset fastDot [1].

Esto quiere decir que tenemos un patrón de movimientos que podemos usarlo para generar métricas y comparar distintos algoritmos de flujo óptico.

10.2 Fundamentos teóricos de las pruebas

10.2.1 Métrica de movimiento

Para la realización de las pruebas, primero se debe medir la velocidad del objeto. Como en este caso es un círculo que se mueve a velocidad constante, podemos obtenerla velocidad en cada punto de la trayectoria para sus diferentes distancias al centro.
El primer paso que se ha realizado es recoger de cada uno de los filtros los datos de velocidad que se han obtenido mediante un fichero .txt. De este fichero hantomadolos puntos donde el filtro ha calculado una velocidad del flujo óptico. Entonces se representan todos esos puntos obteniendose los siguientes resultados de la representación:

Ilustración 40: Nubes de puntos de los puntos donde la velocidad es calculada

Como se puede observar, en la imagen se puede distinguir el centro la zona de movimiento del círculo, y el aro externo. Como primer paso se ha calculado el centro del movimiento en la imagen que está en las coordenadas x e y:(62,62). También se puede extraer la distancia del radio de la circunferencia. Los valores de las distancias van desde 17 a 33 píxeles, que será nuestro rango para calcular cada una de las velocidades extraída de los datos del filtro. Como es un movimiento circular, la velocidad en este caso tiene un perfil como en la siguiente figura:

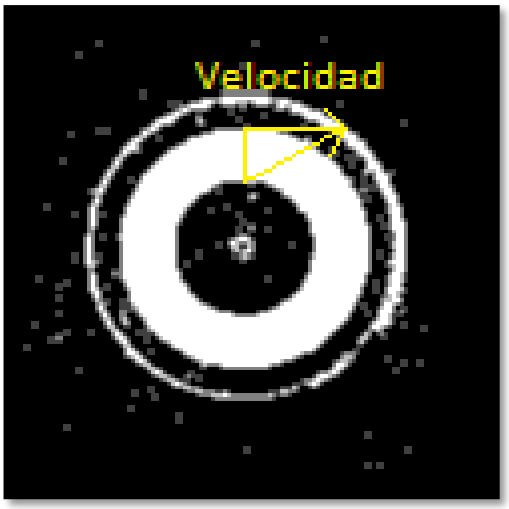

Ilustración 41:Perfil de velocidades dado en el movimiento del círculo.

Este perfil también será parte de la métrica de comparación del filtro, ya que se verá como cada uno es capaz de adaptarse a dicho perfil y el que más lineal sea, será el que mejor puntuación tenga en esta parte.
Por otra parte, también se puede calcular el ángulo en que se sitúa el círculo en su trayectoria, comparándolo con el predicho por la velocidad. Esto también se hará para todo el intervalo de distancias posibles y se mostrará una gráfica comparativa de los tres filtros considerados. Para realizar dicho cálculo del ángulo, se ha elaborado el siguiente desarrollo:

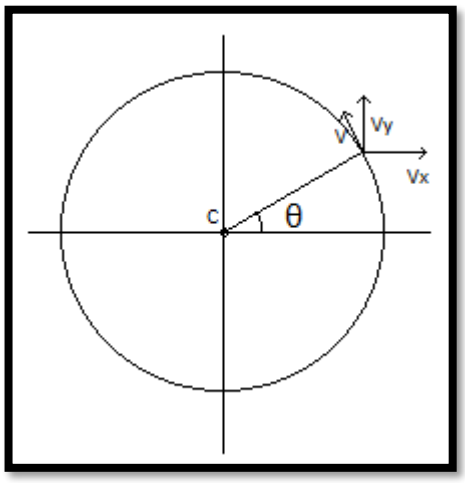

Ilustración 42: Diagrama para el cálculo del ángulo mediante la velocidad.

Para estimar el error en el ángulo estimado por la velocidad que proporciona el filtro, primero hay que calcular el ángulo según las coordenadas del punto donde ha realizado dicho cálculo. Entonces se calcula el ángulo basado en la velocidad del siguiente modo:

$$\theta_{vel} = \tan^{-1}\frac{V_x}{V_y}$$

Para estimar el ángulo de la trayectoria se usa esta ecuación:

$$\theta_{cor} = \tan^{-1}\frac{Y - Y_c}{X - X_c}$$

Entonces el error del ángulo es:

$$e_\theta = |\theta_{vel} - \theta_{cor}|$$

Y por último se hace una media del error que ha resultado para esa distancia al centro y también una media de todas las velocidades calculadas en esa distancia al centro; al tener un movimiento circular uniforme, la velocidad debe ser la misma en todos los puntos de la trayectoria.
Seguidamente, se representa una gráfica con los valores de velocidades y otra con los de errores de ángulo en función de la distancia del centro. Con esto se consigue ver cuánto se acerca el perfil calculado mediante los datos del filtro al perfil de velocidad teórico y cuanto error de ángulo se obtiene de las pruebas.
Ahora se va a explicar el código en Python usado para procesar las muestras de velocidades:

10.2.1.1 Código en Python

Aquí está el código usado para obtenerlas métricas de movimiento:

Código de la métrica del movimiento

```
import numpy as np
import math as mat
import matplotlib.pyplot as plt
import statistics
f=open('test_patchflow.txt', newline='\n')
data_file = csv.reader(f, delimiter=' ', quotechar='|')
line=0
x=[]
y=[]
vx=[]
vy=[]
temp=[]
for row in data_file:
if line<=2:# Se salta de la primera línea ya que es la que tiene
# los titulos de cada columna
pass
else:
# Este bloque se guardan los datos ordenados en cada variable
# de posición y velocidad en cada eje.
x.append(int(row[1]))
y.append(int(row[2]))
vx.append(float(row[4].replace(',','.')))
vy.append(float(row[5].replace(',','.')))

        line+=1


x_new=[]
y_new=[]
vx_new=[]
vy_new=[]
```

```
vt=[]
errores_angle=[]
vels=[]
for r in range(15,33):

for i in range(len(x)):
r_a=mat.sqrt(mat.pow((x[i]-62),2)+mat.pow((y[i]-62),2))
if r_a==r:
# Calcula la velocidad y el error de ángulo para cada distancia r
# y este bloque extrae todos los puntos, cuya distancia al centro es
#r
x_new.append(x[i])
y_new.append(y[i])
vx_new.append(vx[i])
vy_new.append(vy[i])
#Eliminamos los elementos repetidos
x_ant=0
y_ant=0
x_des=[]
y_des=[]
vx_des=[]
vy_des=[]
for i in range(len(x_new)):
r_p=mat.pow((x_new[i]-x_ant),2)+mat.pow((y_new[i]-y_ant),2)
if r_p>0:
x_des.append(x_new[i])
y_des.append(y_new[i])
vx_des.append(vx_new[i])
vy_des.append(vy_new[i])
x_ant=x_new[i]
y_ant=y_new[i]
xyv=[]
for i in range(len(x_des)):
xyv.append([x_des[i],y_des[i],vx_des[i],vy_des[i]])# metemos en un
# vector que sea [x,y,vx,vy]
res =[]
for i in xyv:
if i notin res:#quitamosrepetidos
res.append(i)
xc=62
```

```
yc=62#Coordenadas del centro
angle=[]
error_angle=[]
for i in range(len(res)):
#calculamos velocidad y los ángulos
vt.append(mat.sqrt(mat.pow(res[i][2],2)+mat.pow(res[i][3],2)))
angle.append([(180/mat.pi)*mat.atan2(res[i][1]-yc,res[i][0]-
xc),(180/mat.pi)*mat.atan2(res[i][2],-res[i][3])])
f.close()
for i in range(len(angle)):
error_angle.append(abs(angle[i][0]-angle[i][1]))
ifnot vt or(len(vt)<2):
continue
else:
vels.append([r,statistics.mean(vt),statistics.stdev(vt)])
ifnot error_angle or(len(error_angle)<2):
continue
else:errores_angle.append([r,statistics.mean(error_angle),statistics.stdev(error
_angle)])
```

10.2.2 Métrica de procesamiento

En esta métrica se va a extraer, a partir de los datos de procesamiento proporcionado por la herramienta jAER las velocidades de los distintos algoritmos. Se van a considerar dos valores: Tiempo de procesamiento por evento y eventos procesados por segundo. Una vez extraídos los valores del filtro, se van a comparar con diversos resultados de otros filtros y obtener una tabla comparativa de los diversos filtros.

10.2.2.1 Código en python

Código para la métrica de procesamiento

```
from parse import*
file = open("Res_fil_patch.txt","r")
contador=0
tiempo=0;
numero=0
num=0
for line in file:
num_eventos=search(',  {e} eps,', line).named['e']
tiempo_ev=search('eps, {g} ns/event',line).named['g']
numero+=float(num_eventos.replace(',','.'))
tiempo+=float(tiempo_ev.replace(',','.'))
     contador+=1
```

```
media=numero/contador
media_tiempo=tiempo/contador
print("Media de eventos:",media,"evs")
print("Media tiempo por evento:",media_tiempo,"ns/ev")
```

En esta ocasión se ha usado una librería llamada "parse" para buscar en cada línea del archivo el valor necesario y a continuación con todos ellos se hace una media general. Este script en Python se ejecuta para todos los ficheros que se han obtenido de todos los filtros y a continuación se muestran en una gráfica.

10.3 Resultados de las pruebas realizadas a los filtros

A continuación, se muestran los resultados obtenidos de los diferentes filtros empezando por la métrica de procesamiento para seguir con la métrica de movimiento y se comparará y analizaráen cada caso. Al final se obtendránunas conclusiones de cuál es el filtro con mejor puntuación.

10.3.1 Métrica de procesamiento

Se van a presentar los resultados obtenidos del script de procesamiento de las mediciones de velocidad del filtro:

	Lucas-Kanade	PatchFlow	EMD
Eventos/s	736420.66	13263.93	2009947.86
ns/Eventos	1482.16	75962.08	530.95

Tabla 1: Resultado de la métrica de procesamiento

Como se puede observar el algoritmo EMD, desarrollado e implementado en este TFM, es un claro candidato a la implementación futura en FPGA, ya que al ser sencillo y no requerir de cálculos complejos se alcanzan unas velocidades de procesamiento que permite "Real Time".

10.3.2 Métrica de movimiento

Ahora se mostrarán las gráficas comparativas de los distintos filtros donde el EMD es el desarrollado en este trabajo. También se mostrará la tabla comparativa de errores en base a la regresión lineal, que estima el perfil de velocidades lineales para comparar las respuestas de los distintos filtros y comparar sus respectivas linealidades:

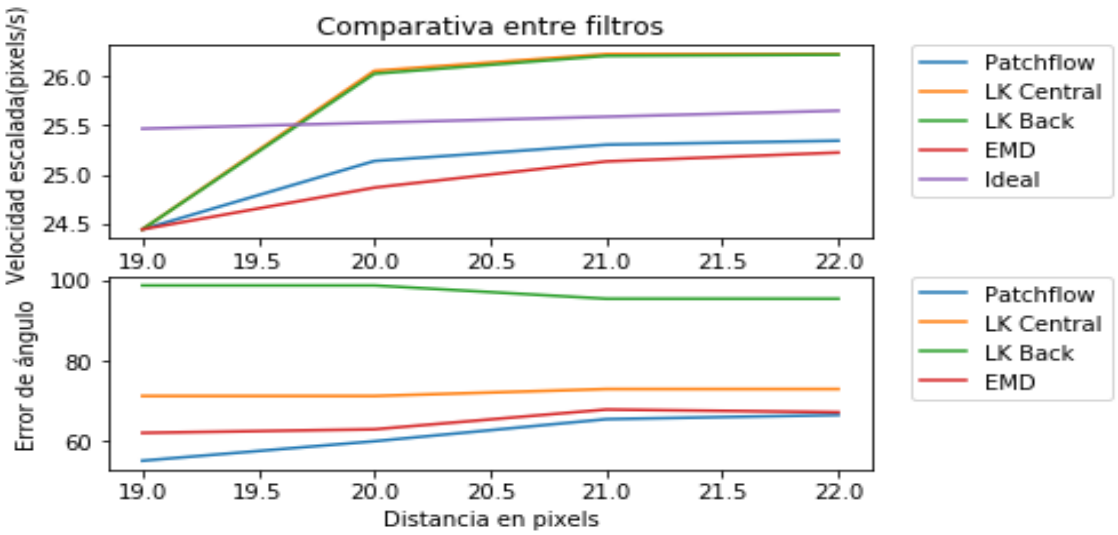

Ilustración 43:Comparativa de filtros

La tabla con el error se muestra a continuación:

Error de linealidad (MSE)			
EMD	PatchFLow	LK Central finite	LK Backward
0,102	0,08	0,24	0,23

Tabla 2: Error de linealidad de los distintos filtros

Como se puede observar, en cuanto al error de linealidad de los filtros, el que tiene menor error es el Patchflow, le sigue el EMD, que es el desarrollado en este trabajo. En cuanto al error de ángulo tenemos el mismo orden.

De estos resultados se deduce que para la configuración de coeficientes encontrados la velocidad será lo más lineal posible. Esta combinación se ha obtenido realizando el siguiente proceso iterativo:

- Se han escogido unos valores al azar altos (en torno a 1s para los coeficientes referidos a los eventos de tipo ON-ON y OFF-OFF y del orden de decenas de ms, lo que llega al combinador es el tiempo de vuelo de cada una de las salidas de los distintos detectores.
- Una vez realizada la simulación y recogidos los datos de velocidad en un archivo .txt, se ejecuta el script mostrado más arriba para el cálculo de las velocidades según la distancia al centro, y se compara con la recta de regresión ideal obtenida a partir de los datos de los diversos filtros.
- Si no es muy bueno el resultado obtenido, es decir, si el error calculado como la media cuadrática es muy alto (del orden de 50 o superior), significa que la respuesta del filtro no es lineal.
- Reajustamos los pesos, y repetimos el proceso hasta que el error cometido se reduzca.

De ese proceso iterativo hemos obtenido que los pesos en nuestro caso y tomando como referencia el archivo de eventos de "fastDot" son:

Coeficientes del filtro			
OFF-OFF	OFF-ON	ON-OFF	ON-ON
496,1m	7,7m	7,8m	415,8m

Tabla 3: Pesos del filtro

11 CONCLUSIONES Y TRABAJOS FUTUROS

Como se ha podido observar a lo largo de todo este trabajo, el filtro "Elementary Motion Detector" o detector de Reichadt, ha supuesto un reto en cuanto a implementación perose han llegado a alcanzar mejoras en la precisión y, en algunos aspectos como el procesamientoincluso ha podido superar otros filtros de flujo óptico.

Aún queda mucho por hacer en el campo de la estimación del movimiento mediante sistemas bioinspirados, y en cuanto al tema tratado en este trabajo, quedan abiertas varias cuestiones en las que sería necesario profundizar. Una de las más evidentes es la implementación en hardware mediante FPGA para evaluar lavelocidad de procesamientode este filtro y la facilidad de implementación. Por otro lado, este filtro tiene una calibración manual. Se han hechos diversas pruebas para calibrar el filtro de formaque su respuesta sea lo más fiel posible al perfil de velocidad que cabría esperar en un movimiento circular uniforme. En un futuro se espera que los parámetros de este filtro puedan ser autosintonizados usando técnicas de redes neuronales o implementar esta idea en algunas de las arquitecturas emergentes de Spinking Neuronal Network.

REFERENCIAS

[1] Patrick Lichtsteiner, Member, IEEE, Christoph Posch, Member, IEEE, and Tobi Delbruck, Senior Member, IEEE A 128 128 120 dB 15 s Latency Asynchronous

Temporal Contrast Vision Sensor.

[2] DVS and DAVIS Specifications

[3]DharaPatel,Saurabh Upadhyay Optical Flow Measurement using Lucas kanade Method

[4] Ryad Benosman, Sio-Hoi Ieng , Charles Clercq , Chiara Bartolozzi , Mandyam Srinivasan. Asynchronous *frame*less event-based optical flow

[5] Ryad Benosman, Charles Clercq, Xavier Lagorce, Sio-Hoi Ieng, Chiara Bartolozzi. Event-based Visual Flow

[6] J. Conradt, Senior Member, IEEE. On-Board Real-Time Optic-Flow for Miniature Event-Based Vision Sensors

[7] Berthold K.P. Horn and Brian G. Rhunck. Determining Optical Flow

[8] https://es.wikipedia.org/wiki/Block_matching

[9] Sonam T. Khawase, Shailesh D. Kamble, Nileshsingh V. Thakur, Akshay S. Patharkar.An Overview of Block Matching Algorithms forMotion Vector Estimation

[10] Razali Yaakob, AlihosseinAryanfar, Alfian Abdul Halin, Nasir Sulaiman. A Comparison of Different Block Matching Algorithms for Motion Estimation

[11] ArohBarjatya, Student Member, IEEE. Block Matching Algorithms

For Motion Estimation

[12]https://es.mathworks.com/help/symbolic/ezsurf.html

[13]Min Liu and Tobi Delbruck. Block-Matching Optical Flow for Dynamic Vision

Sensors: Algorithm and FPGA Implementation

[14]Martin Egelhaaf, Alexander Borst, and Werner Reichardt.Computational structure of a biological motion-detection system as revealed by local detector analysis in the fly's nervous system.

[15]Reichardt, Werner; Egelhaaf, Martin.Properties of individual movement detectors as derivedfrombehavioural experiments on the visual system of the fly.

[16] Hubert Eichner.Internal Structure of the Fly Elementary Motion Detector.

[17] James E Fitzgerald, Damon A Clark.Nonlinear circuits for naturalistic visualmotion estimation.

[18]L.F.Abbott. Lapicque'sintroductionoftheintegrate-and-firemodelneuron(1907)

[19]M. B. Milde,O. J. N. Bertrand,H. Ramachandran,

[20]Wulfram Gerstner and Werner M. Kistler.Spiking Neuron Models

[21]jAER Open Source Project. 2007 [Online]. Available:https://github.com/SensorsINI/jaer

[22] JJ Gibson **What gives rise to the perception of motion?**

[23] Aryan Esfandiari. Real-time and high-speedvibrissae monitoring withdynamic vision sensorsand embedded systems.

Glosario

AER Adress Event Representation

CPLD Complex Programmable Logic Device

CCD charge-coupled device

CMOS complementary metal-oxide-semiconductor

FPGA field-programmable gate array

USB Universal Serial Bus

FIFO First-In, First-Out

CAVIAR European project: "Convolution Address-Event-Representation (AER) Vision Architecture for Real-Time"