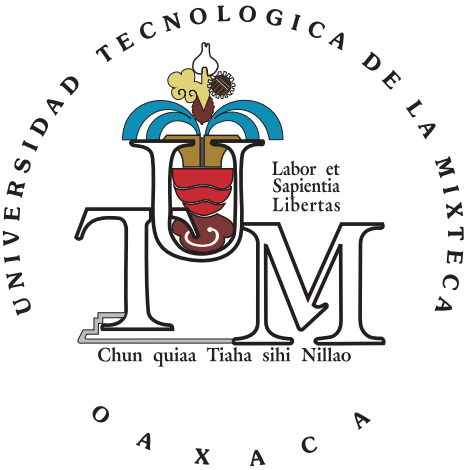


UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA

"CREACIÓN DEL FRAMEWORK nD-EVM/KOHONEN PARA LA REPRESENTACIÓN, SEGMENTACIÓN Y COMPACTACIÓN DE SECUENCIAS DE VIDEO"

PROYECTO DE TESIS

PARA OBTENER EL GRADO DE

MAESTRO EN TECNOLOGÍAS DE CÓMPUTO APLICADO

PRESENTA

ING. JOSÉ YOVANY LUIS GARCÍA

DIRECTOR DE TESIS

DR. RICARDO PÉREZ AGUILA

HUAJUAPAN DE LEÓN, OAXACA, OCTUBRE DEL 2015

Tesis presentada el 19 de Octubre de 2015
ante los siguientes sinodales:

Dr. Felipe de Jesús Trujillo Romero
Dr. Santiago Omar Caballero Morales
Dr. Enrique Guzmán Ramírez
M.C. Gabriel Gerónimo Castillo

Director de tesis:
Dr. Ricardo Pérez Aguila

Dedicatoria

A mis dos amores y la mayor fuente de motivación para seguir adelante:

Maya y Gaby

de todo corazón

Yovany

Agradecimientos

A mis padres *María Concepción* y *José Domingo*, por el apoyo incondicional y por comprender mi deseo por seguir creciendo como persona y profesionalmente.

A mi esposa *Gabriela*, porque juntos todo es posible y por regalarme lo más hermoso que pudo haber pasado, mi hermosa nena *Maya*.

A mi nuevo amor, *Maya Gabriela*, porque llegaste a nuestras vidas cambiando todo nuestro mundo y porque ahora el panorama es prometedor y muy divertido.

A mi director de tesis *Dr. Ricardo Pérez Aguila*, por haber confiado en mis capacidades para desarrollar este proyecto de investigación y por apoyarme en la comprensión de los fundamentos teóricos.

Resumen

Recientemente, las aplicaciones de imagen y video basadas en contenido, han recibido mucho interés, debido a que entre sus aplicaciones destacan: recuperación y búsqueda, resúmenes, análisis de eventos y edición, por mencionar algunas. Para su funcionamiento, estas aplicaciones necesitan identificar lo que sucede en una escena de video, esto es el contenido de la escena. Ello se logra abstrayendo el contenido semántico, el cual tiene su representación mediante objetos de interés que conforman dicha escena. De estos objetos, se obtienen sus características intrínsecas, que los describen de manera explícita y aportan información semántica. A partir de esta información, se puede realizar procesamiento para interpretar los eventos que tienen lugar en la escena. Es por ello que, una de las tareas más importantes para procesamiento de imagen y video, es la separación de la información en sus partes constituyentes, esto es la segmentación de imagen y video. El proceso de segmentación consiste en la partición de datos en grupos de información que comparten características similares. Por ello, este proceso es crucial en la extracción de contenido semántico, en procesamiento de información multimedia digital, reconocimiento de patrones y visión por computadora. En base a lo anterior, este trabajo de investigación propone la creación de un framework para representación, segmentación y compactación de secuencias de video. El cual tendrá como objetivo principal el siguiente flujo de operación: *"A partir de un video de entrada proporcionar una versión segmentada y compactada del mismo"*. Para la representación se propone usar el Modelo de Vértices Extremos (*n-Dimensional Extreme Vertices Model*, *n*D-EVM) y para la segmentación redes de Kohonen.

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Índice de códigos

Parte I

Introducción

Capítulo 1

Planteamiento del Proyecto

1.1 Descripción del Problema

Actualmente, las aplicaciones de imagen y video basadas en contenido han recibido mucho interés, debido a que entre sus aplicaciones destacan: recuperación y búsqueda, resúmenes, análisis de eventos y edición, por mencionar algunas [1, 2]. Con las nuevas tecnologías de alta definición, la información contenida en datos multimedia es muy grande. Esto conlleva a que para desarrollar este tipo de aplicaciones, se requiere procesar grandes volúmenes de información. Sin embargo, esto representa un costo computacional muy elevado. Por ello, se requiere el uso de representaciones de información y algoritmos que permitan un procesamiento eficiente. Es por eso que, en el área de visión por computadora, se han realizado trabajos de investigación que abordan el problema de representación de información multimedia, véase por ejemplo [3].

Por otra parte, para su funcionamiento, las aplicaciones que se mencionaron anteriormente, deben analizar lo que sucede en una escena de video, esto es el contenido de la escena. Lo anterior se logra abstrayendo el contenido semántico, el cual tiene su representación mediante objetos de interés que conforman dicha escena [1, 4]. De estos objetos, se obtienen sus características intrínsecas, que los describen de manera explícita y aportan información semántica. A partir de esta información, es posible realizar procesamiento para interpretar los eventos que tienen lugar. Es por ello que, una de las tareas más importantes para procesamiento de imagen y video, es la separación de la información en sus partes constituyentes, esto es la segmentación de imagen y video.

En términos generales, el proceso de segmentación puede ser visto como la partición de datos en grupos de información, en donde dichos grupos comparten características similares [5, 6, 7]. Por lo anterior, el proceso de segmentación es crucial en sistemas de extracción de contenido semántico, procesamiento de información multimedia digital, reconocimiento de patrones y visión por computadora.

Para el procesamiento de video (o imagen) se debe definir un marco de trabajo (*framework*), en el que se especifican, de manera general, las etapas a seguir para realizar dicho procesamiento. Para ello se han propuesto varios enfoques, lo cuales proporcionan pautas a seguir en el procesamiento [8, 9]. No obstante, debido a que las aplicaciones que se derivan del procesamiento de video tienen necesidades diferentes, el framework puede ser ajustado en base a éstas.

Un framework básico para el procesamiento de imagen y video se ilustra en la figura 1.1, en donde se observa que la etapa de segmentación es de crucial importancia, ya que las etapas subsecuentes dependen directamente de los resultados obtenidos en ella.

Las etapas del framework que se muestra en la figura 1.1 se describen a continuación:

Figura 1.1: Framework básico para procesamiento de imagen y video.

- *Video de Entrada:* Esta etapa consiste en la captura de video, en donde se consideran aspectos como la configuración de la cámara e iluminación.
- *Pre-procesamiento:* En general, en esta etapa se realiza un acondicionamiento del video de entrada para facilitar el procesamiento en etapas posteriores. Ejemplos de pre-procesamiento son: eliminar ruido, suavizar bordes y superficies y ajuste de brillo.
- *Segmentación:* En esta etapa se realiza la separación del video en regiones con información similar. Por ejemplo, una región puede contener solo información del área verde (o bosque) en un video.
- *Representación:* Se efectúa el cálculo de descriptores para cada región obtenida en la etapa anterior. Es deseable que los descriptores proporcionen información discriminante para diferenciar las regiones.
- *Clasificación:* En esta etapa se realiza una clasificación de las regiones en base a la información contenida en los descriptores.

1.1.1. Un Nuevo Enfoque para la Segmentación de Video

Dicho lo anterior, este trabajo de investigación aborda el problema de representación, segmentación y compactación de video. Para la representación se propone usar el Modelo de Vértices Extremos (*n-Dimensional Extreme Vertices Model*, nD-EVM) y para la segmentación mapas auto-organizados (*Self-Organized Maps*, SOM) en específico el modelo de Kohonen.

El nD-EVM es un modelo completo y conciso de representación de Pseudo-Politopos Ortogonales n-Dimensionales (*n-Dimensional Orthogonal Pseudo-Polytopes*, nD-OPP) [10, 11]. Una de las principales ventajas de este modelo, es que permite obtener una representación de nD-OPPs mediante el conjunto de Vértices Extremos, este modelo ha sido utilizado de manera satisfactoria para representar conjuntos de datos en el espacio Euclidiano n-Dimensional. Algunas aplicaciones notables son las siguientes: Representación y visualización de video [3], modelado de conjuntos de datos en 3D y 4D [12, 13], y extracción de información, tal como extracción de fronteras [14] y cálculo de Compacidad Discreta (*Discrete Compactness*, DC) [15].

Para el nD-EVM, se cuenta con un conjunto de algoritmos y operaciones que son útiles para el desarrollo de aplicaciones. Considerando esto, el nD-EVM proporciona una ventaja significativa, ya que permite representar datos en múltiples dimensiones. Lo anterior es útil para este trabajo de investigación, ya que al procesar un video como una secuencia de imágenes, se puede obtener una representación empleando varias dimensiones, por ejemplo: dimensiones para las coordenadas de la posición espacial de los píxeles, dimensiones para representar la profundidad de color usando el modelo RGB y la dimensión para la variable temporal.

Ahora, los SOMs son una clase de red neuronal artificial, que se caracterizan por obtener de manera automática un mapa con la estructura intrínseca de un conjunto de patrones de entrada [16, 17, 18]. El mapa que se obtiene con dicha red, agrupa conjuntos de patrones con características similares en clases.

Para realizar el agrupamiento de patrones, un SOM se apoya en reglas de similitud o disimilitud, que son indicadores para asociar un patrón a una neurona en particular. También dichas reglas proporcionan criterios para determinar la similitud entre los patrones de entrada. La regla clásica en la teoría de mapas auto-organizados de Kohonen es la distancia Euclidiana [16, 19, 20], no obstante se pueden utilizar otras reglas, véase por ejemplo [21].

En este trabajo se propone usar el descriptor de DC como descriptor para pequeñas regiones dentro de una animación. La compacidad discreta es un descriptor para objetos con representación por hiper-voxelización, y presenta una mejora significativa al descriptor clásico de Compacidad de Forma [15, 22, 23]. De esta manera, empleando un SOM se podrán agrupar regiones con características similares.

1.2 Planteamiento del Proyecto de Investigación

Con base en lo descrito en la sección 1.1, el presente trabajo de investigación plantea abordar el problema de representación y segmentación de video utilizando el modelo de nD-EVMs y SOMS bajo el modelo de Kohonen. Como resultado se presentará un *framework*, el cual tendrá como objetivo principal el siguiente flujo de operación: *"A partir de un video de entrada proporcionar una versión segmentada y compactada del mismo"*.

Es importante considerar que, este proyecto forma parte de los objetivos relacionados con la generación de recursos humanos del Grupo de Investigación Multidisciplinaria Aplicada a la Educación y la Ingeniería (GIMAEI). Este grupo está constituido en el Marco de la Universidad Tecnológica de la Mixteca (UTM) y por Profesores adscritos al Instituto de Computación y al Instituto de Física y Matemáticas.

De manera concreta, el proceso de representación y segmentación de video propuesto, consiste de tres etapas principales. Dichas etapas se ilustran en la figura 1.2 y se describen a continuación:

- **Representación mediante el nD-EVM:** En esta etapa, se toma como entrada un video en forma de secuencia de imágenes ordenadas en el tiempo. Se obtiene su representación mediante el nD-EVM, para $n = 6$, en donde se contemplan las siguientes dimensiones: Dos dimensiones para la posición espacial de los píxeles, es decir, las coordenadas (x, y); Tres dimensiones para representar el color usando el modelo RGB (*Red, Green, Blue*); Y por último, una dimensión para representar la variable temporal, sobre la cual se ordena la secuencia de imágenes.

- **Entrenamiento del SOM:** Esta etapa consiste en realizar el entrenamiento de un SOM de una dimensión (1D) utilizando como punto de partida el 6D-EVM obtenido anteriormente. La etapa de entrenamiento genera conjuntos de máscaras 6D a partir de la representación del video en el 6D-EVM, las cuales serán agrupadas según sus propiedades geométricas y topológicas.

- **Segmentación de video:** Para esta etapa se utiliza el SOM generado en el proceso de entrenamiento de la etapa anterior. El objetivo principal es la clasificación y recuperación de una versión segmentada del video a partir de su representación en el 6D-EVM. Es decir, se obtendrán segmentos de video en base al grupo en el que se encuentran las máscaras. Más aún, la versión segmentada es básicamente una versión compactada y con menor número de dimensiones respecto del modelo 6D-EVM original, por ende, se pueden realizar operaciones propias de este modelo para abstraer información adicional.

Para este proyecto se considera que los videos de entrada tienen un esquema de color RGB. Sin embargo, es importarte afirmar que el *framework* permitirá procesar videos en cualquier esquema de color.

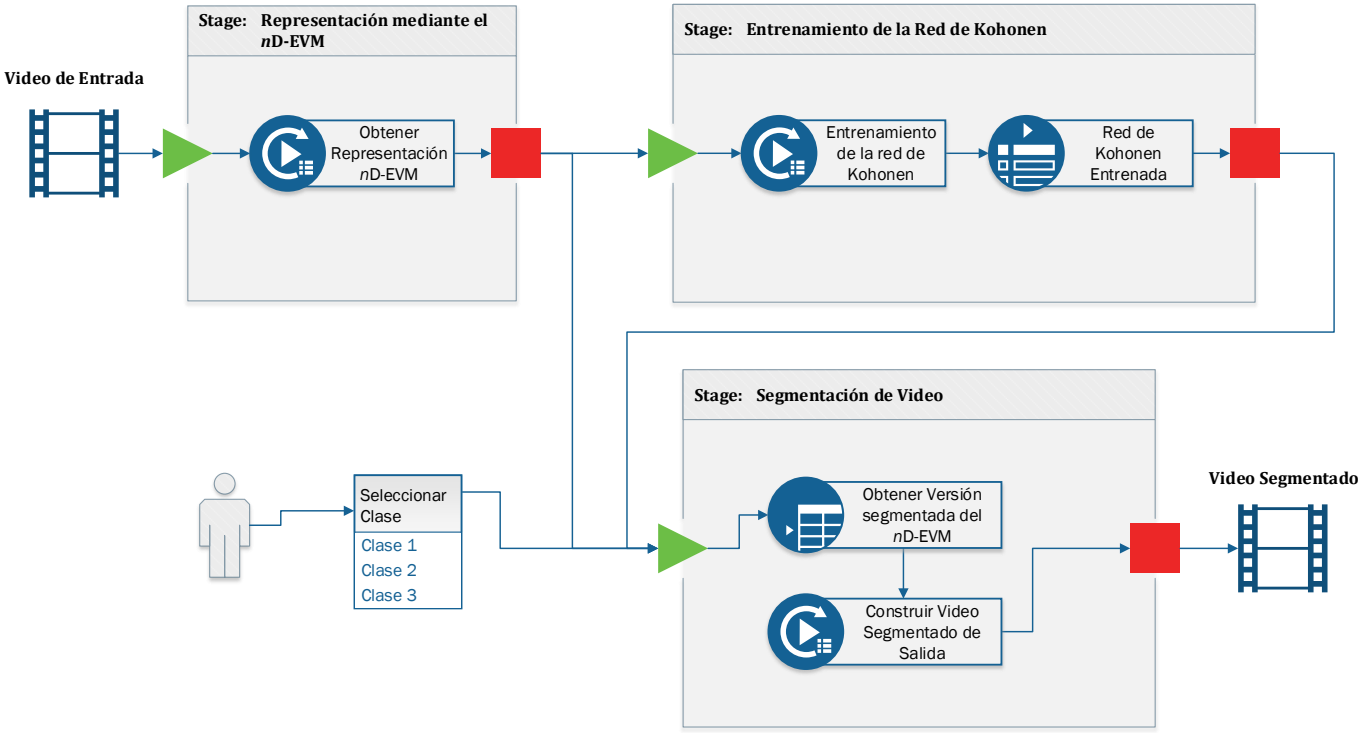


Figura 1.2: Etapas del framework propuesto para representación y segmentación de videos.

1.3 Justificación

El desarrollo de este trabajo de investigación, representa el aporte de un nuevo enfoque en el proceso de segmentación de video. Los beneficios que se derivan de este trabajo, se reflejan en las posibles aplicaciones futuras. Ya que, como se ha mencionado anteriormente, las aplicaciones de abstracción de contenido semántico en video, dependen en gran medida del proceso de segmentación. También, este trabajo proporciona los fundamentos para diversas líneas investigación, ya sea en aplicaciones de video o trabajos de investigación sobre las versiones segmentadas.

1.4 Hipótesis

Empleando el modelo nD-EVM y un SOM, es posible generar un marco de trabajo para representar y segmentar secuencias de video, en el cual se tendrá como entrada un video y se obtendrá como salida una versión segmentada y compactada.

1.5 Objetivos

1.5.1. Objetivo General

Proponer un marco de trabajo para representar y segmentar secuencias de video mediante el modelo nD-EVM y SOMs, en donde se tiene como entrada un video y se obtiene como salida una versión segmentada y compactada de dicho video.

1.5.2. Objetivos Específicos

- Implementar el modelo nD-EVM en el lenguaje de programación C++.
- Implementar los procedimientos básicos del nD-EVM en C++.
- Representar una secuencia de video en el nD-EVM, considerando cualquier esquema de color.
- Implementar, en el lenguaje de programación C++, los procesos de entrenamiento y consulta para un SOM.
- Obtener la versión segmentada de una secuencia de video representada en el nD-EVM.

1.6 Metas

- Contar con una representación, en una estructura de datos, del modelo nD-EVM.
- Obtener una implementación en C++ de la representación del nD-EVM.
- Construir una librería en C++ que contenga los procedimientos básicos para el modelo nD-EVM.
- Construir un módulo para obtener la representación de una secuencia de video en el nD-EVM.
- Contar con la implementación de los procesos de entrenamiento y consulta para una red de Kohonen.
- Construir un módulo para el entrenamiento de la red de Kohonen sobre la secuencia de video representada en el nD-EVM.
- Construir un módulo para realizar consultas a la red de Kohonen, el cual proporcionará una versión segmentada del video de entrada.

1.7 Limitaciones y Delimitaciones

Debido a la naturaleza de los SOMs y el problema que se aborda en este trabajo, se presentan las siguientes limitaciones:

- El tamaño del la red neuronal del SOM debe ser fijo, debido a que operan con un arreglo fijo de neuronas. Esto denota el número de grupos en que se desea segmentar el video.
- Para realizar el entrenamiento de un SOM, se debe proporcionar una regla de similitud fija. Debido a que, como se argumentó anteriormente, existen varias reglas de similitud. Por ello se debe elegir una en particular.
- La segmentación por clases, que proporciona un SOM, no cuenta con etiquetas. Es decir, que se obtienen clases sin nombre, ya que el proceso de etiquetado pertenece a otra etapa en el procesamiento de video. Por lo que el etiquetado de clases queda fuera del alcance del proyecto propuesto.
- El módulo en el que se aplican las operaciones sobre los 6D-EVMs generados, se utilizará solo para realizar consultas y obtener información geométrico/topológicas.

Ahora, con las limitaciones anteriores, se proyectan las siguientes delimitaciones:

- Se debe especificar, por el usuario, el tamaño de la red de Kohonen. Éste es un parámetro a priori, el cual denota el número de clases que se obtendrán en la segmentación.
- Se realizarán experimentos con videos de imagen real y animaciones.
- Se utilizará la Compacidad Discreta como descriptor de regiones y se utilizará la distancia Euclidiana como medida de proximidad para el SOM.

- En la segmentación solo se obtendrán clases sin etiquetas.

1.8 Metodología

Para el desarrollo del framework, que se plantea en este trabajo de investigación (figura 1.2), se propone usar una metodología orientada al desarrollo de procesos de software [24]. Dicha metodología se ilustra en la figura 1.3 y se describe a continuación:

- Primeramente se plantea una Definición Formal del Problema.
- La Investigación Documental consiste en profundizar en los fundamentos teóricos. Es decir, conocer a detalle el modelo nD-EVM y los SOMs, esto con la finalidad de obtener una implementación adecuada.
- Posteriormente, se evalúan los diferentes enfoques y técnicas de implementación de la solución, con la finalidad de elegir la mejor posible.
- Por consiguiente, se debe realizar la implementación del prototipo de la solución.
- Luego, dicha implementación debe ser sometida a pruebas para verificar y validar que se hallan satisfecho los requerimientos.
- Si la solución cumple con los requerimientos, entonces se procede a documentar y publicar los resultados. En caso contrario, se deben realizar cambios, ya sea en la propuesta de solución o en la implementación. Este proceso se realiza hasta que los resultados de las pruebas sean consistentes con los requerimientos.

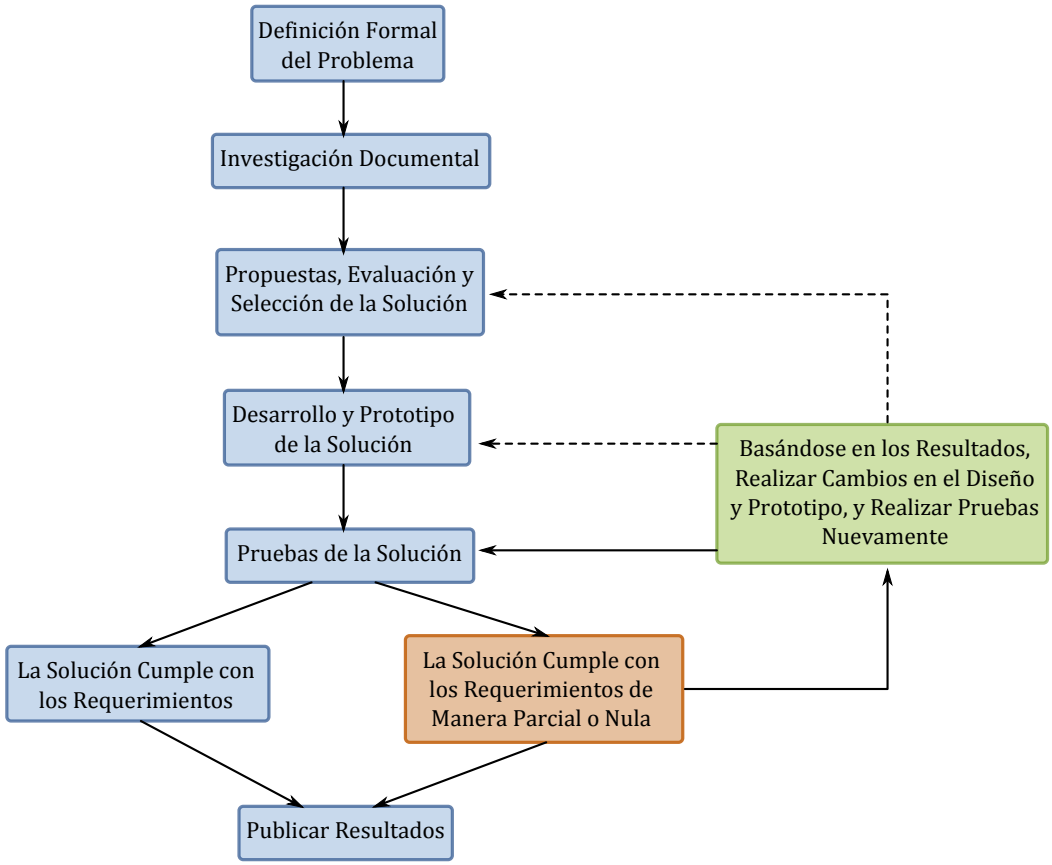


Figura 1.3: Metodología propuesta para el desarrollo del framework.

1.9 Organización del Documento

Para una mejor comprensión del proyecto planteado, a partir de este punto, este documento se organiza en dos Partes principales, que se describen a continuación:

- Marco Teórico: Se abordan los fundamentos teóricos sobre los que se sustenta este proyecto de investigación, y consiste de tres Capítulos. En el Capítulo 2 se presenta una revisión sobre el problema de segmentación de video y sus aplicaciones. El Capítulo 3 presenta los fundamentos teóricos del modelo nD-EVM. El Capítulo 4 presenta la teoría relacionada a los SOMs.
- Desarrollo: Se presenta el desarrollo del framework propuesto y los resultados obtenidos al procesar algunos videos de ejemplo. En el Capítulo 5 se presentan los fundamentos para realizar la representación y segmentación de video. En el Capítulo 6 se muestra la implementación del framework en el lenguaje de programación C++. El capítulo 7 presenta las pruebas y los resultados obtenidos al procesar algunos videos seleccionados. En el capítulo 8 se presentan las conclusiones y perspectivas de trabajos futuros.

Parte II

Marco Teórico

Capítulo 2

Segmentación de Video y sus Aplicaciones

Para comprender en que consiste el problema de segmentación de video, es necesario abordar los fundamentos que lo originan. Por ello, en este capítulo se presenta una descripción de la evolución en materia de procesamiento de imágenes y video como punto de partida. Se presenta un *framework* genérico, desde el punto de vista ingenieril, en el cual se plantean las distintas etapas requeridas en dicho procesamiento. Posteriormente, se da una definición formal del problema de segmentación de imagen y video. También, en este capítulo se presenta una clasificación por categorías, de los diferentes métodos de procesamiento que se han desarrollado hasta la actualidad. Y por último, se hace énfasis en las aplicaciones que se derivan de la segmentación de video.

2.1 Procesamiento de Imagen y Video

Básicamente, el elemento principal en contenido multimedia, que un ser humano puede apreciar visualmente, es una imagen. En este sentido, se puede hablar de una imagen fija, una animación, un video, un dibujo, etc. Una imagen en sí, puede contener mucha información, la cual puede ser abstraída por un ser humano de manera intuitiva. Es decir, que a partir de una imagen nosotros podemos obtener información (contenido semántico) como: la cantidad de personas en la imagen, si fue tomada en el campo o en la ciudad, si hay árboles o no, si es de día o de noche, y así sucesivamente podríamos elaborar una lista considerable de elementos en una imagen. Sin embargo, delegar esta tarea a una computadora no es fácil. Esto se debe a que la información que nosotros podemos obtener de una imagen, está en función de las habilidades cognitivas que hemos desarrollado para procesar información visual. No obstante, es bien sabido que no se tiene conocimiento total sobre los procesos cognitivos del cerebro humano.

En el procesamiento de imágenes, se pretende que un software ejecutándose en una computadora realice las tareas de abstracción de información a partir de imágenes, lo cual constituye los fundamentos de área de visión por computadora. Actualmente en el área de visión por computadora y reconocimiento de patrones, la tarea de obtener información semántica a partir de una imagen, sigue siendo un reto y un área de intensa investigación. En este sentido, se han desarrollado diversos métodos y técnicas para procesamiento de imagen y video. Estos métodos pueden ser clasificados en diferentes categorías según su aplicación [8]:

- **Compresión de imagen y video:** Que como su nombre lo indica, son métodos para compresión de datos de imagen.
- **Manipulación de imagen:** En donde se realizan tareas de edición de imagen, como rotación, ampliación y mejora.

- **Procesamiento de imagen:** En esta categoría se encuentran métodos de mejora de segmentos de imagen. Estos segmentos son regiones de interés, que son mejoradas ignorando el resto de la imagen.
- **Procesamiento de video:** En donde se hallan la mayoría de los métodos de procesamiento de imágenes, pero con la diferencia que aplican métodos que contemplan la variable temporal.
- **Análisis de imagen:** Aquí se encuentran los métodos para el análisis de imágenes que buscan objetos de interés, de los cuales se obtienen sus características.
- **Visión artificial:** Esta categoría engloba los métodos de procesamiento de video, procesamiento y análisis de imagen que normalmente son usados para propósito industrial.
- **Visión por computadora:** Haciendo una analogía de la visión humana, en una computadora se tiene la visión por computadora. En esta categoría se encuentran algoritmos que simulan las capacidades humanas en el contexto de visión. Un ejemplo muy común es el reconocimiento de rostro.

Particularmente, en el área de segmentación de imagen, uno de los primeros métodos fue propuesto en 1965 por Roberts y Tippett [9, 25]. El método propuesto realiza la tarea de detectar bordes en una imagen, el operador Roberts o detector de bordes Roberts. Este operador detecta diferentes partes o regiones en una imagen. Éste es uno de los primeros trabajos relacionados con la descomposición de imágenes en sus elementos que la constituyen. A partir de entonces, se han desarrollado muchos métodos en este campo. Desde el año de la publicación del trabajo de Roberts a la actualidad, han pasado casi 50 años de trayectoria en trabajos de investigación que abordan el problema de segmentación. Y pese a que se han logrado muchos avances, esta área sigue siendo muy activa en cuanto a investigación y desarrollo de aplicaciones.

2.2 Framework General para Sistemas de Procesamiento de Imagen y Video

En general, en el desarrollo de sistemas, que hacen uso de métodos de procesamiento de imágenes y video, se debe emplear un esquema de desarrollo por etapas. En [8, 9] se proponen frameworks para el desarrollo de este tipo de sistemas. Aunque éstos no son idénticos, debido a que el desarrollo depende en gran medida de la aplicación. Sin embargo, estos frameworks proveen una pauta a seguir y hacen énfasis en los elementos fundamentales y necesarios para el procesamiento de imagen y video. En la figura 2.1 se ilustra un esquema para el framework general que contempla conceptos planteados en [8, 9].

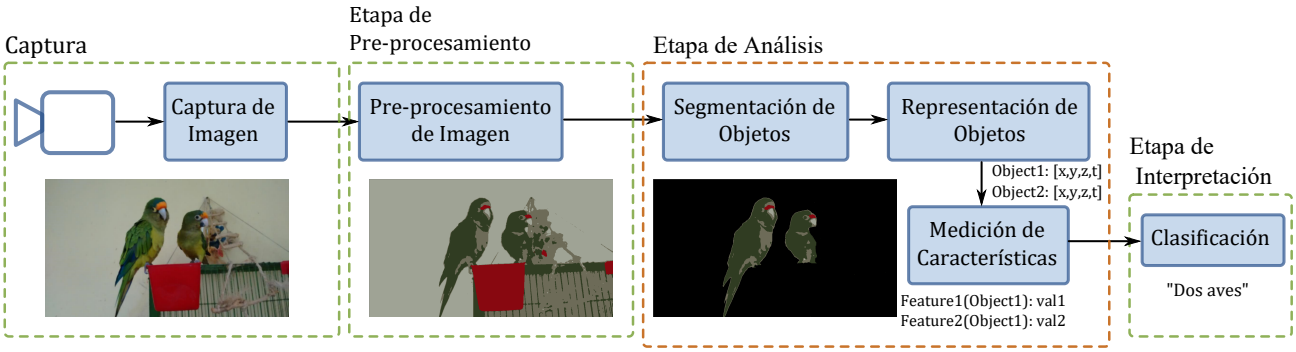


Figura 2.1: Framework general para sistemas de procesamiento de imagen y video.

El Framework de la Figura 2.1 contempla tres etapas principales: Pre-procesamiento, Análisis e Interpretación. La captura de imagen no es considerada una etapa en sí del procesamiento, ya que en esta fase influyen parámetros de configuración de los dispositivos de captura. La etapa de pre-procesamiento consiste en realizar cambios o ajustes a la imagen antes de ser sometida a otras

etapas de procesamiento. Estos cambios pueden ser: Ajuste de color, conversión de color a escala de grises o recorte de zonas de mayor de interés, etc.

Por otro lado, la etapa de Análisis es la etapa más importante en el procesamiento. Ya que en ésta se extrae información de la imagen, la cual es representada a través de datos. La extracción de información requiere de tres procesos, la segmentación, la representación de objetos y la medición de las características de los objetos. La segmentación consiste en dividir una imagen en partes, es decir, extraer regiones de interés (objetos) homogéneas y distintas. La segmentación es una de las tareas mas importantes, ya que los resultados que se obtengan afectarán de manera directa a los procesos y etapas subsecuentes. La representación de objetos se realiza almacenando información representativa en una estructura de datos, tal como la posición espacial. Posteriormente, la medición de características proporciona información intrínseca relacionada con los objetos, como la intensidad de color, textura del objeto, forma, etc.

Y por último, con la información obtenida en la etapa de Análisis, en la etapa de Interpretación, se puede realizar una clasificación de los objetos de la imagen. Es decir, se determina si un objeto pertenece o no a una clase de interés. Con ello, se obtiene información semántica asociada a dichos objetos.

2.3 Definición Formal del Problema de Segmentación

Para abordar el problema de la segmentación de imágenes, se han propuesto varias definiciones formales. En [9], se cita a una definición general que fue propuesta por Fu [5]. Por otro lado, Wan y Higgins propusieron una definición con un enfoque de relaciones de equivalencia [26]. Y, en [6] se describe la definición formal propuesta por Bhattacharyya [7].

Primeramente, considerando al problema de segmentación de imágenes como el proceso de dividir una imagen en un conjunto de regiones (no superpuestas), y que la imagen puede ser reconstruida mediante la unión de dichas regiones (segmentos), hay una serie de requerimientos que estas regiones deben cumplir [27]:

- Deben ser uniformes y homogéneas con respecto a una característica intrínseca.
- El interior debe ser simple y sin huecos.
- Las regiones adyacentes deben tener valores diferentes en cuanto a sus características.
- Las fronteras deben ser simples y suavizadas.

La definición formal propuesta por Fu [5], contempla a una imagen R, la cual puede ser dividida en R_i, $i = 1, 2, \ldots, m$, regiones disjuntas y no vacías. Estas regiones satisfacen las siguientes condiciones:

1. $\bigcup_{i=1}^{m} R_i = R$.
2. $\forall i, j, i \neq j, R_i \cap R_j = \emptyset$.
3. Para $i = 1, 2 \ldots, m$, se tiene que $P(R_i) = TRUE$.
4. Por otra parte, para todo $i \neq j$, se tiene que $P(R_i \cup R_j) = FALSE$.

$P(R_i)$ es un predicado que denota uniformidad en los elementos que constituyen R_i. La condición (1) indica que la unión de todas las regiones forman la imagen original. La condición (2) establece que la intersección de dos regiones distintas es el conjunto vacío, por ende no hay regiones superpuestas. La condición (3) indica que los elementos (píxeles) que forman una región segmentada comparten propiedades similares, y viceversa, elementos de regiones diferentes tienen propiedades diferentes, condición (4).

Por otra parte, una imagen 2D (p. ej. en escala de grises) puede ser representada por una función $f(x,y)$. De esta manera, se puede representar una imagen 3D como una función $f(x,y,z)$. Así, se puede realizar una extensión de la definición de una imagen estática a una secuencia de imágenes (video) de la siguiente manera: $f(x,y) \mapsto f(x,y,t)$. Esto conlleva a extender la definición de segmentación planteada anteriormente, ya que ahora las regiones se encuentran en espacios de mayor dimensión. Como ahora las propiedades de los elementos de la imagen son vectores, la lógica planteada en (3) y (4) debe abordar de manera adecuada la información de dichos vectores.

Desde otra perspectiva, en [6, 7] se propone una definición enfocada a valores de intensidad (de color o escala de grises). Una imagen puede ser vista como una permutación específica de valores de intensidad $p_{i,j}$, es decir, la distribución de intensidad. Supongamos que la distribución de intensidad de una imagen de tamaño $m_1 \times m_2$ se representa de la siguiente manera:

$$\mathscr{P} = \{p_{i,j}, i \leq m_1, j \leq m_2\} \tag{2.1}$$

Si se desea particionar la imagen en K segmentos (clases). El proceso de segmentación se puede definir como una función c de la siguiente forma:

$$c : \mathscr{P} \to [0 \dots K] \tag{2.2}$$

Una clasificación de 256 clases para una imagen en escala de grises, implica formar grupos con base en los niveles de gris $\{t_k, 0 \leq k \leq K\}$, tal que:

$$0 \leq t_0 \leq t_1, \dots, \leq t_k \leq t_{K-1} \leq t_K \leq 255 \tag{2.3}$$

De esta manera, los píxeles con niveles de gris en el rango de $[t_{K-1}, t_K) = s_k$ pertenecen a la clase *k*. Estas clases, al igual que en la definición anterior, satisfacen las siguientes condiciones:

- $s_k \cap s_l = 0$, para $k \neq l, 1 \leq k, l \leq K$.
- $\bigcup_{k=1}^{K} s_k = [0 \dots 255]$.

2.4 Aplicaciones

Retomando el problema de analizar una imagen, surgen dudas acerca de cómo, a partir de ella podemos hallar objetos de interés de la escena y cómo nuestro cerebro procesa y entiende dicha escena. Otra duda importante sería: ¿Cuántas actividades están involucradas en el proceso de reconocimiento? Para esto, la posible respuesta radica en el procesamiento de contenido de una escena en busca de información semántica [1, 4], en donde se puede analizar la imagen dividiéndola en objetos, objetos semánticos. La tarea de abstraer los objetos semánticos, desde los primeros trabajos sobre segmentación hasta la actualidad, representa un reto en las áreas de visión por computadora y reconocimiento de patrones. En el procesamiento de imagen y video, estos objetos pueden ser detectados y usados para tener acceso y manipulación basados en contenido. Ejemplos de procesamiento basados en contenido son: Indexación de videos en bases de datos, tareas de edición y corrección, y codificación eficiente de regiones de interés [1, 2].

Recientemente, se ha dado mucho interés al desarrollo de aplicaciones de video basadas en contenido. Aplicaciones como: reparación y búsqueda de video, resumen de video, análisis de eventos en videos, y edición de video, por mencionar algunas. En este sentido, al efectuar el cómputo de las tareas mencionadas anteriormente, la eficiencia es un factor crucial. Lo anterior se debe a que se procesan grandes volúmenes de información.

Primeramente, para entender el contenido de una escena, es necesario analizar y comprender sus componentes fundamentales. Para ello, los objetos semánticos deben tener un conjunto de

características fundamentales, las cuales describen de manera explícita el significado de éste. Es decir, un objeto semántico establece una relación entre el objeto y aspectos del ambiente en donde se encuentra.

La segmentación puede ser definida como el proceso de particionar información en grupos de subconjuntos de información, los cuales comparten propiedades similares. Este proceso se ha vuelto esencial en la extracción de contenido semántico, procesamiento de información multimedia, reconocimiento de patrones y visión por computadora. Las aplicaciones más comunes de segmentación de imagen y video son:

- **Reconocimiento de objetos:** El objetivo principal detectar objetos mediante la unión de subconjuntos de información con propiedades similares. Las tareas principales de reconocimiento son extracción de características y ajuste de modelos, las cuales dependen en gran medida en la calidad del proceso de segmentación [28].
- **Monitoreo a través de video:** Se realiza mediante la división de objetos en múltiples piezas, que sirven para realizar rastreo (posicionamiento) de objetos en movimiento con referencia a la variable temporal [29].
- **Indexación de video:** Que se efectúa mediante la asignación de comentarios relacionados a segmentos del video [30, 31]. Con ello, es posible retornar resultados a solicitudes de consulta relacionadas con el contenido del video. Por ejemplo, en [15] se realiza la indexación de video con base en el uso de compacidad discreta.
- **Compresión de datos:** Se hace uso de algoritmos para codificación de objetos de manera independiente, dando como resultado una mejora en la calidad del video (de manera subjetiva). En este caso, la segmentación es usada para particionar cada cuadro (frame) de una secuencia de video en objetos con significado semántico y forma arbitraria. Consecuentemente, se pueden asignar una mayor cantidad de bits de información a las regiones de estos objetos [32], lo cual puede reducir defectos visuales en codificaciones con bajas tasas de bits.
- **Visión por computadora:** En donde se enfoca a la reconstrucción en escenas 3-D a partir de imágenes 2-D tomadas desde diferentes ángulos [33].
- **Aplicaciones video-telefónicas:** Que codificando las áreas de interés con una mayor calidad, se obtiene una mejor apreciación de la escena. Esto se debe a que se asignan codificaciones con menores tasas de bits a la parte de fondo del video, y mayores tasas de bits para las áreas de interés para el usuario [34].
- **Entretenimiento digital:** Tal como la superposición de imágenes, en donde se emplea la segmentación de objetos para cambiar un objeto a diferentes escenas, una escena virtual por ejemplo [35].

Capítulo 3

Modelo de Vértices Extremos en el Espacio n-Dimensional (nD-EVM)

Organización del Capítulo

En este capítulo se presenta la teoría relacionada con el *Modelo de Vértices Extremos*. Se presentan los fundamentos matemáticos que sustentan este modelo y su aplicación. Después del material de introducción de la Sección 3.1, este capítulo se organiza de la siguiente manera:

- La Sección 3.2 presenta los fundamentos para *Pseudo-Politopos Ortogonales n-Dimensionales*.
- La Sección 3.3 aborda los fundamentos del modelo de Vértices Extremos.
- En las Secciones 3.4 y 3.5, se presentan fundamentos para la obtención de elementos internos de politopos representados mediante el Modelo de Vértices Extremos.
- La Sección 3.6, presenta el Teorema relacionado a la *operación XOR regularizada*. Ya que es muy importante para el cálculo de los elementos internos.
- Por último, en la Sección 3.7, se describen los algoritmos básicos para este modelo.

3.1 Introducción

El *Modelo de Vértices Extremos* en el espacio n-Dimensional (*nD-EVM*) consiste en un esquema completo de representación de *Pseudo-Politopos Ortogonales n-Dimensionales* (*nD-OPP*) [10, 11]. Dicha representación consiste en tomar solo un subconjunto de vértices de un nD-OPP, el conjunto de *Vértices Extremos*. Mediante este modelo en [11] se han desarrollado algoritmos para tareas requeridas en el modelado de politopos, tales como: *Operaciones Booleanas cerradas y Regularizadas*, *división de sólidos*, operaciones *Set Membership Classification* y operaciones de medición sobre nD-OPPs.

Algunos trabajos de investigación relacionados al modelo nD-EVM, abordan aplicaciones sobre *representación y visualización de videos* [3], *modelado de conjuntos de datos en 3D y 4D* [12, 13], y extracción de información, tal como *extracción de fronteras* [14] y cálculo de *compacidad discreta* [15]. Estos trabajos muestran que el modelo nD-EVM tiene fundamentos sólidos y que puede ser extendido a diferentes aplicaciones.

Para comprender el modelo nD-EVM, en las siguientes secciones se presentan los fundamentos que sustentan los nD-OPPs (sección 3.2) y el modelo nD-EVM (sección 3.3). Estos fundamentos forman parte de los resultados obtenidos en [10, 11], por ello solo se presentan algunas proposiciones, y su correspondiente justificación y demostración se hallan en los trabajos citados. Por último, en la

sección 3.7 se presentan los procedimientos básicos para extraer información, sobre nD-OPPs, a partir de su representación mediante el modelo nD-EVM.

3.2 Fundamentos de los Pseudo-Politopos Ortogonales n-Dimensionales (nD-OPPs)

Definición 3.1 ([36]). *Una Hiper-Caja n-Dimensional en* $\mathbb{R}^n$ *está dada por la función continua:*

$$I^n : [0,1]^n \to [0,1]^n$$
$$x \sim I^n(x) = x$$

Una Hiper-Caja General k-Dimensional en el conjunto cerrado $A \subset \mathbb{R}^n$ *es descrita por la función continua* $c : [0,1]^k \to A$.

La función I^n de la definición anterior, define una hiper-caja unitaria ($[0,1]^n$) en el espacio n-Dimensional. Por ejemplo: en el espacio 2D se define una caja unitaria y en el espacio 3D se define un cubo unitario.

Definición 3.2 ([36]). *Para todo* $i, 1 \leq i \leq n$*, las dos Hiper-Cajas (n-1)D* $I^n_{(i,0)}$ *e* $I^n_{(i,1)}$ *son definidas de la siguiente manera:*

$$Si\ x \in [0,1]^{n-1}, entonces:$$
$$I^n_{(i,0)}(x) = I^n(x_1, \ldots, x_2, 0, x_i, \ldots, x_{n-1}) = (x_1, \ldots, x_2, 0, x_i, \ldots, x_{n-1})$$
$$I^n_{(i,1)}(x) = I^n(x_1, \ldots, x_2, 1, x_i, \ldots, x_{n-1}) = (x_1, \ldots, x_2, 1, x_i, \ldots, x_{n-1})$$

Definición 3.3 ([36]). *En una Hiper-Caja General c se define una celda* (i, α) *como:* $c_{(i,\alpha)} = c \circ I^n_{(i,\alpha)}$.

De las definiciones anteriores, y considerando que $x_1, x_2 \in [0,1]$, para una hiper-caja en el espacio 2D se pueden definir las siguientes cuatro hiper-cajas:

$$I^2_{(1,0)}(x_1, x_2) = I^2(0, x_2) = (0, x_2)$$
$$I^2_{(1,1)}(x_1, x_2) = I^2(1, x_2) = (1, x_2)$$
$$I^2_{(2,0)}(x_1, x_2) = I^2(x_1, 0) = (x_1, 0)$$
$$I^2_{(2,1)}(x_1, x_2) = I^2(x_1, 1) = (x_1, 1)$$

Las hiper-cajas $I^2_{(1,0)}(x_1, x_2)$ y $I^2_{(1,1)}(x_1, x_2)$, definen las aristas de la hiper-caja 2D, alineadas con respecto a la primera coordenada x_1, ya que ésta permanece constante mientras que x_2 varía en el rango definido ($x_2 \in [0,1]$). De la misma manera, las hiper-cajas $I^2_{(2,0)}(x_1, x_2)$ e $I^2_{(2,1)}(x_1, x_2)$, definen las aristas alineadas con respecto a la coordenada x_2. Estas cuatro aristas representan la frontera de la hiper-caja unitaria en el espacio 2D, como se muestra en la figura 3.1.

Definición 3.4 ([36]). *La orientación de una celda (n-1)D* $c \circ I^n_{(i,\alpha)}$ *se determina por* $(-1)^{i+\alpha}$.

Definición 3.5 ([36]). *Partiendo de las Definiciones 3.3 y 3.4, se define ahora una celda orientada (n-1)D, la cual esta dada por la función* $(-1)^{i+\alpha} \cdot c \circ I^n_{(i,\alpha)}$.

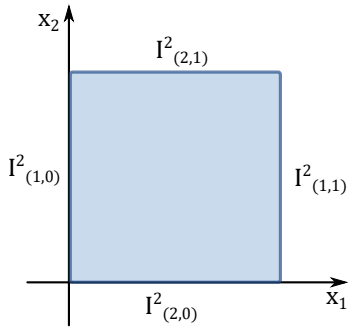


Figura 3.1: Frontera de una hiper-caja.

Para las definiciones anteriores, en el caso de la hiper-caja unitaria en el espacio 2D, se obtienen sus aristas frontera y un signo de orientación correspondiente. En la figura 3.2 se ilustran las celdas orientadas para una hiper-caja unitaria en 2D.

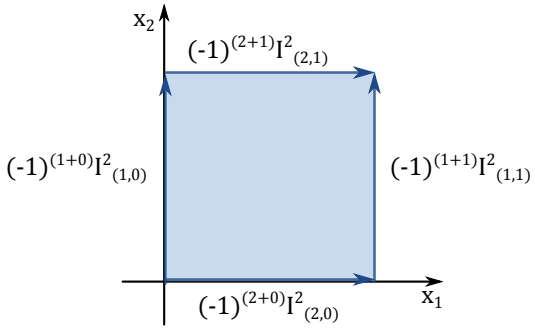


Figura 3.2: Celdas orientadas para una hiper-caja en 2D.

Con base en las definiciones 3.4 y 3.5, a continuación se define una combinación lineal de celdas orientadas, que representa la frontera de una hiper-caja general.

Definición 3.6 ([36])**.** *Una combinación lineal de Hiper-Cajas Generales kD, en donde* $1 \leq k \leq n$*, para un conjunto cerrado A define una k-cadena.*

Definición 3.7 ([36])**.** *A partir de una Hiper-Caja kD* I^n *se define la (n-1)-cadena que denota la frontera de* I^n *de la siguiente forma:*

$$\partial(I^n) = \sum_{i=1}^{n} \left(\sum_{\alpha=0,1} (-1)^{i+\alpha} \cdot I^n_{(i,\alpha)} \right)$$

Definición 3.8 ([36])**.** *Dada una Hiper-Caja General c, se define la (n-1)-cadena para la frontera de c como:*

$$\partial(c) = \sum_{i=1}^{n} \left(\sum_{\alpha=0,1} (-1)^{i+\alpha} \cdot c \circ I^n_{(i,\alpha)} \right)$$

Definición 3.9 ([36])**.** *Para una n-cadena* $\sum c_i$ *en donde cada* c_i *es una Hiper-Caja General nD, se define la frontera de la siguiente manera:* $\partial\left(\sum c_i\right) = \sum \partial(c_i)$.

Basándose en las definiciones anteriores, en la figura 3.3 se muestra un poliedro formado por dos hiper-cajas generales 3D, c_1 y c_2. Dichas hiper-cajas se definen de la siguiente manera:

$$
\begin{aligned}
c_1 : [0,1]^3 &\to \left\{(x_1,x_2,x_3) \in \mathbb{R}^3 : 0 \le x_1 \le 1, 0 \le x_2 \le 1, 0 \le x_3 \le 1\right\} \\
c_1(x) &= (x_1,x_2,x_3) \\
c_2 : [0,1]^3 &\to \left\{(x_1,x_2,x_3) \in \mathbb{R}^3 : 0 \le x_1 \le 1, 0 \le x_2 \le 1, -1 \le x_3 \le 0\right\} \\
c_1(x) &= (x_1,x_2,x_3-1)
\end{aligned}
$$

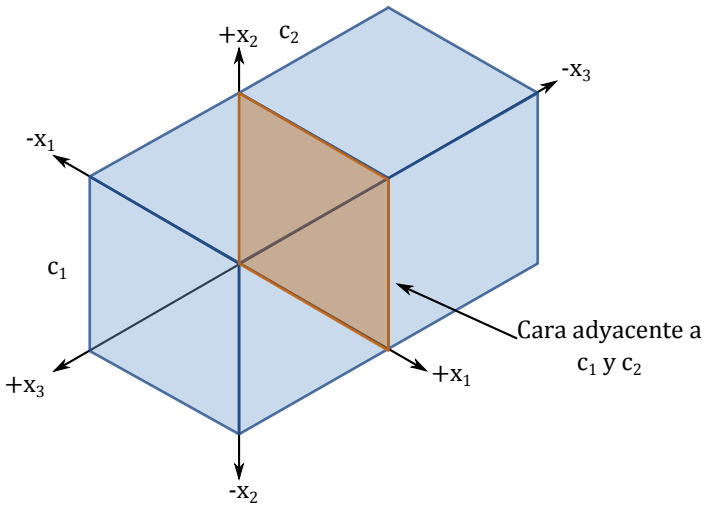


Figura 3.3: Ejemplo de un poliedro formado por la unión de dos cubos unitarios.

La hiper-caja c_1 tiene su vértice inicial en el origen de los ejes coordenados $(0,0,0)$, y la hiper-caja c_2 tiene su vértice inicial en $(0,0,-1)$. De esta manera, c_1 y c_2 forman una 3-cadena que define al poliedro. A partir de dicha cadena, se puede obtener la frontera, como se establece en la definición 3.9. En la figura 3.3, también se indica que c_1 y c_2 tienen una cara adyacente a ambas. Sin embargo, al realizar el cálculo de la frontera, ésta se elimina.

La frontera de la 3-cadena $\sum_{i=1}^{2} c_i$, se obtiene de la siguiente manera:

$$
\begin{aligned}
\partial\left(c_1+c_2\right) &= \partial(c_1)+\partial(c_2) \\
&= \sum_{i=1}^{3}\left(\sum_{\alpha=0,1}(-1)^{i+\alpha}\cdot c_1\left(I^3_{(i,\alpha)}\right)\right) \\
&+ \sum_{i=1}^{3}\left(\sum_{\alpha=0,1}(-1)^{i+\alpha}\cdot c_2\left(I^3_{(i,\alpha)}\right)\right)
\end{aligned}
$$

Con ello se obtienen las celdas orientadas, considerando que en este caso, se obtendrán celdas 2D. Las celdas orientadas se obtienen de la siguiente manera:

$$
\begin{aligned}
\partial\left(c_1+c_2\right) = \{&[-1\cdot(0,x_2,x_3)]+[1\cdot(1,x_2,x_3)]+[1\cdot(x_1,0,x_3)]+[-1\cdot(x_1,1,x_3)]+ \\
&[-1\cdot(x_1,x_2,0)]+[1\cdot(x_1,x_2,1)]\} \\
+\{&[-1\cdot(0,x_2,x_3-1)]+[1\cdot(1,x_2,x_3-1)]+[1\cdot(x_1,0,x_3-1)]+[-1\cdot(x_1,1,x_3-1)]+ \\
&[-1\cdot(x_1,x_2,-1)]+[1\cdot(x_1,x_2,0)]\}
\end{aligned}
$$

Al realizar la adición de las celdas orientadas, se obtiene que la celda $-1 \cdot (x_1, x_2, 0)$ de la hiper-caja c_1 y la celda $1 \cdot (x_1, x_2, 0)$ de la hiper-caja c_2, se eliminan mutuamente debido al signo negativo en la primera celda. De esta manera, se obtiene la 2-cadena $\partial\left(\sum c_i\right)$, que solo contiene las celdas que forman la frontera del 3D-OPP.

Definición 3.10. *Una colección $c_1, c_2, \ldots, c_k$, $1 \leq k \leq 2^n$, de Hiper-Cajas Generales nD se define como una combinación de Hiper-Cajas nD sí y solo sí:*

$$\left[\bigcap_{\alpha=1}^{k} c_\alpha([0,1]^n) = (\underbrace{0,\ldots,0}_{n})\right] \wedge \left[(\forall i, j, i \neq j, 1 \leq i, j \leq k)\left(c_i([0,1]^n) \neq c_j([0,1]^n)\right)\right]$$

La definición anterior indica que en una combinación de Hiper-Cajas, la intersección de todas las Hiper-Cajas nD se efectúa en el origen. También, se establece que no deben existir Hiper-Cajas nD superpuestas. Basándose en esta propiedad, ahora se puede dar la siguiente definición:

Definición 3.11. *Un Pseudo-Politopo Ortogonal n-Dimensional (nD-OPP) p, consiste de una n-cadena compuesta por Hiper-Cajas nD, las cuales están ordenadas de tal manera que al seleccionar un vértice, en cualquiera de dichas Hiper-Cajas, se tiene que el vértice seleccionado es el origen para una combinación de Hiper-Cajas nD compuesta por hasta 2^n Hiper-Cajas. Es decir, que el vértice seleccionado está rodeado por hasta 2^n Hiper-Cajas.*

3.3 Fundamentos del modelo nD-EVM

La teoría referente al modelo nD-EVM, se encuentra en [11], en donde se presentan fundamentos matemáticos que concluyen en una serie de definiciones. En este apartado se presentan solo algunas de ellas, con las cuales se pretende dar al lector un panorama general para la comprensión de este modelo.

Definición 3.12. *Considerando a c, una combinación de Hiper-Cajas nD, una Arista Impar es aquella que tiene un número impar de Hiper-Cajas incidentes de c.*

En la figura 3.4 se muestra un nD-OPP en un espacio 2D, en donde se ilustran las cajas que lo conforman. Como se observa, las aristas punteadas son aquellas que tienen un número par de cajas incidentes, y las aristas con línea continua son aquellas que tienen un número impar de cajas incidentes, estas son las aristas impares. Para el caso 2D, una arista (par o impar) solo puede tener como máximo dos cajas incidentes. No obstante, para el caso 3D se tendrán un máximo de 3 cajas incidentes, y así sucesivamente.

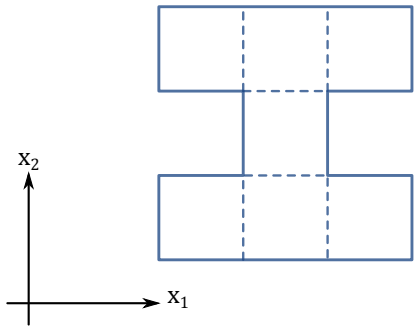


Figura 3.4: Ejemplo de un 2D-OPP y las cajas que lo componen.

Definición 3.13. *Un Brink consiste del segmento máximo sin interrupciones, que puede ser construido a partir de una secuencia de aristas impares de un nD-OPP, dichas aristas son contiguas y colineales.*

Definición 3.14. *Los Vértices Extremos de un nD-OPP p son los vértices finales de todos los Brinks en p. El conjunto de Vértices Extremos de p se denota como EV(p).*

Definición 3.15. *Sea p un nD-OPP, el conjunto de vértices extremos denotado por $EV_i(p)$, es el conjunto de Vértices Extremos de todos los Brinks de p que son paralelos al eje X_i.*

Para ejemplificar las definiciones anteriores, en la figura 3.5 se ilustra un ejemplo de un 3D-OPP y su conjunto de vértices extremos. Se observa que los vértices v_1, v_2 y v_3 no son vértices extremos, ya que tienen un número par de aristas impares que le inciden. La arista representada con una línea punteada, es una arista con un número par de cajas que le inciden, por ello no se considera como arista impar. Los vértices restantes tienen un número impar de aristas impares incidentes, en este caso todos tienen tres aristas impares incidentes.

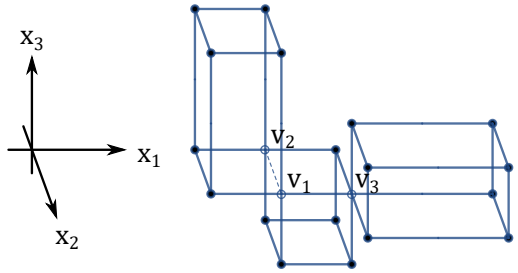
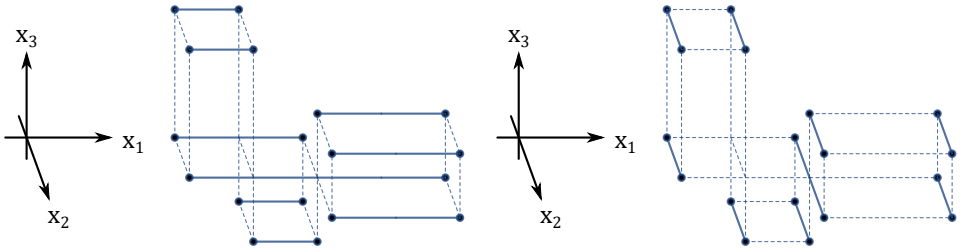


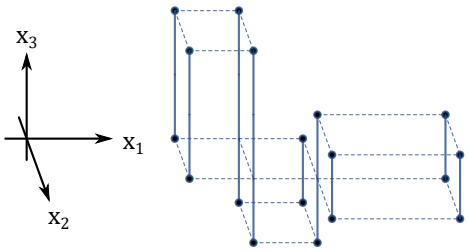
Figura 3.5: Vértices extremos en un 3D-OPP.

Por otra parte, los Brinks en un nD-OPP pueden ser clasificados en base al eje coordenado al cual son paralelos. De esta manera, para el caso 3D, se pueden obtener *3* clasificaciones (conjuntos) de Brinks, paralelos a los ejes coordenados X_i, $1 \leq i \leq 3$ respectivamente. Sin embargo, considerando que los vértices extremos denotan los puntos finales de cada Brink, con cualquiera de las posibles *3* clasificaciones de Brinks, se obtendrían exactamente los mismos vértices extremos. En la figura 3.6 se ilustra lo descrito anteriormente para el caso de un 3D-OPP.



(a) Brinks paralelos al eje X_1.

(b) Brinks paralelos al eje X_2.



(c) Brinks paralelos al eje X_3.

Figura 3.6: Brinks en un 3D-OPP.

Teorema 3.1. *Un vértice de un nD-OPP p, para $n \geq 1$, cuando es descrito por un conjunto de Hiper-Cajas nD que lo rodean, es considerado un Vértice Extremo, sí y sólo sí, está rodeado por un número impar de tales Hiper-Cajas nD.*

La aplicación de este Teorema se puede apreciar en la figura 3.5, en particular en el vértice v_1, el cual tiene dos cajas que lo rodean. También se puede observar que v_1 forma parte de dos Brinks, sin embargo, no es un vértice final para cualquiera de ellos.

Teorema 3.2. *Cualquier Vértice Extremo de un nD-OPP p, para $n \geq 1$, cuando es descrito por Hiper-Cajas nD que lo rodean, tiene exactamente un número n de aristas impares linealmente independientes que le inciden.*

Recurriendo al ejemplo mencionado anteriormente, en el caso de un 3D-EVM como el de la figura 3.5, los vértices extremos tienen tres aristas impares que le inciden. Esto también se puede obtener al considerar que, para las tres posibles clasificaciones de brinks (por cada eje coordenado), se obtienen los mismos vértices extremos. Es decir, que en el caso 3D a un vértice extremo le inciden tres Brinks, cada uno alineado con un eje coordenado diferente.

Teorema 3.3. *Sea p un nD-OPP con sus conjuntos $EV_1(p),\ EV_2(p), \ldots,\ EV_{n-1}(p),\ EV_n(p)$. Entonces se tiene que $EV_1(p) = EV_2(p) = \ldots = EV_{n-1}(p) = EV_n(p)$.*

Definición 3.16. *Sea p un nD-OPP. Se define el Modelo de Vértices Extremos de p, el cual se denota por $EVM_n(P)$, como el modelo que contiene todos los vértices extremos de p.*

Definición 3.17. *Sea p un nD-OPP. Un Hiper-Volumen Extendido kD de p, $1 < k < n$, denotado por $\phi(p)$, es el máximo conjunto de celdas kD de p que se encuentran en un espacio kD, tal que una celda kD e_0 pertenece a un Hiper-Volumen Extendido kD, sí y sólo sí, e_0 pertenece a una celda (n-1)D presente en $\partial(p)$,*

$$(e_0 \in \phi(p)) \Leftrightarrow (\exists c, c\ pertenece\ a\ \partial(p))\ \left(e_0\left([0,1]^k\right) \subseteq c\left([0,1]^{n-1}\right)\right)$$

Definición 3.18. *Sea p un nD-OPP. Se define un Couplet kD de p, $1 < k < n$, como el conjunto máximo de celdas kD de p, tal que una celda kD e_0 pertenece a un Hiper-Volumen Extendido kD, sí y solo sí, e_0 pertenece a una celda (n-1)D presente en $\partial(p)$, la frontera de p.*

En la figura 3.7, se ilustran los Couplets para un 3D-OPP perpendiculares a los ejes coordenados X_1, X_2 y X_3, respectivamente. Como se observa, los Couplets pueden ser vistos como las Caras perpendiculares a su respectivo eje. Los Couplets están formados por un conjunto de celdas (n-1)D contiguas, que en este caso son celdas 2D. Y además, los couplets forman parte de la frontera del 3D-OPP.

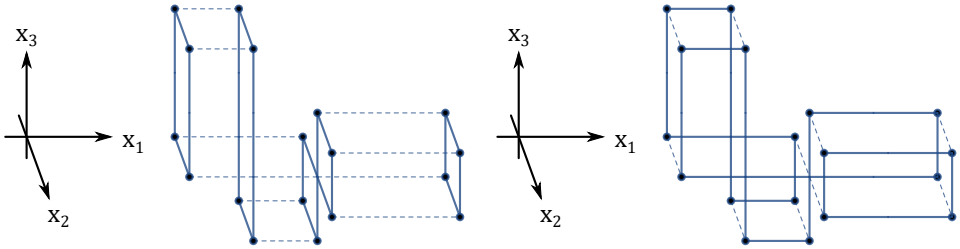
3.4 Secciones y Slices de un nD-OPP

A continuación, se da la definición del operador de proyección, el cual se aplica sobre una celda (n-1)D o bien puntos y uno de los ejes coordenados. Es decir, si se tiene un 3D-OPP, al realizar una proyección en x_1, se obtendrá un 2D-OPP que existe en el espacio formado por los ejes x_2 y x_3.

Definición 3.19. *El operador de Proyección para celdas (n-1)D, puntos y conjunto de puntos, se define de la siguiente manera:*

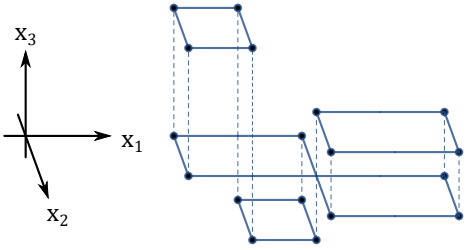
- *Para una celda (n-1)D $c\left(I^n_{(i,\alpha)}(x)\right) = (x_1, \ldots, x_n)$ inmersa en el espacio nD, $\pi_j\left(c\left(I^n_{(i,\alpha)}\right)\right)$ denota el operador de proyección de dicha celda. Esta proyección existe en el Hiper-plano que es perpendicular al eje X_j: $\pi_j\left(c\left(I^n_{(i,\alpha)}\right)\right) = (x_1, \ldots, \hat{x_j}, \ldots, x_n)$.*
- *Sea $v = (x_1, \ldots, x_n)$ un punto en $\mathbb{R}^n$. Para este punto, su proyección en el espacio (n-1)D $\pi_i(v)$ esta dado por $\pi_i(v) = (x_1, \ldots, \hat{x_j}, \ldots, x_n)$.*
- *Para un conjunto de puntos Q en $\mathbb{R}^n$, se define su proyección, denotada por $\pi_j(Q)$, como el conjunto de puntos en $\mathbb{R}^{(n-1)}$ tal que $\pi_j(Q) = \left\{p \in \mathbb{R}^{(n-1)} : p = \pi_j(x), x \in Q \subset \mathbb{R}^n\right\}$*

En las definiciones anteriores, se tiene que $\hat{x_j}$ es la coordenada correspondiente al eje X_j, la cual será eliminada.



(a) Couplets perpendiculares al eje X_1.

(b) Couplets perpendiculares al eje X_2.



(c) Couplets perpendiculares al eje X_3.

Figura 3.7: Couplets en un 3D-OPP.

Definición 3.20. *Para un nD-OPP p se define lo siguiente:*

- np_i *es el número de distintas coordenadas presentes en los vértices de p a través del eje coordenado* X_i, $1 \leq i \leq n$.
- $\Phi_k^i(p)$ *es el k-ésimo couplet (n-1)D de p, el cual es perpendicular al eje coordenado:*

  $X_i, 1 \leq k \leq np_i$

La definición anterior, indica que los couplets se clasifican con respecto al eje coordenado correspondiente y se ordenan en base a las coordenadas presentes en dicho eje. Lo que quiere decir, que para un eje coordenado, hay tantos couplets perpendiculares como coordenadas existentes en el eje.

Definición 3.21. *Un Slice es la región contenida en un nD-OPP p entre dos couplets consecutivos de p.* $Slice_k^i(p)$ *denota el k-ésimo Slice de p, el cual esta delimitado por* $\Phi_k^i(p)$ *y* $\Phi_{k+1}^i(p)$, $1 \leq k < np_i$.

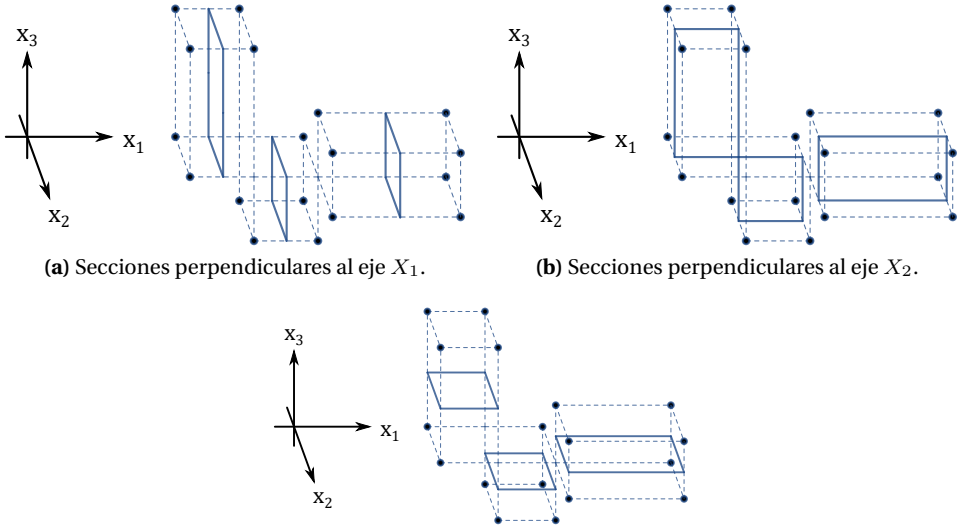
Definición 3.22. *Una Sección, de un nD-OPP p, es un (n-1)D-OPP, con* $n > 1$, *el cual resulta de la intersección entre p y un Hiper-plano (n-1)D perpendicular al eje coordenado* X_i, $1 \leq i \leq n$, *el cual no coincide con ningún couplet (n-1)D de p. En este sentido, una sección puede ser externa o interna, depende de si esta vacía o no.* $S_k^i(p)$ *denota la k-ésima sección de p entre el couplet* $\Phi_k^i(p)$ *y el couplet* $\Phi_{k+1}^i(p)$, $1 \leq k < np_i$.

Para ejemplificar el concepto de sección, en la figura 3.8 se ilustran, para un 3D-OPP, los conjuntos de secciones internas perpendiculares a los ejes coordenados X_1, X_2 y X_3, respectivamente.

Como se observa en la figura 3.8, una Sección es una región que se encuentra acotada entre dos Couplets consecutivos, considerando que los Couplets están coordenados en base a la coordenada a la que son perpendiculares. Con ello, se puede decir que las secciones representan el interior de un nD-OPP.

3.5 Cálculo de Couplets y Secciones

A continuación, se describe cómo se pueden obtener los couplets a partir del conjunto coordenado de secciones. También, se puede realizar el cálculo de las secciones a partir del conjunto



(a) Secciones perpendiculares al eje X_1.

(b) Secciones perpendiculares al eje X_2.

(c) Secciones perpendiculares al eje X_3.

Figura 3.8: Secciones internas para un 3D-OPP.

coordenado de couplets. Ello se logra mediante la operación XOR regularizada.

Teorema 3.4. *La proyección del conjunto de Couplets (n-1)D, $\pi_i\left(\Phi_k^i(p)\right)$, $1 \leq i \leq n$, de un nD-OPP p, se puede obtener mediante el cálculo de la operación XOR regularizada ($\otimes^*$) entre las proyecciones de sus secciones previa y próxima, $\pi_i\left(S_{k-1}^i(p)\right)$ y $\pi_i\left(S_k^i(p)\right)$ respectivamente. Entonces:*

$$\pi_i\left(\Phi_k^i(p)\right) = \pi_i\left(S_{k-1}^i(p)\right) \otimes^* \pi_i\left(S_k^i(p)\right), \forall k \in [1, np_i]$$

Teorema 3.5. *La proyección de cualquier sección, $\pi_i\left(S_k^i(p)\right)$, de un nD-OPP p, puede ser obtenida mediante el cálculo de la operación XOR regularizada entre la proyección de su sección previa, $\pi_i\left(S_{k-1}^i(p)\right)$, y la proyección de su Couplet previo $\pi_i\left(\Phi_k^i(p)\right)$.*

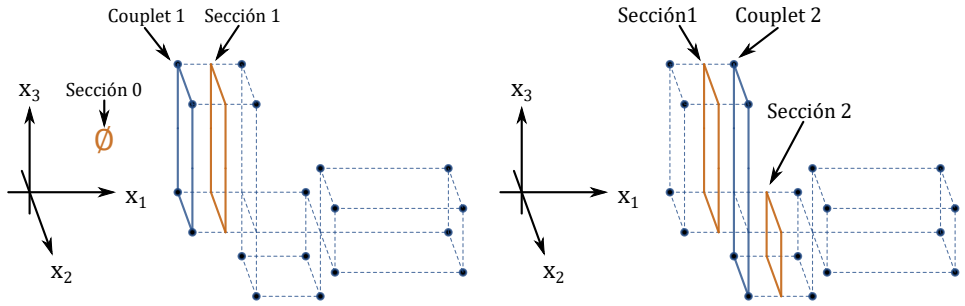
Como se establece en los Teoremas 3.4 y 3.5, para realizar el cálculo de Couplets y Secciones, se debe utilizar el operador de proyección. Esto se debe a que los Couplets y Secciones, para el caso de un 3D-OPP, están inmersos en un espacio 2D. Por lo que, se debe considerar que la coordenada sobre la que se efectúa la proyección, ya no forma parte del espacio 2D de proyección.

Cuando en los Teoremas 3.4 y 3.5, se dice que se obtienen proyecciones de las Secciones y Couplets, significa que éstos se obtienen en un espacio 2D. Por lo que no se proporciona una coordenada para la tercera dimensión. Un ejemplo de proyección se puede efectuar sobre el eje coordenado x_1, entonces el espacio del 2D-EVM resultante, es el plano formado por x_2x_3, el cual es perpendicular al eje x_1.

En la figura 3.9 se ilustra el cálculo de dos secciones, perpendiculares al eje x_1, a partir de los couplets del 3D-EVM que se muestra en la figura 3.5. Con base en el Teorema 3.5, en la figura 3.9a se realiza el cálculo de la primera sección, pero considerando que la sección inicial es un 2D-EVM vacío. Entonces, al realizar la operación XOR regularizada entre el 2D-EVM vacío y el primer Couplet, en esencia se obtiene una copia de este Couplet. En la figura 3.9b, se ilustra el cálculo de la segunda Sección. Esto se realiza al aplicar la operación XOR regularizada entre la Sección anterior y el segundo Couplet.

3.6 Operación XOR Regularizada para el nD-EVM

Teorema 3.6. *Dados dos nD-OPPs p y q, con sus modelos nD-EVM, $EVM_n(p)$ y $EVM_n(q)$, para la aplicación del operador XOR regularizado se tiene la siguiente expresión:*



(a) Cálculo de la primera Sección. **(b)** Cálculo de la segunda Sección.

Figura 3.9: Cálculo de Secciones perpendiculares al eje x_1 a partir de Couplets en un 3D-EVM.

$EVM_n(p \otimes *q) = EVM_n(p) \otimes EVM_n(q)$.

De esta manera, se puede obtener una expresión para calcular secciones de nD-OPPs partiendo de Couplets y viceversa. Esto se logra tomando la representación de sus respectivos EVMs y mediante la combinación del Teorema 3.6 y los Teoremas 3.4 y 3.5. Generando así los siguientes corolarios:

Corolario 3.1. $EVM_{n-1}\left(\pi_i(\Phi_k^i(p))\right) = EVM_{n-1}\left(\pi_i(S_{k-1}^i(p))\right) \otimes EVM_{n-1}\left(\pi_i(S_k^i(p))\right)$

Corolario 3.2. $EVM_{n-1}\left(\pi_i(S_k^i(p))\right) = EVM_{n-1}\left(\pi_i(S_{k-1}^i(p))\right) \otimes EVM_{n-1}\left(\pi_i(\Phi_k^i(p))\right)$

3.7 Algoritmos del nD-EVM

En esta sección se presentan los algoritmos básicos del nD-EVM. Estos algoritmos forman parte de la librería básica para realizar Operaciones Booleanas. Es por ello que se presentan algunos algoritmos para realizar la extracción e inserción de Couplets, los cuales serán operados posteriormente usando el algoritmo de Operaciones Booleanas.

Algoritmo 3.1: Constructor de un nD-EVM.

```
   Output: Un nuevo nD-EVM vacío
1  Procedure initEVM()
2  |  Se retorna un conjunto vacío.
```

Algoritmo 3.2: Insertar un Couplet (n-1)D en un nD-EVM.

```
   Input  : Un (n-1)D-EVM inmerso en un espacio nD.
   Output: Un nD-EVM p.
1  Procedure putCouplet(EVM couplet, EVM p)
2  |  Se inserta el Couplet (n-1)D en p, dicho Couplet es perpendicular al eje coordenado x1.
```

El algoritmo 3.2, se utiliza para insertar un Couplet (n-1)D en un nD-EVM. Una utilidad de este algoritmo radica en que, al realizar la obtención de Couplets a partir de una secuencia de secciones, éstos se pueden insertar en un nD-EVM vacío, y así, al final se obtendrá un nD-EVM resultante con la secuencia de los Couplets. Este algoritmo es necesario porque los Couplets obtenidos al realizar operaciones entre secciones y couplets, están proyectados en un espacio (n-1)D.

Con los algoritmos 3.3 y 3.4, se realiza la lectura de los Couplets que pertenecen a un nD-EVM p. Con ello, se obtiene una secuencia de Couplets (n-1)D y la exploración termina cuando se ha llegado al

final de p.

Algoritmo 3.3: Leer un Couplet (n-1)D de un nD-EVM *p*.

```
   Input : Un nD-EVM p.
   Output: Un hiper-volumen (n-1)D hvl.
1  Procedure readCouplet(EVM p)
2      Se extrae el próximo (n-1)D Couplet, el cual es perpendicular al eje coordenado x_1.
```

Algoritmo 3.4: Determinar si se ha llegado al final del EVM, cuando se leen Couplets (n-1)D consecutivos.

```
   Input : Un nD-EVM p.
   Output: Un valor booleano (True, False).
1  Procedure endEVM(EVM p)
2      if Se llegó al final de p then
3          return True
4      else
5          return False
```

En el algoritmo 3.5, se toma un Couplet (n-1)D y se le agrega una dimensión adicional, en donde se establece la coordenada deseada. Esto es de gran utilidad cuando se obtiene un Couplet o Sección (n-1)D, ya que para formar el correspondiente nD-EVM, se necesita ordenar la secuencia de Couplets o Secciones sobre la coordenada x_1. Con ello , el algoritmo 3.6 obtiene la coordenada para el caso de un Couplet o Sección que se le ha agregado una dimensión y coordenada adicional.

Algoritmo 3.5: Establecer coordenada.

```
   Input : Un (n-1)D-EVM p inmerso en un espacio (n-1)D y una coordenada.
   Output: Un (n-1)D-EVM p inmerso en un espacio nD.
1  Procedure setCoord(EVM p)
2      Establece la coordenada en el eje coordenado x_1 a cada uno de los elementos de p.
```

Algoritmo 3.6: Obtener coordenada.

```
   Input : Un (n-1)D-EVM p inmerso en un espacio nD.
   Output: La coordenada del eje coordenado x_1 del Couplet p.
1  Procedure getCoord(EVM p)
2      Se obtiene la primer coordenada del eje coordenado x_1 del Couplet p.
```

Ahora, el algoritmo 3.7 indica que, cuando se realiza una operación XOR a dos nD-EVMs, el resultado de dicha operación debe ser un nuevo nD-EVM. La operación XOR consiste en preservar los vértices que no están presentes en ambos nD-EVMs. Es decir, que se eliminan los vértices que pertenecen a ambos nD-EVMs. Con este algoritmo 3.7, se puede obtener una Sección a partir de la aplicación de la operación XOR entre una Sección anterior y el Couplet actual. Véase el algoritmo 3.8.

Algoritmo 3.7: Realizar la operación XOR entre dos nD-EVMs.

```
   Input : Dos nD-EVMs p y q.
   Output: Un nD-EVM resultante.
1  Procedure mergeXOR(EVM p,EVM q)
2      Aplica la operación XOR a los nD-EVMs p y q y retorna el resultado como un nuevo nD-EVM.
```

De la misma manera que en el caso anterior, y por medio del algoritmo 3.9 se puede obtener un Couplet, a partir de la aplicación de la operación XOR entre dos secciones consecutivas, una previa al Couplet y otra posterior. Cabe mencionar que los algoritmos 3.8 y 3.9 corresponden a la

Algoritmo 3.8: Obtener una Sección a partir de la Sección previa y el Couplet actual.

```
   Input : Una Sección y un Couplet (n-1)D.
   Output: Un (n-1)D-EVM resultante.
1  Procedure getSection(EVM section,EVM couplet)
2      return mergeXOR(section,couplet);
```

implementación de los corolarios 3.1 y 3.2.

Algoritmo 3.9: Obtener un Couplet a partir de dos Secciones consecutivas.

```
   Input : Dos Secciones (n-1)D.
   Output: Un (n-1)D-EVM resultante.
1  Procedure getCouplet(EVM section1,EVM section2)
2      return mergeXOR(section1,section2);
```

Con los algoritmos 3.1, 3.3, 3.4 y 3.8, se establece el algoritmo 3.10 que obtiene una secuencia de Secciones de manera ordenada. Este algoritmo es importante ya que establece la base para otros algoritmos, en donde se requiere procesar nD-EVMs en base a sus Secciones.

Algoritmo 3.10: Cálculo de la secuencia de Secciones de un nD-EVM p.

```
   Input : Un nD-EVM p.
1  Procedure sectionSequence(nD-EVM p)
2      EVM couplet;
3      EVM S_i , S_j ;                                   // Sección previa y actual
4      couplet ← initEVM();
5      S_i ← initEVM();
6      S_j ← initEVM();
7      couplet ← readCouplet(p);
8      while !endEVM(p) do
9          S_j ← getSection(S_i,couplet);
10         Procesar la Sección actual S_j;
11         S_i ← S_j;
12         couplet ← readCouplet(p);
```

El algoritmo 3.10 realiza la obtención de la secuencia de Secciones de la siguiente manera:

- Se inicializa el Couplet y las Secciones previa y actual ($S_i,\ S_j$) y se extrae el primer Couplet perpendicular al eje x_1. Véase las líneas 4-7.
- Se calcula la Sección actual en base al Couplet actual y la sección previa, línea 9. Es importante considerar que inicialmente se tiene una Sección previa con un EVM vacío.
- Se realiza el procesamiento requerido con la Sección actual, véase la línea 10.
- Se actualiza la Sección previa para la siguiente iteración y se obtiene el siguiente Couplet. Véase las líneas 11 y 12.

3.7.1. Algoritmo de Operaciones Booleanas

En este apartado se presenta un algoritmo para realizar las Operaciones Booleanas Regularizadas. Las Operaciones Booleanas Regularizadas se utilizan en el modelado de politopos, ya que si se consideran las Operaciones Booleanas clásicas de teoría de conjuntos en dos politopos, no siempre se garantiza que se obtentrá un politopo como resultado. Las operaciones Regularizadas abordan este problema y siempre garantizan que se obtendrá un politopo, o bien el conjunto vacío. En este caso el resultado siempre será un nD-EVM válido.

Las Operaciones Booleanas Regularizadas son las siguientes $\{\cup^*, \cap^*, -^*, \otimes^*\}$, de manera textual:

Unión*, Intersección*, Diferencia* y OR Exclusiva*. En el algoritmo 3.11 se describe el procedimiento para realizar las Operaciones Booleanas, el cual lleva el siguiente flujo de operación:

- Como primer paso se tiene la inicialización de las variables que se van a utilizar. Estas variables constan de una sección del nD-EVM p y otra sección para el nD-EVM q (pSection y qSection), la sección previa y actual del nD-EVM resultante (rPrevSection y rCurrSection). También se cuentan con variables de control, que determinan si se obtendrá una sección de p y/o de q (fromP y fromQ). Una variable para determinar la coordenada donde se insertará el Couplet resultante (coord). Y por último se tiene una variable para el nD-EVM resultante $result$. Véase las líneas 2-6 y 10-12.
- La estructura de repetición $While$ de la línea 14 permite explorar los Couplets de p y q de manera ordenada, y esta estructura de repetición se detiene cuando no hay más Couplets en uno de los dos operandos.
- La función *nextObject* de la línea 26 determina, en base al ordenamiento de los Couplets de *p* y *q* sobre el primer eje coordenado (de la iteración actual), si se leerá el Couplet de *p* y/o de *q*, considerando que se pueden leer de ambos (líneas 15-20).
- Posteriormente, se realiza el cálculo de la Sección correspondiente al resultado, al aplicar la Operación Booleana a las Secciones de *p* y *q*, pero en el contexto de un espacio (n-1)D y se hace de manera recursiva (línea 22). Entonces, se realizan todos los pasos anteriores hasta llegar al caso base 1D, en donde se aplicarán las Operaciones Booleanas a segmentos 1D (línea 8).
- Ahora, cuando se retorna de la llamada recursiva, se obtiene una Sección *result*, con la que se puede obtener el siguiente Couplet, el cual se extiende a nD mediante la adición de una coordenada *coord* sobre el eje x_1. Posteriormente se inserta el Couplet resultante en *r*, y de esta manera se va formando *result* hasta obtener la versión final. Véase las líneas 23-26.
- Por último, las estructuras de repetición $while$ de las líneas 27 y 30 se utilizan para procesar los Couplets que resten de alguno de los operandos. Se exploran los Couplets del operando correspondiente para evaluar si serán incluidos en $result$ de acuerdo a la Operación Booleana efectuada. Por ejemplo, al efectuar la operación unión sí se deberán incluir los Couplets restantes, por otro lado para la operación intersección no se incluyen, en la operación diferencia se incluyen solo si es el primer operando el que tiene Couplets restantes, y en la operación XOR si se incluyen.

3.7.2. Operaciones Booleanas para 1D-EVMs

Como se mencionó anteriormente, el caso base para las Operaciones Booleanas de nD-EVMs, es cuando se tienen 1D-EVMs que son segmentos en 1D, los cuales son acotados por dos vértices extremos. Es decir, un segmento 1D tiene un vértice inicial y un vértice final.

En la tabla 3.1 se muestran las Operaciones Booleanas para todos los posibles casos en 1D. En esta tabla se considera que se tienen dos 1D-EVMs de entrada, el primero de ellos tiene los vértices *a* y *b*, y el segundo tiene los vértices *c* y *d*. Dentro de los posibles casos para las operaciones, se distinguen cinco posibles escenarios:

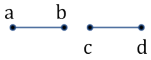
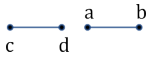
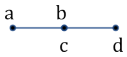
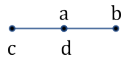
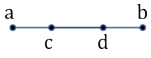
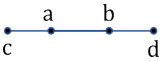
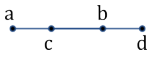
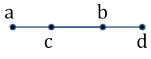
- **Segmentos Disjuntos:** Cuando los segmentos 1D se encuentran separados y no tienen contacto alguno.
- **Segmentos Contiguos:** Este caso se presenta cuando los segmentos tienen un vértice en común. En donde el vértice final de uno de ellos coincide con el vértice inicial del otro segmento.
- **Segmentos Coincidentes:** Este caso se presenta cuando los segmentos tienen todos sus vértices en común. Es decir, que los vértices iniciales y finales coinciden.

Algoritmo 3.11: Algoritmo para aplicar las Operaciones Booleanas a dos nD-EVMs.

```
Input : Dos nD-EVMs p y q, la Operación Booleana a aplicar y el número de dimensiones n.
Output: El nD-EVM resultante.
Procedure booleanOperation(nD-EVM p, nD-EVM q, booleanOperator op, integer n)
    EVM pSection, qSection;
    EVM couplet;
    boolean fromQ, fromP;
    real coord;
    EVM result, rPrevSection, rCurrSection
    if n = 1 then
        return booleanOperation1D(p, q, op);
    else
        pSection ← initEVM();
        qSection ← initEVM();
        rCurrSection ← initEVM();
        nextObject(p, q, fromP, fromQ);
        while !(endEVM(p)) and !(endEVM(q)) do
            if fromP then
                couplet ← readCouplet(p);
                pSection ← getSection(pSection, couplet);
            if fromQ then
                couplet ← readCouplet(q);
                qSection ← getSection(qSection, couplet);
            rPrevSection ← rCurrSection;
            rCurrSection ← booleanOperation(pSection, qSection, n − 1, op);
            couplet ← getCouplet(rPrevSection, rCurrSection);
            setCoord(couplet, coord);
            putCouplet(couplet, result);
            nextObject(p, q, coord, fromP, fromQ);
        while !endEVM(P) do
            couplet ← readCouplet(p);
            putBool(couplet,result,op);
        while !endEVM(Q) do
            couplet ← readCouplet(q);
            putBool(couplet,result,op);
    return result
```

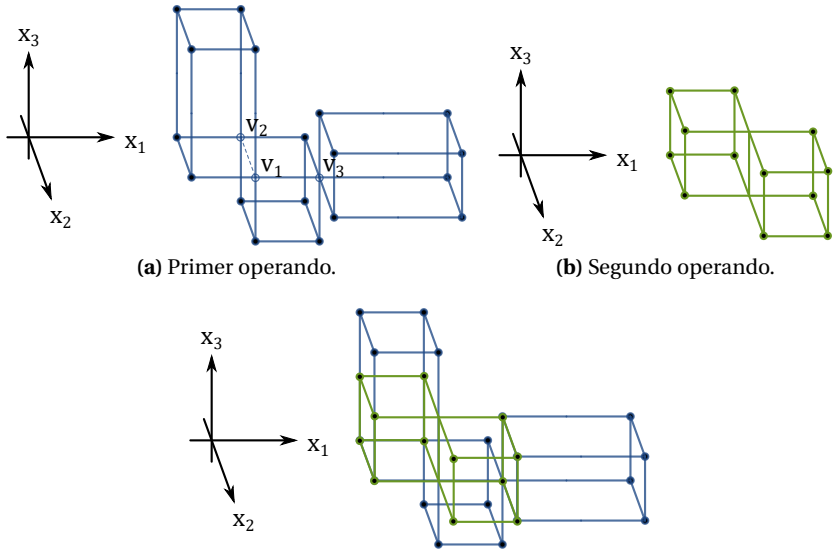
- **Segmentos Inclusivos:** Este caso se presenta cuando un segmento se encuentra dentro de otro.
- **Segmentos Sobrepuestos:** Para este caso, los segmentos comparten solo una parte de ellos. Es decir, que el vértice inicial de uno de ellos, se encuentra en algún lugar entre los vértices inicial y final del otro segmento.

Tabla 3.1: Operaciones Booleanas Regularizadas para 1D-OPPs representados mediante 1D-EVMs.

Casos	1D-EVMs entrantes	$\overline{ab} \cup^* \overline{cd}$	$\overline{ab} \cap^* \overline{cd}$	$\overline{ab} -^* \overline{cd}$	$\overline{ab} \otimes^* \overline{cd}$
Disjuntos:					
	Para: $b < c$	$\{a, b, c, d\}$	$\emptyset$	$\{a, b\}$	$\{a, b, c, d\}$
Disjuntos:					
	Para: $d < a$	$\{c, d, a, b\}$	$\emptyset$	$\{a, b\}$	$\{c, d, a, b\}$
Contiguos:					
	Para: $b = c$	$\{a, d\}$	$\emptyset$	$\{a, b\}$	$\{a, d\}$
Contiguos:					
	Para: $a = d$	$\{c, b\}$	$\emptyset$	$\{a, b\}$	$\{c, b\}$
Coincidentes:					
	Para: $a = c\ y\ b = b$	$\{a = c, b = d\}$	$\{a = c, b = d\}$	$\emptyset$	$\emptyset$
Inclusivos:					
	Para:				
	$a < c < d < b$	$\{a, b\}$	$\{c, d\}$	$\{a, c, d, b\}$	$\{a, c, d, b\}$
	$a = c < d < b$	$\{a = c, b\}$	$\{a = c, d\}$	$\{d, b\}$	$\{d, b\}$
	$a < c < d = b$	$\{a, d = b\}$	$\{c, d = b\}$	$\{a, c\}$	$\{a, c\}$
Inclusivos:					
	Para:				
	$c < a < b < d$	$\{c, d\}$	$\{a, b\}$	$\emptyset$	$\{c, a, b, d\}$
	$c = a < b < d$	$\{c = a, d\}$	$\{c = a, b\}$	$\emptyset$	$\{b, d\}$
	$c < a < b = d$	$\{c, b = d\}$	$\{a, b = d\}$	$\emptyset$	$\{c, a\}$
Sobrepuestos:					
	Para: $a < c < b < d$	$\{a, d\}$	$\{c, b\}$	$\{a, c\}$	$\{a, c, b, d\}$
Sobrepuestos:					
	Para: $c < a < d < b$	$\{c, b\}$	$\{a, d\}$	$\{d, b\}$	$\{c, a, d, b\}$

3.7.3. Ejemplo de Operaciones Booleanas para dos 3D-EVMs

En este apartado se presenta un ejemplo de la aplicación del algoritmo 3.11 para cada una de las Operaciones Booleanas Regularizadas, en donde se utilizan dos 3D-EVMs como operandos. En la figura 3.10 se muestran dichos 3D-EVMs, y en la figura 3.10c se muestra la disposición espacial de ambos operandos.



(a) Primer operando. **(b)** Segundo operando.

(c) Posición espacial de ambos operandos.

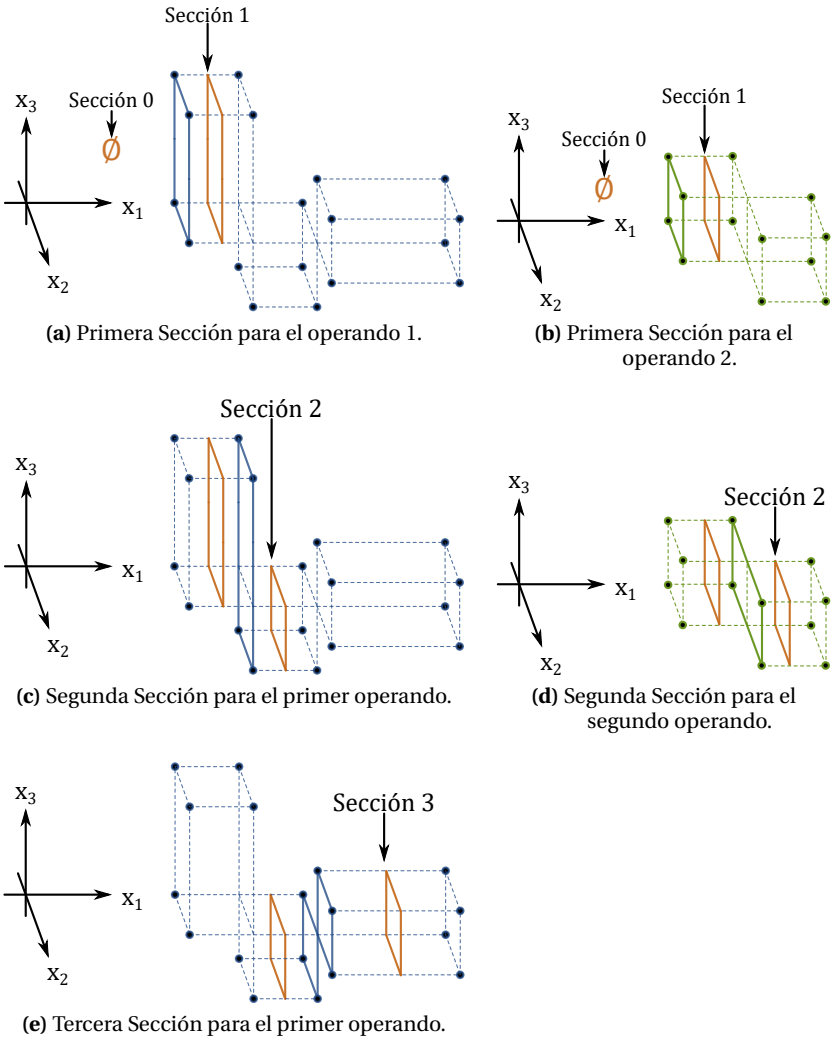
Figura 3.10: Operandos utilizados para los ejemplos de Operaciones Booleanas.

Como ya se ha mencionado anteriormente, el algoritmo de Operaciones Booleanas indica que se debe realizar el cálculo de Secciones a partir de los Couplets, ya que las Operaciones Booleanas se efectúan sobre dichas Secciones. Por ello es importante mostrar cómo se realiza este cálculo para ambos operandos, y como ya se había mencionado anteriormente, el algoritmo 3.10 proporciona los pasos para extraer la secuencia de Secciones de un nD-EVM.

Considerando el algoritmo 3.8, se puede obtener una Sección a partir de la Sección previa y el Couplet actual. Es importante que la Sección previa y el Couplet actual sean consecutivos, ya que de esta manera se obtiene apropiadamente la nueva Sección. Esto se realiza de manera iterativa hasta alcanzar el final del nD-EVM.

En las figuras 3.11a y 3.11b se muestra de manera gráfica cómo se obtiene la primera Sección 2D para ambos operandos. Con base en el algoritmo 3.10, se considera que las Secciones iniciales son 2D-EVMs vacíos, por lo que la primera Sección es igual al primer Couplet. Para la obtención de las siguientes Secciones, en las figuras 3.11c 3.11d se muestra cómo se realiza este proceso utilizando la Sección anterior (sección 1) y el siguiente Couplet, y como se establece en el algoritmo 3.10 se utiliza el método del algoritmo 3.8.

Por último, en la figura 3.11e se muestra el cálculo de la tercer Sección para el primer operando. Para el segundo operando no se tiene una tercer sección, debido a que se ha llegado al final de éste. Con ello y en base al algoritmo 3.11, se presenta el caso en donde se debe especificar para qué Operaciones Booleanas serán considerados los Couplets restantes del primer operando.



(a) Primera Sección para el operando 1.

(b) Primera Sección para el operando 2.

(c) Segunda Sección para el primer operando.

(d) Segunda Sección para el segundo operando.

(e) Tercera Sección para el primer operando.

Figura 3.11: Obtención de las secciones en ambos operandos.

3.7.3.1. Aplicación de la Operación Unión

En esta sección se presenta de manera textual y gráfica, la ejecución del algoritmo de Operaciones Booleanas y la operación Unión, en donde se utiliza como operandos los 3D-EVMs descritos anteriormente y que se muestran en la figura 3.10.

Ahora, en base a las primeras Secciones 2D que se muestran en las figuras 3.11a y 3.11b, se debe realizar la operación Unión entre ellas. Esto implica realizar la operación unión de los 2D-EVMs correspondientes. En la figura 3.12 se muestra cómo se realiza dicha unión, en donde se observa que se deben obtener las Secciones 1D para cada Sección 2D, con lo que se obtiene el caso base descrito del algoritmo 3.11.

Consecuentemente se realiza las Operación Unión sobre las Secciones 1D como se indica en la tabla 3.1. Para este caso solo se tiene una Sección 1D por cada Sección 2D, además es importante considerar que éstas se encuentran en la misma posición espacial en base a la coordenada x_2. Como se observa en la figura 3.12, con la Sección 1D resultante de la operación Unión, se obtienen dos Couplets 1D para formar la Sección 2D resultante.

Para las siguientes Secciones 2D de los operandos que se muestran en las figuras 3.11c y 3.11d,

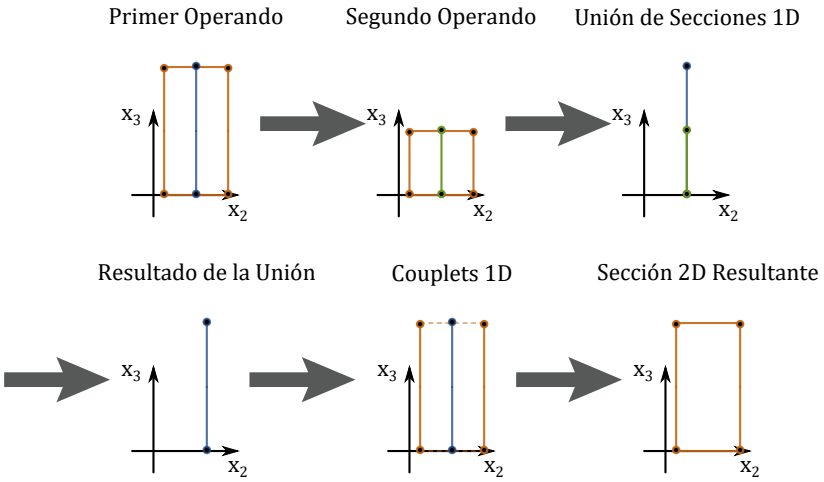


Figura 3.12: Unión de las primeras Secciones 2D de los operandos.

ahora en la figura 3.13 se muestra cómo se realiza la operación Unión entre ellas. Como se observa, la operación se realiza respetando el orden espacial de las Secciones 1D, ya que las Secciones 2D no se encuentran en la misma posición espacial con respecto a x_2. De igual manera que en el caso anterior, a partir de las Secciones 1D resultantes, se deben obtener Couplets 1D y así formar la Sección 2D resultante.

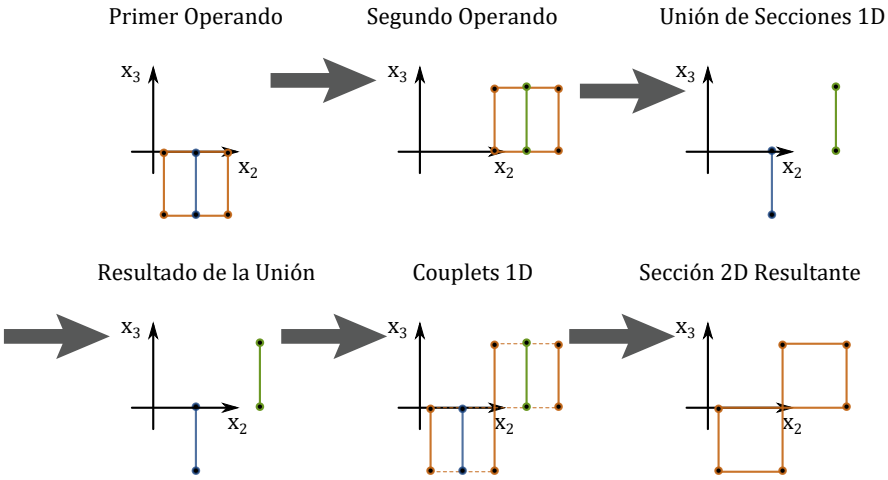


Figura 3.13: Unión de las segundas Secciones 2D de los operandos.

Como se observa en la figura 3.13, ahora se obtienen tres Couplets 1D, ello se debe a que las Secciones 1D resultantes se encuentran separadas con respecto al eje x_2. También se puede apreciar que el Couplet 1D que se encuentra en la parte media está formado por las dos Secciones 1D, lo cual se debe a que para obtener este Couplet se utiliza el algoritmo 3.9, y éste aplica la operación XOR.

Para el caso de la tercer Sección de la figura 3.11e, basándose en el algoritmo 3.11, solo se debe decidir si ésta será o no incluida en el EVM resultante. Esto debido a que para el segundo operando ya no se tienen más Secciones con las cuales seguir realizando la operación. Para el caso de la operación Unión sí se incluye la tercera Sección del primer operando en el EVM resultante.

Hasta este punto se han obtenido las Uniones entre las Secciones 2D de los dos operandos. Éstas forman la parte interna del 3D-EVM resultante, no obstante se deben obtener los Couplets 2D correspondientes. Esto debido a que para formar un nD-EVM solo se consideran los Couplets y las Secciones son descartadas, más sin embargo éstas son necesarias para obtener dichos Couplets

2D.

En la figura 3.14 se muestra cómo se obtienen los Couplets 2D (color azul) a partir de las Secciones 2D resultantes (color rojo). También se muestra el 3D-EVM resultante, el cual está formado con dichos Couplets 2D.

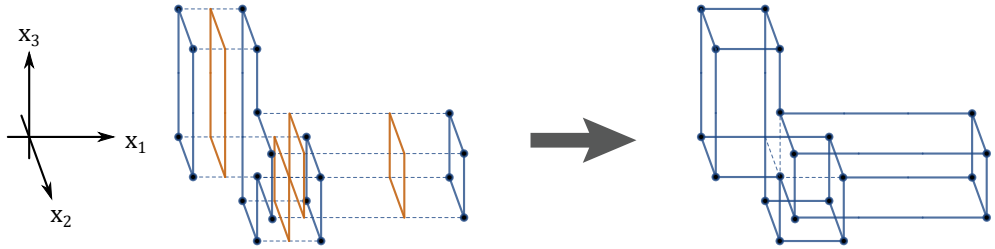


Figura 3.14: Obtención de Couplets 2D y formación del 3D-EVM resultante para la operación unión.

3.7.3.2. Aplicación de la Operación Intersección

En este apartado se presenta la aplicación de la operación intersección usando los operandos de la figura 3.10. Cabe mencionar que como resultado de esta operación, solo se consideran las partes de los 3D-EVMs que son comunes en ambos operandos y el resto es descartado.

Para las primeras Secciones 2D de los operandos que se muestran en las figuras 3.11a y 3.11b, se efectúa la operación intersección entre sus respectivas Secciones 1D. En la figura 3.15 se muestra el resultado de esta operación, en donde se observa que éste viene dado por la región común a ambas Secciones 1D. También, a partir de la Sección 1D resultante se obtienen los Couplets 1D para formar la Sección 2D resultante.

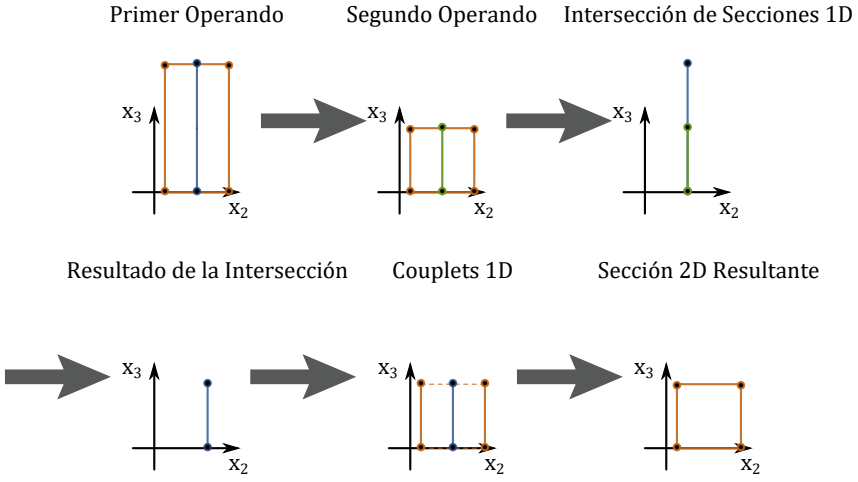


Figura 3.15: Intersección de las primeras Secciones 2D de los operandos.

Ahora, continuando con las siguientes Secciones 2D que se muestran en las figuras 3.11c y 3.11d, la operación intersección de éstas se realiza de manera ordenada considerando su posición espacial. Ello se debe a que dichas Secciones 2D se encuentran en diferentes posiciones con respecto al eje x_2. En la figura 3.16 se muestra el resultado de la intersección de las Secciones 2D de los dos operandos, en donde se observa que se obtienen como resultado una Sección vacía.

Para el caso de la tercera Sección 2D del primer operando que se muestra en la figura 3.11e, en base al algoritmo 3.11 nuevamente se debe considerar si se incluye en el 3D-EVM resultante, debido a que para el segundo operando no hay más Secciones que extraer. Entonces considerando que para la operación intersección se obtiene como resultado las regiones que son comunes entre ambos

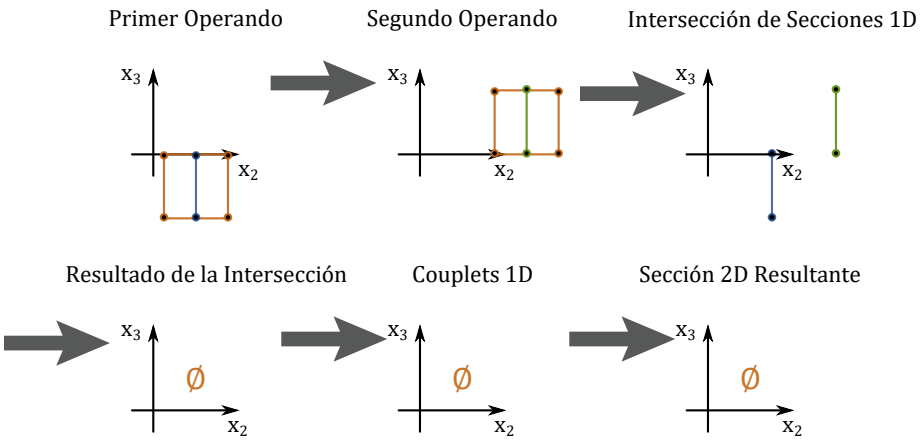


Figura 3.16: Intersección de las segundas Secciones 2D de los operandos.

operandos, la tercera Sección 2D del primer operando no será incluida. Esto porque, de manera general, al realizar la intersección de un objeto con un objeto vacío, se obtiene un objeto vacío.

Como resultado de la operación Intersección se obtuvo solo una Sección 2D que se muestra en la figura 3.15. Con ello se obtienen los Couplets 2D correspondientes, y de esta manera se forma el 3D-EVM resultante de la operación Intersección. En la figura 3.17 se muestra este resultado, en donde se observa que efectivamente se tiene solo la región que tienen ambos operandos en común.

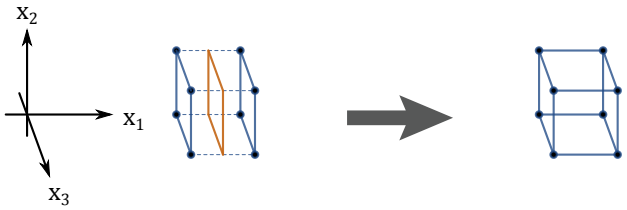


Figura 3.17: Obtención de Couplets 2D y formación del 3D-EVM resultante para la operación intersección.

3.7.3.3. Aplicación de la Operación Diferencia

En este apartado se presenta el análisis de la operación Diferencia entre los operandos de la figura 3.10. De manera general, la operación Diferencia consiste en tomar el primer operando y restarle el segundo. Es decir, que para el primer operando se eliminarán las regiones que tenga en común con el segundo operando. Esta operación es sensitiva al orden de los operandos, por lo que si éstos se invierten se obtendrá un resultado diferente.

Aplicando la operación Diferencia entre las primeras Secciones 2D de los operandos, que se muestran en las figuras 3.11a y 3.11b, se obtiene el resultado que se muestra en la figura 3.18. Como se observa, al aplicar la operación diferencia entre las Secciones 1D, se obtiene como resultado la región del primer operando que no es común al segundo operando. Posteriormente, se obtienen los Couplets 1D para así obtener la Sección 2D resultante.

Ahora, aplicando la operación diferencia entre las segundas Secciones 2D de los operandos de las figuras 3.11c y 3.11d, se obtiene el resultado que se muestra en la figura 3.19. En donde nuevamente, al momento de operar las Secciones 1D, se toma en consideración la posición espacial de las Secciones 2D con respecto al eje x_2. En este caso no existe una región en común entre ambas Secciones, por lo tanto el primer operando se conserva mientras que el segundo operando no es considerado. Posteriormente, a partir de la Sección 1D resultante, se realiza el cálculo de los Couplets 1D y así formar la Sección 2D resultante.

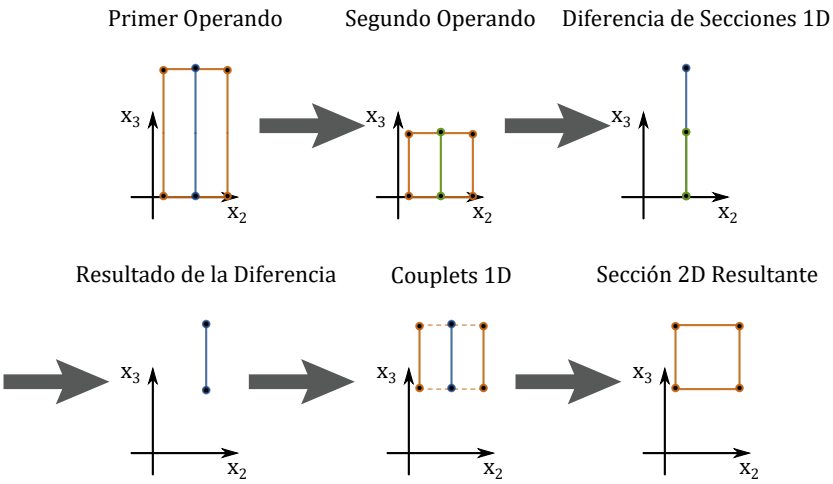


Figura 3.18: Diferencia de las primeras Secciones 2D de los operandos.

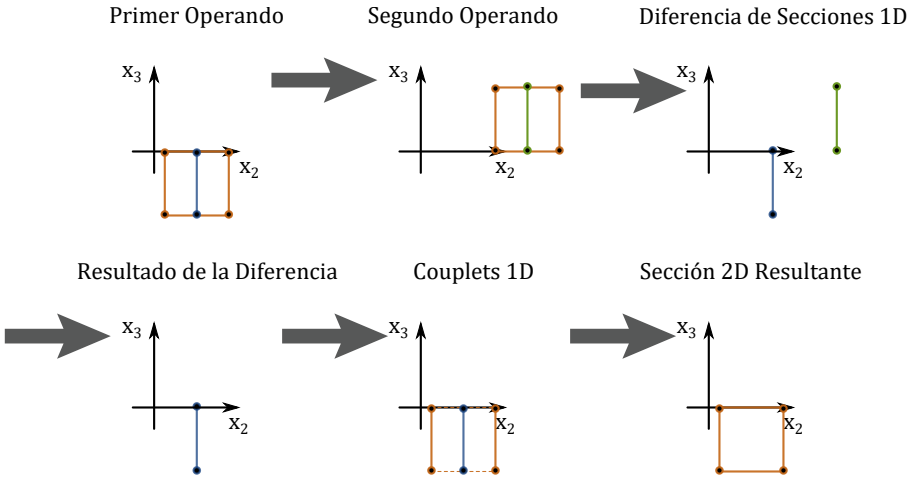


Figura 3.19: Diferencia de las segundas Secciones 2D de los operandos.

Para la tercera Sección del primer operando que se muestra en la figura 3.11e, y en base ala operación Diferencia, el algoritmo de Operaciones Booleanas decide que la última Sección 2D del primer operando será incluida en el resultado.

Por último, considerando las Secciones 2D resultantes, se obtienen los Couplets 2D que forman el 3D-EVM resultante. En la figura 3.20, se muestra el cálculo de los Couplets 2D y el 3D-EVM final. En donde se observa que la región que tienen en común los operandos originales es eliminada del resultado final.

3.7.3.4. Aplicación de la Operación XOR

En este apartado se presenta cómo se aplica el operador XOR a los operandos de la figura 3.10. De manera general, la operación XOR aplicada a dos operandos consiste en descartar las regiones que son comunes a ambos, pero en este caso si se conservan las demás regiones de ambos operandos. Es importante notar que esta operación puede ser efectuada mediante el algoritmo 3.7 (*mergeXOR*), pero también se puede efectuar mediante el algoritmo de Operaciones Booleanas (algoritmo 3.11). La diferencia entre éstos algoritmos radica en el modo de realizar esta operación, el algoritmo 3.7 realiza la operación sobre los vértices extremos, y el algoritmo 3.11 realiza la operación mediante el cálculo de Secciones y Couplets.

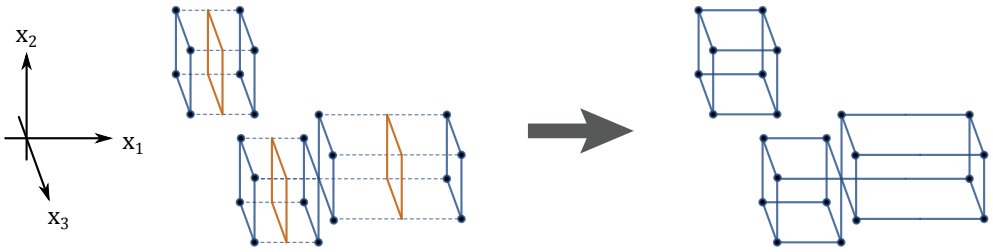


Figura 3.20: Obtención de Couplets 2D y formación del 3D-EVM resultante para la operación diferencia.

Al aplicar la operación XOR entre las primeras Secciones 2D de los operandos, que se muestran en las figuras 3.11a y 3.11d, se obtiene el resultado que se muestra en la figura 3.21. En donde se observa que al aplicar la operación XOR a las Secciones 1D, el resultado indica que se elimina la parte común entre éstas.

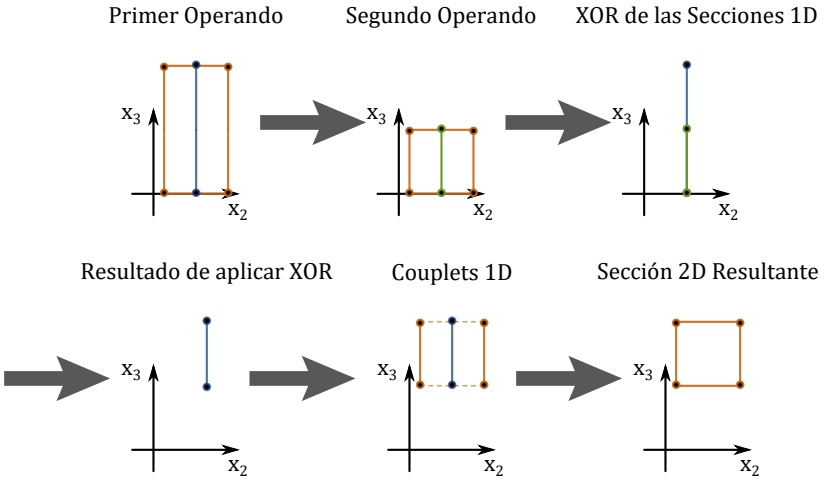


Figura 3.21: Operación XOR entre las primeras Secciones 2D de los operandos.

Por otro lado, para las siguientes Secciones 2D que se muestran en las figuras 3.11c y 3.11d, se obtiene como resultado la Sección 2D que se muestra en la figura 3.22. Como se observa, se obtienen dos Secciones 1D como resultado, ello se debe a que las Secciones 2D se encuentran en diferentes posiciones espaciales con respecto al eje x_2. Lo anterior implica que no existe ninguna región en común entre las Secciones 2D, y por ende ambas son incluidas en el resultado. En este caso, debido a que las Secciones 1D tienen diferentes posiciones espaciales, se forman tres Couplets 1D y así se forma una Sección 2D con dos hiper-cajas 2D.

Nuevamente, pasando al caso en donde se determina si se incluye o no la última Sección 2D del primer operando que se muestra en la figura 3.11e, se debe considerar que la operación XOR descarta regiones comunes. Entonces, al no haber más Secciones del segundo operando, sí se toma en consideración la última Sección 2D del primer operando, ya que no hay una región en común.

Partiendo de las Secciones 2D resultantes obtenidas anteriormente, se obtienen los Couplets 2D para formar el 3D-EVM resultante. En la figura 3.23 se muestran los Couplets 2D obtenidos a partir de las Secciones 2D resultantes, y también se muestra el 3D-EVM final. Como se observa, el resultado contiene las regiones de ambos operandos, pero la región que es común a éstos no es considerada.

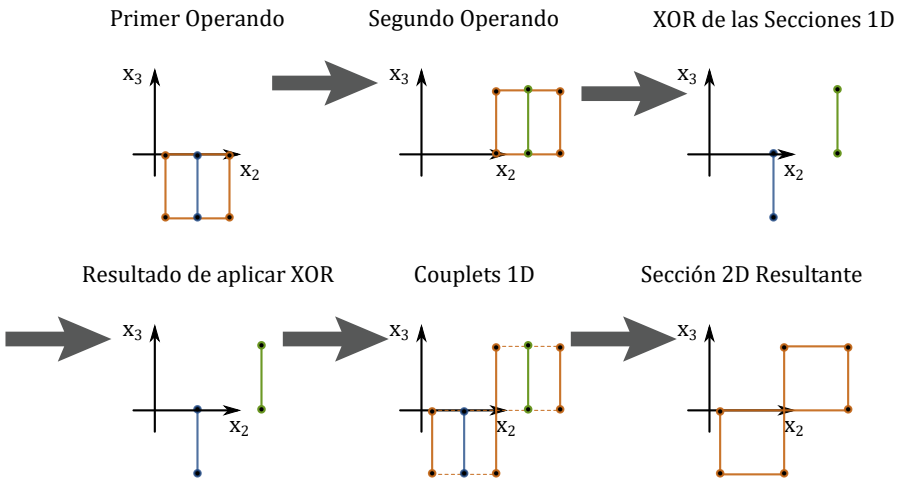


Figura 3.22: Operación XOR entre las segundas Secciones 2D de los operandos.

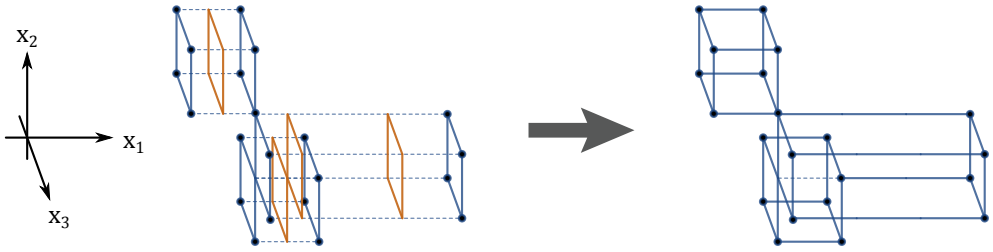


Figura 3.23: Obtención de los Couplets 2D y formación del 3D-EVM resultante para la operación XOR.

3.7.4. Algoritmo de Cálculo de Contenido n-Dimensional

Este algoritmo se utiliza para calcular el contenido de un EVM. En este sentido, el contenido consiste en la cantidad de espacio nD que contiene un EVM. Como se describe en [11], se parte de la idea de que para generar un hiper-prisma nD se requiere de un (n-1)D-OPP como su base y una altura.

Consideremos el caso en que se requiere generar un hiper-prisma nD P_n, su base es un (n-1)D-OPP P_{n-1} que tiene un contenido C_{n-1}, y la altura es h_n, entonces el contenido de P_n está dado por:

$$Content(P_n) = Content(P_{n-1}) \cdot h_n = C_{n-1} \cdot h_n \tag{3.1}$$

La ecuación 3.1 también puede ser aplicada para calcular el contenido del hiper-prisma P_{n-1}, ya que se requiere un (n-2)D-OPP P_{n-2} y una altura h_{n-1}. De esta manera se establece el cálculo de contenido de un hiper-prisma de manera recursiva, y el caso base se presenta cuando se tiene un 1D-OPP. El contenido de un 1D-OPP es la longitud del segmento 1D, que está dada por la distancia entre los dos puntos en la frontera.

De manera general la ecuación para calcular el contenido para un hiper-prisma nD se denota de la siguiente manera:

$$Content(P_n) = \begin{cases} h_1, & n = 1 \\ Content(P_{n-1}) \cdot h_n, & n > 1 \end{cases} \tag{3.2}$$

Para extender esta idea al cálculo el contenido de un espacio nD englobado por un nD-OPP se deben considerar los Slices que lo forman. Un Slice puede ser visto como un conjunto de uno o

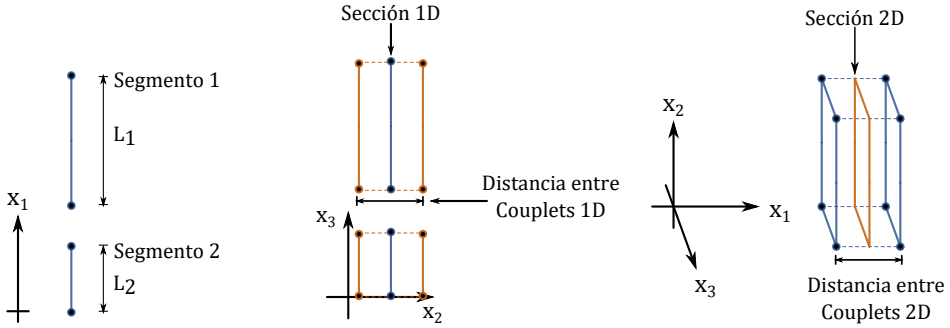
mas hiper-prismas nD, el cual tiene una Sección (n-1)D en la parte interna y puede ser considerara como su base, y tiene dos Couplets (n-1)D en la frontera.

Tomemos como ejemplo el cálculo del contenido de un 3D-OPP p ya que es sencillo visualizar su resultado. El contenido de p puede obtenerse de la suma de los volúmenes de sus Slices. Ahora, el volumen de un $Slice_k^i(p)$ está dado por el producto entre el área de la Sección $S_k^i(p)$ correspondiente y la distancia que separa los Couplets frontera $\Phi_k^i(p)$ y $\Phi_{k+1}^i(p)$, dicha distancia es la altura del Slice 3D.

Ahora supongamos que se tiene un 2D-OPP $q = S_k^i(p)$, el cual es una Sección de p y la base del Slice $Slice_k^i(p)$. El contenido de q (su área) es la suma de las áreas de los Slices 2D que lo forman. El área de un Slice 2D $Slice_k^i(q)$ está dado por el producto de la longitud de la Sección 1D $S_k^i(q)$ como su base, y la distancia entre los Couplets 1D $\Phi_k^i(q)$ y $\Phi_{k+1}^i(q)$.

Con lo anterior se obtiene el caso base que es para un 1D-OPP, y su contenido se calcula como la suma de las longitudes de sus brinks. En la figura 3.24 se ilustra un ejemplo de cómo se calcula el contenido para 1D-OPPs (3.24a), 2D-OPPs (3.24b) y 3D-OPPs (3.24c). La ecuación general para calcular el contenido de un nD-OPP p es la siguiente:

$$Content_n(P) = \begin{cases} Length(p), & n = 1 \\ \sum_{k=1}^{np_i-1} \left\{ Content_{n-1}\left(S_k^i(p)\right) \cdot \quad Dist\left(\Phi_k^i(p), \Phi_{k+1}^i(p)\right) \right\}, & n > 1 \end{cases} \tag{3.3}$$



(a) Cálculo de contenido para un 1D-EVM. **(b)** Cálculo de contenido para un 2D-EVM. **(c)** Cálculo de contenido para un 3D-EVM.

Figura 3.24: Cálculo de contenido 3D (volumen) en el EVM.

Ahora, con la ecuación 3.3 se forma el algoritmo de cálculo de contenido para nD-OPPs representados mediante un nD-EVM, el cual se muestra en el algoritmo 3.12. En este algoritmo se observa que se tiene una fase de definición e inicialización de las variables que se utilizan (líneas 2-4 y 7-9), en la línea 10 se obtiene el primer Couplet del nD-EVM con respecto al eje coordenado x_1, en la estructura de repetición $while$ de la línea 11 se obtiene el contenido de cada uno de los Slices y en la línea 14 se obtiene el contenido de la Sección del Slice (n-1)D. Como se observa este algoritmo se ejecuta de manera recursiva hasta obtener los casos base, que son para 1D-EVMs tal como se muestra en la línea 5.

3.7.5. Algoritmo de Cálculo de Compacidad Discreta

La Compacidad Discreta (*Discrete Compactness*, DC) puede ser utilizada como descriptor de forma para un nD-EVM. De esta manera, es posible caracterizar a un EVM y distinguirlo entre otros. Este descriptor presenta una mejora con respecto al descriptor clásico de Compacidad de Forma, el cual presenta variaciones significativas ante variaciones en la forma de los objetos [23, 37].

Algoritmo 3.12: Algoritmo de cálculo de contenido de un nD-EVM.

```
   Input  : Un nD-EVM p y la dimensión n.
   Output: Contenido de p.
1  Procedure Content(nD-EVM p, n)
2      contSum ← 0;
       /* Inicialización de objetos                                          */
3      EVM couplet1, couplet2;
4      EVM section;
5      if n == 1 then
6          return Length(p);
7      couplet1 ← InitEVM();
8      couplet2 ← InitEVM();
9      section ← InitEVM();
10     couplet1 ← readCouplet(p);
11     while !endEVM(p) do
12         couplet2 ← readCouplet(p);
13         section ← getSection(p);
14         contSum ← contSum + Content(section,n − 1) * dist(couplet1,couplet2);
15         couplet1 =couplet2;
16     return contSum
```

En este apartado es necesario introducir el concepto de hiper-voxelización [10, 11], debido a las condiciones que se requieren en los nD-EVMs para realizar apropiadamente el cálculo de DC. Una hiper-voxelización consiste en representar un politopo en base a un esquema de enumeración de hiper-espacio ocupado (*Hyperspatial Occupancy Enumeration*), en donde se forma la lista de celdas en el hiper-espacio (del mismo tamaño) que ocupa dicho politopo. Estas celdas son llamadas hiper-voxels, y son hiper-cajas de un tamaño fijo que se encuentran en un espacio n-Dimensional.

En este sentido, un hiper-voxel en 2D es un píxel, un hiper-voxel en 3D es un voxel, y para el caso 4D se recomienda utilizar el término *rexel*. Una colección de hiper-voxels se puede codificar mediante un arreglo de dimensionalidad n $C_{x_1,x_2,\ldots,x_n}$, en el cual se define si un hiper-voxel se encuentra ocupado (*Black*) o vacío (*White*) cuando $C_{x_1,x_2,\ldots,x_n} = 1$ y $C_{x_1,x_2,\ldots,x_n} = 0$ respectivamente. Se debe notar que solo se especifica una coordenada para representar un hiper-voxel, para este trabajo se considera la coordenada inicial.

Con lo anterior, se define un politopo mediante la colección de los hiper-voxels ocupados que éste engloba en el hiper-espacio. Es importante considerar que los hiper-voxels son disjuntos o cuasi-disjuntos, es decir que no se traslapan. Debido a que un conjunto de hiper-voxels ocupados representan un nD-OPP p se tiene lo siguiente[10, 11],:

$$p = \bigcup_{\lambda} BlackHypervoxel_{\lambda} \tag{3.4}$$

Para cada hiper-voxel ocupado se pueden generar 2^n vértices extremos, por lo que la correspondiente representación en el nD-EVM de p se obtiene de la siguiente manera:

$$EVM_n(p) = EVM_n\big(\bigcup_{\lambda} BlackHypervoxel_{\lambda}\big) = \otimes_{\lambda} EVM_n(BlackHypervoxel) \tag{3.5}$$

Para el cálculo de DC es importante considerar que los nD-OPPs son el resultado de una hiper-voxelización, en donde los hiper-voxels son hiper-cubos unitarios nD, y el espacio nD consiste de ejes con coordenadas enteras. En el caso 3D se considera que un objeto está formado por un conjunto de cubos tal como se describe en [22, 38].

Por lo anterior, para el uso adecuado del descriptor DC se requiere que el espacio nD de los nD-OPPs esté formado por ejes coordenados con valores enteros. Entonces, un nD-OPP está formado

por la unión de hiper-cubos disjuntos o cuasi-disjuntos, es decir que no se traslapan, tal como se describe en [10, 11]. Con ello, suponiendo que se tienen dos nD-OPPs p y q con dichas propiedades, y considerando su representación en el EVM, entonces la unión de éstos está dada por:

$$EVM_n(p \cup q) = EVM_n(p) \otimes EVM_n(q)$$

Como se observa para este caso particular, la operación Unión es equivalente a la operación XOR en el sentido que los vértices comunes serán eliminados porque son vértices internos. Con estas premisas establecidas, y considerando que aplican para el algoritmo de cálculo de Contenido es posible establecer el cálculo de DC para nD-OPPs.

En [22, 38] se presentan los fundamentos para el cálculo de compacidad discreta para objetos 2D y 3D respectivamente. En estos trabajos de investigación se plantean casos específicos, y se proponen reglas que facilitan el cálculo de DC.

Para el caso general, en [15] se describe que para calcular la DC para un objeto nD, se utiliza la siguiente ecuación:

$$DC = \frac{L_c - L_{Min}}{L_{Max} - L_{Min}} \tag{3.6}$$

En donde:

- L_c: Cantidad de contactos internos del objeto.
- L_{Min}: Es la cantidad mínima posible de contactos internos considerando un objeto nD con la misma cantidad de hiper-voxels.
- L_{Max}: Cantidad máxima posible de contactos internos considerando un objeto nD y la misma cantidad de hiper-voxels.

Con lo anterior, para el cálculo de DC se debe especificar una cota superior y una inferior. Estas cotas se especifican en base a la aplicación que se desea abordar. No obstante es necesario describir el algoritmo para el cálculo de contactos internos.

Los contactos internos de un objeto nD-EVM *p*, son la cantidad de celdas (n-1)D que están compartidas entre los hiper-voxels que constituyeron originalmente al politopo ahora expresado en el EVM. Para el cálculo de contactos internos, se realiza un procesamiento de los Slices (rebanadas) del EVM. Este procesamiento se realiza para cada eje coordenado y de manera secuencial. Entonces es posible obtener la cantidad de hiper-voxels de cada Slice, esto mediante el cálculo de contenido de la Sección y la separación entre los Couplets correspondientes a dicho Slice.

Entonces dado un nD-EVM *p* y el *j*-ésimo $Slice_j^i(p)$ con respecto eje coordenado *i*, se realizan los siguientes pasos:

- Se obtiene la cantidad de coordenadas internas del Slice con respecto al eje coordenado *i*. Es decir la longitud del Slice con respecto al eje *i*, pero sin considerar las coordenadas finales en ambos extremos. Con ello, dadas las coordenadas c_1 y c_2, de los Couplets inicial y final respectivamente, que acotan al Slice con respecto al eje *i*, se obtiene la cantidad de coordenadas internas como:
$$nCoords = c_2 - c_1 - 1$$
- Ahora, para calcular los contactos internos de $Slice_j^i(p)$ se utiliza la siguiente ecuación:

$$IC = nCoords * Content(Section_j)$$

En donde *Content* hace referencia al algoritmo de Contenido que se muestra en 3.12. Como se mencionó anteriormente, solo se consideran las coordenadas internas de cada Slice. Ello se debe a que existe la posibilidad que el EVM contenga Slices aislados, o que solo ciertas regiones estén compartidas entre Slices. Por ello el siguiente paso es hallar los contactos internos entre los Slices. Para ello se requieren dos pasos adicionales:

- Realizar la operación intersección entre Slices consecutivos utilizando el algoritmo 3.11.
- Obtener el contenido del resultado de la intersección anterior.

Con esto se garantiza que se estén considerando las regiones compartidas entre los Slices. En el algoritmo 3.13 se muestra el procedimiento genérico para obtener la cantidad de contactos internos de un nD-EVM.

Algoritmo 3.13: Algoritmo de cálculo de contactos internos de un nD-EVM.

```
   Input : Un nD-EVM p y la dimensión n.
   Output: Cantidad de contactos internos perpendiculares al primer eje coordenado.
1  Procedure InternalContacts(nD-EVM p, n)
2      EVM couplet,currentSection,previousSection ;          // Elementos que describen a un Slice
3      EVM sectionInt ;                                      // Resultado de la intersección
4      integer c1,c2 ;                                       // Coordenadas del Couplet previo y actual
5      integer nCoords ;                                     // Total de coordenadas internas
       // Inicialización de variables
6      integer ic ← 0 ;                                      // Total de contactos internos
7      previousSection ← InitEVM();
8      c1 ← getCoord(p);
9      couplet ← readCouplet(p);
10     while !endEVM(p) do
           // Cálculo de contactos internos del Slice actual
11         currentSection ← getSection(previousSection,couplet);
12         c2 ← getCoord(p);
13         nCoords ← c2- c2- 1;
14         ic ← ic + nCoords * Content(currentSection,n −1);
           // Cálculo de contactos internos entre Slices
15         sectionInt ← BooleanOperation(previousSection,currentSection,"intersection",n − 1);
16         ic ← ic + Content(sectionInt,n − 1);
17         previousSection ← currentSection;
18         c1 ← c2;
19         couplet ← readCouplet(p) ;                        // Leer el siguiente Couplet
20     return ic
```

Como se observa, en el algoritmo 3.13 se describe el procedimiento para calcular los contactos internos de un EVM, perpendiculares al primer eje coordenado. Por lo que es necesario realizar este proceso para los demás ejes coordenados, y así obtener los contactos internos totales.

Lo anterior se logra mediante combinaciones en el ordenamiento de los ejes coordenados. Con ello, el conjunto de posibles ordenamientos de ejes coordenados para un nD-EVM es el siguiente:

$$AxisSorting = \big\{\{x_1, x_2, \ldots, x_{n-1}, x_n\}, \{x_2, x_3, \ldots, x_n, x_1\}, \{x_3, x_4, \ldots, x_1, x_2\}, \ldots, \{x_n, x_1, \ldots, x_{n-2}, x_{n-1}\}\big\}$$

Es sencillo determinar que la cardinalidad del conjunto *AxisSorting* es n, con lo que se obtienen n diferentes combinaciones a las que se tiene que aplicar el algoritmo de contactos internos. Posteriormente, con la suma acumulada de los contactos internos se obtiene el total de éstos. Por lo anterior, se define el algoritmo de *Contactos Internos Totales* en 3.14.

Suponiendo que el conjunto $AxisSorting$ se encuentra ordenado, la función $sortEVM$, que se muestra en la línea 4 del algoritmo 3.14, se utiliza para obtener el siguiente ordenamiento de los

Algoritmo 3.14: Algoritmo de cálculo de contactos internos totales de un nD-EVM.

```
   Input  : Un nD-EVM p y la dimensión n.
   Output: Cantidad de contactos internos totales.
1  Procedure TotalInternalContacts(nD-EVM p, n)
2      integer ic ← 0 ;                                   // Contactos internos totales
3      for each sorting ∈ AxisSorting do
4          SortedP ← sortEVM(p,n,sorting);
5          ic ← ic + InternalContacts(SortedP,n)
6      return ic
```

ejes coordenados para el nD-EVM p diferente al actual. Con ello se exploran todos los posibles ordenamientos que se requieren.

Capítulo 4

Mapas Auto-Organizados de Kohonen

Organización del Capítulo

En este capítulo se presenta la teoría básica sobre mapas auto-organizados de Kohonen. Después del material de introducción de la Sección 4.1, en la Sección 4.2 se presentan los fundamentos del modelo de Kohonen, en donde se describen los procesos de competencia, cooperación y adaptación. Por último, se describe el proceso de entrenamiento.

4.1 Introducción

Un *mapa auto-organizado* (*Self-Organizing Map*, SOM) es una clase de *red neuronal artificial*, cuyo proceso de aprendizaje se basa en la competencia de sus elementos de procesamiento que son las neuronas [16, 17, 18]. El *proceso de aprendizaje* consiste en realizar un ajuste de pesos sinápticos para cada neurona. Los *pesos sinápticos* definen la relación entre el vector de entrada y las neuronas. En un SOM, el conjunto de patrones de entrada constituyen el *espacio del mapa*.

Mediante el proceso de aprendizaje, las neuronas se ajustan hacia varias clases de patrones, con lo cual, las neuronas ordenan sus posiciones espaciales, generando así un *sistema coordenado* de las diferentes características intrínsecas de dichos patrones. Este sistema coordenado, constituye la estructura interna del espacio de los patrones de entrada, también conocido como *Mapa* o *Esquema del Espacio*, en donde las neuronas son indicadores de características intrínsecas de los patrones.

Una analogía ilustrativa se efectúa en [39], al considerar que el espacio de entrada es particionado en regiones, en donde cada neurona del SOM se especializa en una región. Es decir, cada neurona identifica los patrones que corresponden a su región en cuestión.

En el *aprendizaje competitivo*, las neuronas compiten entre ellas para ser activadas, por ello, solo una neurona es seleccionada como la *neurona ganadora*[19]. Este aprendizaje está inspirado en el mecanismo de *auto-organización* del cerebro humano, específicamente de la corteza cerebral. Dicho mecanismo consiste en procesar las diferentes actividades e impulsos sensoriales, que son de naturaleza similar (p. ej. tacto, habla y visión), a través de regiones (de la corteza cerebral) que se encuentran próximas o vecinas.

En este sentido, una gran cantidad de investigadores han realizado trabajos experimentales, y han obtenido resultados prometedores [40, 41]. Actualmente, se conoce que muchas de las estructuras neuronales de la corteza cerebral, tienen topologías unidimensionales (1D) y bidimensionales (2D). Por lo anterior, en general, para los SOMs se utilizan estructuras 1D y 2D en sus arreglos de neuronas. Se pueden utilizar estructuras multi-dimensionales, sin embargo no son muy comunes.

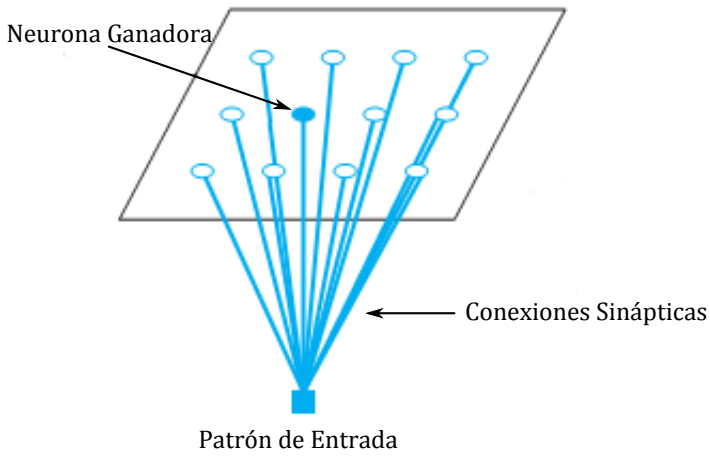


Figura 4.1: Ejemplo del modelo de Kohonen para un arreglo de neuronas bidimensional.

Existen varios modelos para SOMs , por ejemplo el modelo de *Willshaw* y *Malsburg* [17], sin embargo el modelo propuesto por el finlandés *Teuvo Kohonen* en 1982, ha tenido mayor aceptación. Este modelo también es conocido como *Mapa Auto-Organizado de Kohonen* (*Kohonen Self-Organized Map*, KSOM) [19, 16]. En la figura 4.1 se ilustra un ejemplo básico del modelo de Kohonen para un arreglo de neuronas en 2D. Este modelo toma características fundamentales de los mapas de procesamiento formados en el cerebro, y aún así, es computacionalmente tratable. Por ello se consideran tres aspectos importantes para su implementación:

- Definición de una vecindad.
- Las neuronas dentro de una vecindad tendrán conexión con otras neuronas que formen parte de la misma vecindad.
- Los arreglos de neuronas tienen una topología 1D o 2D.

4.2 El Modelo de Kohonen

En el modelo de Kohonen, el algoritmo para la formación del SOM, considera como paso inicial, la inicialización de pesos sinápticos con valores pequeños y aleatorios. Consecuentemente, se consideran tres procesos importantes: *competencia*, *cooperación* y *adaptación*.

El proceso de competencia consiste en utilizar una regla de similitud, que es un indicador para asociar un patrón a una neurona en particular. La regla de similitud clásica en la teoría de redes de Kohonen es la *distancia Euclidiana* [19, 16], no obstante se pueden utilizar otras reglas, tal como la Compacidad de Forma (*Shape Compactness*) que sirve para comparar qué tanto se parece un objeto (en el espacio multidimensional) a otro [22, 23]. Sin embargo, esta métrica es sensitiva a pequeños cambios en la forma (variaciones en la información) [42, 23]. Por otro lado, la Compacidad Discreta (*Discrete Compactness*) proporciona un criterio de comparación más robusto y es menos sensitivo a pequeñas variaciones en la información [15]. Esta similitud es un indicador usado para competir entre neuronas, con lo que se selecciona una neurona como la ganadora.

Mediante la cooperación, se definen las *relaciones de vecindad* entre neuronas. Es decir, que cuando una neurona es activada, se efectúa una actualización de la misma y también de las neuronas involucradas en su vecindad. Con ello, se permite la cooperación entre neuronas, para procesar los patrones de entrada, con características o ubicaciones espaciales similares. La adaptación no es más que un ajuste de pesos sinápticos en la neurona ganadora, y también de las neuronas en su vecindad.

4.2.1. Proceso de Competencia

Consideremos el modelo de Kohonen para construir el mapa de un espacio *n*-Dimensional. Para este problema se tendrá un conjunto de *m* neuronas. El vector, asociado a un patrón de entrada, está compuesto por *n* elementos:

$$\boldsymbol{X} = [x_1, x_2, \ldots, x_n]^T \tag{4.1}$$

Ahora, el *vector de pesos sinápticos* asociado a la *j*-ésima neurona, consiste de *n* elementos, uno para cada componente del vector de entrada. Se definen los vectores de pesos para las *m* neuronas de la siguiente manera:

$$\boldsymbol{W}_j = [w_{j,1}, w_{j,2}, \ldots, w_{j,n}]^T \ , \qquad j = 1, 2. \ldots, m \tag{4.2}$$

Retomando la noción de que las neuronas definen regiones en el espacio de entrada, el vector de pesos $\boldsymbol{W}_j$ puede ser visto como el *centro de la región* asociada a la neurona *j*. Ahora, a partir de los vectores $\boldsymbol{X}$ y $\boldsymbol{W}_j$, es posible realizar una comparación de similitud del patrón de entrada y la neurona *j*. Debido a que las regiones en el mapa son disjuntas, solo se seleccionará una sola *neurona ganadora*, esta propiedad se conoce como *"El ganador se lo lleva todo"* (*"The Winner takes all"*) [19, 16]. Con esto, la neurona ganadora será aquella que tenga la menor distancia Euclidiana entre su vector de pesos y el patrón de entrada $\boldsymbol{X}$. Por lo anterior, se define la función del índice de la neurona ganadora de la siguiente manera:

$$i(\boldsymbol{X}) = \arg \min_j \parallel \boldsymbol{X} - \boldsymbol{W}_j \parallel \tag{4.3}$$

4.2.2. Proceso de Cooperación

Ahora, es importante definir cómo se establecen las *relaciones de vecindad*. Para ello, la noción de vecindad está definida con base en la *distancia entre neuronas*. Por ejemplo, la *vecindad de radio r* asociada a la *k*-ésima neurona, en donde $1 \leq k \leq m$, consiste en el conjunto de neuronas que se encuentran a una distancia *r* como máximo. En el caso de un arreglo de neuronas 1D, la vecindad de radio *r*, contempla las neuronas ubicadas en *r* posiciones, como máximo, a la derecha o izquierda de la neurona ganadora. Lo que quiere decir que, la vecindad es de *naturaleza simétrica*.

La vecindad centrada en una neurona, indica la relación entre las neuronas en función a la distancia (radio). Es decir, que la neurona tiene una relación más fuerte con neuronas más cercanas. Esta *fuerza de relación* tiende a disminuir cuando la distancia aumenta, es decir con neuronas más lejanas. Por tanto, la *fuerza de la vecindad* tiene su punto máximo cuando la distancia es cero, en la neurona ganadora. Y, tiene un valor cero cuando la distancia tiende a infinito. Lo anterior se sustenta en aspectos neurobiológicos del cerebro humano, relacionados a la forma en que las neuronas son excitadas por impulsos sensoriales [43, 44, 45].

Una buena opción para satisfacer la necesidad anterior, es utilizar una *función Gaussiana*. Con la cual, la función de la fuerza de la vecindad decaerá a distancias mas grandes, teniendo como punto máximo en la neurona ganadora. No obstante, debido a que la idea original consiste en que las neuronas se especialicen en características intrínsecas de los patrones de entrada, esto equivale a que los vectores de pesos de las neuronas se distribuyan uniformemente en el espacio.

Por lo anterior, la vecindad de las neuronas debe ir decreciendo conforme a las iteraciones en el proceso de entrenamiento. Por ello, la función de fuerza de la vecindad debe estar en función de las iteraciones del proceso de entrenamiento. Esto se logra, como se argumenta en [16, 17, 18], haciendo que la distancia efectiva, de la función (Gaussiana) de fuerza de vecindad, disminuya en función de las iteraciones (tiempo). Por tanto, la función de fuerza de la vecindad se puede definir de la siguiente manera:

$$h\left(j, i(\boldsymbol{X}), d_{j,i}, t\right) = exp\left(-\frac{d_{j,i}^2}{2\sigma(t)^2}\right) \tag{4.4}$$

En donde *j* es la neurona a la cual se le calcula la fuerza de enlace. $i(\boldsymbol{X})$ indica qué neurona es la ganadora. $d_{j,i}$ es la distancia entre la neurona *j* y la neurona ganadora *i*. $\sigma(t)$ es la distancia efectiva de la fuerza de vecindad, la cual indica el grado en que las neuronas vecinas son activadas. Como se observa, este parámetro depende del tiempo, ello se debe a que la fuerza de la vecindad debe ir decreciendo con el tiempo. Por lo tanto, $\sigma(t)$ es de naturaleza decreciente. Y por último, *t* es la iteración actual en el proceso de entrenamiento.

4.2.3. Proceso de Adaptación

Este proceso contempla la actualización de *pesos sinápticos*. Entonces, el vector de pesos de cada neurona es actualizado mediante la siguiente regla:

$$\boldsymbol{W}_j(t+1) = W_j(t) + \alpha \cdot h\left(j, i(\boldsymbol{X}), d_{j,i}, t\right) \cdot (\boldsymbol{X} - \boldsymbol{W}_j) \tag{4.5}$$

Esta regla es aplicada a todas las neuronas en la vecindad de la neurona ganadora. Con ello, no solo se actualiza el vector de pesos de la neurona ganadora, sino que se actualizan los vectores de pesos de las neuronas es su vecindad. Es decir, que si el vector de pesos de la neurona ganadora es atraído hacia un patrón de entrada, los vectores de pesos de las neuronas vecinas también son atraídos, pero con menor intensidad. Este efecto de atracción es moderado por el producto $\alpha \cdot h\left(j, i(\boldsymbol{X}), d_{j,i}, t\right)$, en donde α es la tasa de aprendizaje. La *tasa de aprendizaje* puede tener un valor constante, pero también puede ir decreciendo con el tiempo. Con esto, se mantiene el efecto de decrecimiento de la fuerza de vecindad en el proceso de formación del SOM.

4.2.4. Algoritmo de Entrenamiento

Los procesos de formación del SOM, descritos en las secciones anteriores, constituyen el *algoritmo de entrenamiento*, el cual se muestra de manera genérica en el algoritmo 4.1 y se describe continuación de manera resumida:

1. **Inicialización**. Se definen e inicializan las variablea a utilizar, y se inicializa de manera aleatoria la matriz de pesos sinápticos, véase las líneas 2-6.
2. **Iteraciones**. El conjunto de patrones de entrenamiento es usado para realizar los ajustes en el SOM, lo cual se realiza un número determinado de iteraciones, tal como se observa en la estructura de repetición *for* de la línea 7.
    a) Para cada patrón $\boldsymbol{X}$ en el conjunto de entrenamiento (línea 8), se realiza lo siguiente:
        1) **Neurona ganadora**. Se identifica a la neurona ganadora $i(\boldsymbol{X})$, lo cual se realiza usando el criterio de la distancia mínima planteado en la ecuación 4.3 (línea 9).
        2) **Ajuste de pesos**. Se realiza el ajuste de los vectores de pesos de todas las neuronas (en especial las neuronas en la vecindad), mediante la regla de actualización formulada en la ecuación 4.5 (línas 13-15).

Una vez que se ha formado el SOM, al introducir un patrón cualquiera (que no este en el conjunto de entrenamiento), se puede realizar una consulta para identificar la región o clase en el mapa correspondiente a dicho patrón. Esta consulta se realiza mediante la búsqueda de la neurona cuyo vector de pesos tiene la mínima distancia Euclidiana hacia el patrón de entrada.

Algoritmo 4.1: Algoritmo de entrenamiento para una red de Kohonen.

```
Input : El número de neuronas m, la dimensionalidad de los patrones n, el número máximo de
        iteraciones y el conjunto de entrenamiento dataSet.
Output: La matriz de pesos sinápticos del SOM entrenado.
1  Procedure SOMTraining(integer m, integer n, integer iterations, Set of reals dataSet)
2      integer j, k, winningNeuron;
3      real neighborhoodDistance, pattNeuronDistance;
4      Matrix of reals weightMatrix ;                                  // Matriz de pesos sinápticos
5      real array nWeight,wWeight ;          // Pesos de la neurona a actualizar y la neurona ganadora.
       /* Inicializar la matriz de pesos sinápticos con base en la cantidad de neuronas m y las dimensiones k.  */
6      wieghtMatrixInit(weightMatrix,m,n);
       /* Presentaciones del conjunto de datos de entrenamiento.                                              */
7      for t = 0 to iterations do
8          for each x ∈ dataSet do
9              winningNeuron ← arg mín_j ‖ x − −weightMatrix[j] ‖ ;
10             for j = 0 to m do
11                 nWeight ← weightMatrix [j];
12                 wWeight ← weightMatrix [winningNeuron ];
13                 pattNeuronDistance ← euclideanDistance(nWeight,wWeight);
14                 neighborhoodDistance ← neighborhood(nWeight,wWeight,pattNeuronDistance,t);
15                 weightMatrix [j] ← nWeight + α· neighborhoodDistance ·(x − nWeight)
16     return weightMatrix;
```

Un aspecto importante está relacionado con la distribución del conjunto de entrenamiento. Si éste no está *uniformemente distribuido*, los patrones de entrenamiento tienden a formar anidamientos en el espacio n-D. Lo cual provoca que los vectores de pesos en el SOM, no puedan distribuirse de manera uniforme en el espacio de entrada. Por ello, es necesario una redistribución del conjunto de patrones de entrenamiento, tal como se describe en [39].

Parte III

Desarrollo

Capítulo 5

Procesamiento de video usando el modelo nD-EVM

5.1 Representación de Secuencias de Frames

En esta sección se presentan los fundamentos necesarios para realizar la representación de un video, dado en forma secuencia de frames, en el modelo nD-EVM. Este proceso de representación, básicamente consiste en las etapas que se muestran en la figura 5.1, y que se describen a continuación:

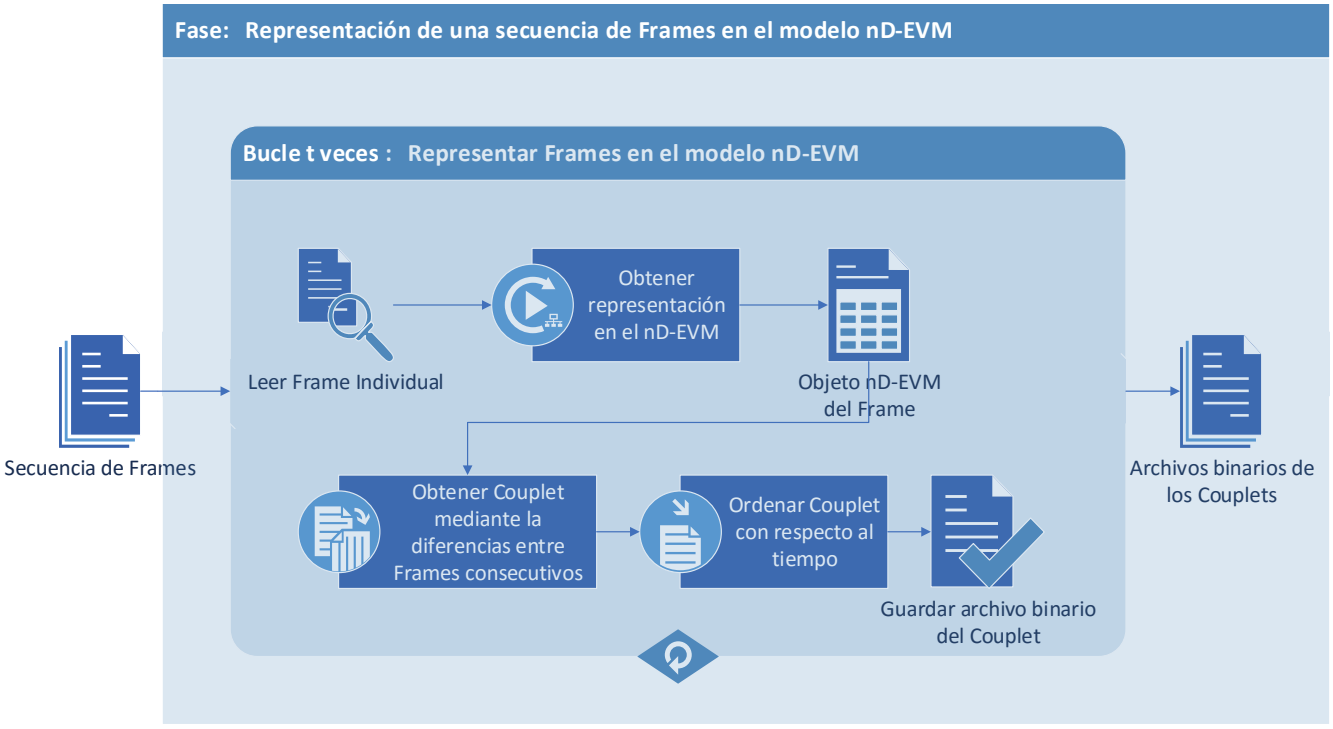


Figura 5.1: Etapas del proceso de representación de secuencias de frames.

- Para cada frame se debe obtener su representación en el modelo nD-EVM, lo cual se realiza mediante una extrusión para cada componente de color.
- Posteriormente, se computan las diferencias a través de la operación XOR entre frames consecutivos, lo cual es importante porque solo se desean almacenar las diferencias entre los frames. Con ello se reduce la cantidad de información que se almacena. De esta manera, los resultados de las diferencias forman los Couplets del EVM resultante. No obstante es posible obtener los frames originales, ello se logra obteniendo las Secciones de dicho EVM.
- Una vez obtenido un Couplet este se almacena en disco. Ello debido a que se desea optimizar el uso de memoria RAM en la PC donde se ejecute este proceso. De esta manera, se obtiene un archivo binario para cada Couplet, el cual contiene la información de todos los vértices que se encuentran en él.

Como se observa, básicamente son tres etapas principales para realizar la representación de una secuencia de frames. Por ello, en las siguientes secciones se describe cada una de ellas, con la finalidad de presentar los fundamentos que se consideraron para su desarrollo.

Es importante mencionar que, el desarrollo del framework propuesto no considera una etapa de extracción de secuencias de frames a partir de un video. Esto con base en que existen diversas herramientas para realizar dicho proceso, con las cuales es posible utilizar videos prácticamente de cualquier formato, y por ello este trabajo no considera un formato de video en particular. Unos ejemplos de herramientas gratuitas para procesar videos son las siguientes: *OpenCV*, *FFmpeg* y *Blender*. Por otro lado también existen herramientas de uso comercial como *MATLAB* y *Photoshop*

También es importante mencionar que los algoritmos propuestos para la representación de frames no consideran un formato de imagen en particular, ya que se plantea que hay librerías específicas para cada formato de imagen, las cuales tienen métodos que permiten obtener la información de los píxeles.

5.1.1. Representación de Frames

Para la representación de frames, se toma como premisa que éstos son imágenes convencionales. Para este caso se consideran imágenes con formato BMP, pero en realidad se puede utilizar cualquier formato. De esta manera, se considera que las imágenes consisten en un conjunto de píxeles ordenados en un plano 2D (x, y), en donde cada píxel cuenta con un color específico.

Actualmente existen varios modelos de color que se pueden utilizar, algunos de ellos son: Escala de Grises, RGB y CMYK. Por ejemplo, para el modelo RGB se tienen tres componentes de color: Rojo, Verde y Azul. Para el modelo CMYK se tienen cuatro componentes: Cyan, Magenta, Amarillo y Negro (*Key*). Por último para el modelo de escala de grises solo se cuenta con un componente de color. No obstante, hay otras variantes de éstos, en donde se agrega un componente adicional, como la transparencia (*Alpha*).

Para el caso del modelo de color RGB, un esquema muy conocido es el *RGB24*. El cual consiste en que cada componente de color cuenta con 8 bits de información para su representación, con lo cual se obtienen 24 bits en total. Con ello, se tienen 256 códigos o niveles en cada componente, con lo que se obtienen aproximadamente 16 millones de combinaciones diferentes entre los tres componentes, por ende esa es la cantidad de colores que se pueden representar con este modelo.

Para representar frames en el EVM se realiza una extrusión en cada componente de color. Es decir, que para un frame se toma como base la región 2D definida en este, y se agrega una dimensión adicional por cada componente de color, así se forman politopos en mas de dos dimensiones.

Con base en lo anterior, para frames bajo el esquema de escala de grises, se obtiene un politopo en 3D, ya que solo hay un componente de color. Ahora, para frames bajo el esquema RGB se obtienen politopos en 5D, ya que se cuentan con tres componentes de color. No obstante, es importante considerar que para animaciones se requiere una dimensión adicional para la variable temporal. Entonces un video en RGB se representa a través de un 6D-EVM.

Cabe mencionar que para frames bajo el esquema RGB se están considerando componentes separados. Esto es diferente a la forma en que normalmente se maneja la representación de un color específico, ya que convencionalmente se maneja un valor entero tomando en cuenta los 24 bits en conjunto. Es decir, que se concatenan los 8 bits de cada uno de los tres componentes de color, como se muestra en la figura 5.2.

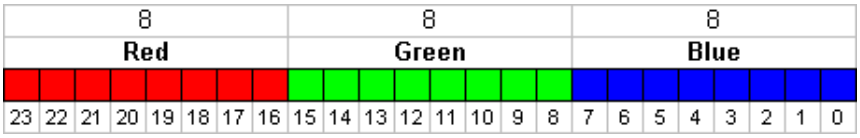
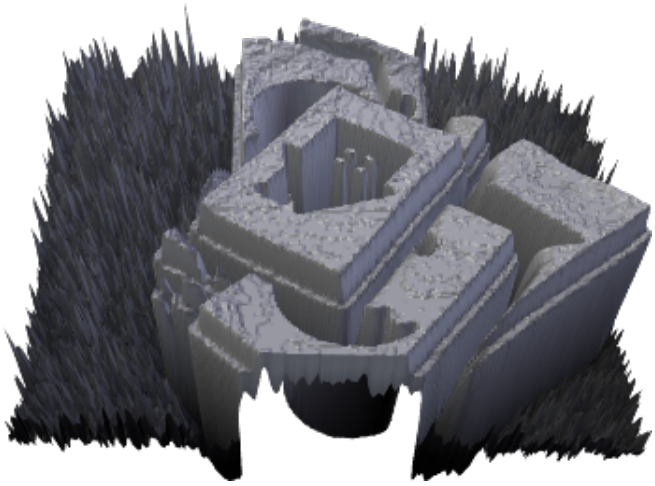


Figura 5.2: Esquema de color RGB24.

En [3] se presentan los fundamentos para la representación de video a color en el nD-EVM. En donde un frame a color es visto como un 3D-OPP, por ende se representa a través de un 3D-EVM. Una desventaja de este enfoque consiste en que el color es considerado como una sola dimensión. Para el esquema RGB24 de la figura 5.2, el color se representa como un valor entero con 24 bits, lo cual implica que para pequeñas variaciones en los bits mas significativos se producen grandes variaciones en los valores enteros. Es por ello que en este trabajo de investigación se optó por separar cada uno de los componentes de color.

En el caso del esquema de color RGB se obtienen politopos 5D, por lo que es difícil visualizar este tipo de información. Por otro lado, para ejemplificar la extrusión de color, el caso más simple y que es posible visualizar, es para frames con un esquema de color de escala de grises, ya que se obtiene un politopo 3D. Con base en lo anterior, cuando se realiza una extrusión de la escala de gris se obtiene que la tercera dimensión contiene la profundidad de la escala de gris, asociada a un píxel (punto) en la región 2D del frame. Por ejemplo, tomando como base el frame que se muestra en la figura 5.3a, se obtiene su extrusión 3D como se muestra en la figura 5.3b.

(a) Frame en escala de gris.



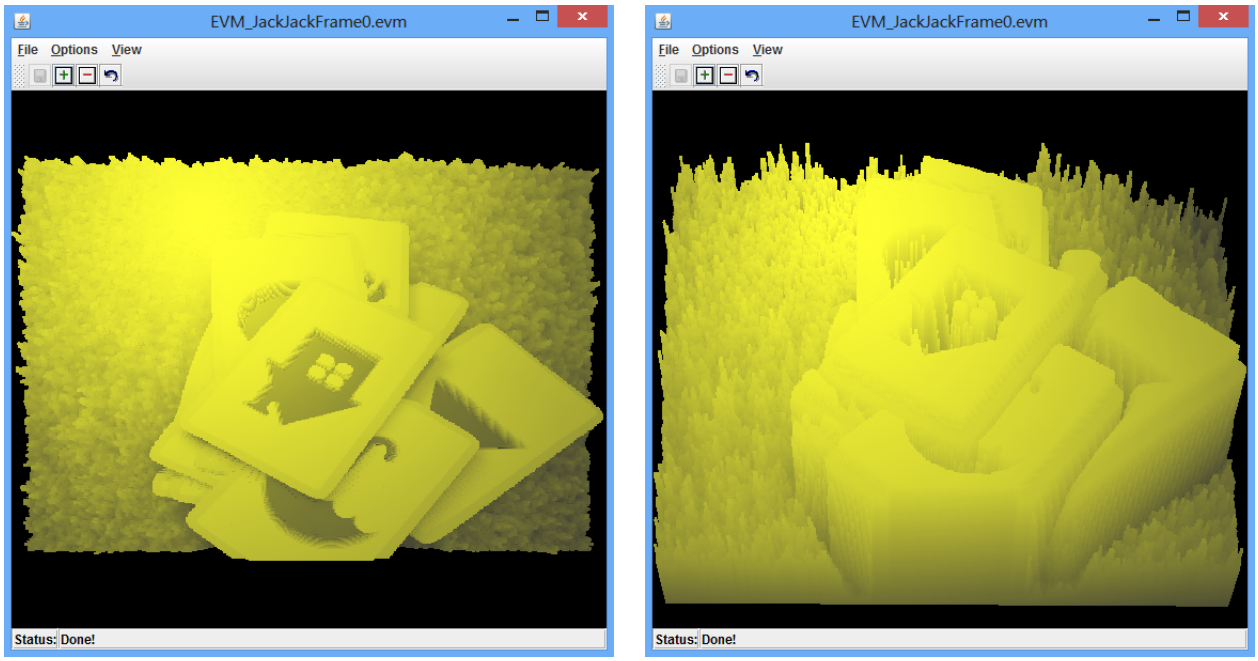
(b) Extrusión del componente de escala de gris.

Figura 5.3: Extrusión del componente de gris de un frame.

Como se observa en la figura 5.3, la extrusión en 3D indica que los píxeles con un nivel de gris alto son aquellos con color más cercano al blanco. Por otro lado, los píxeles con niveles de gris bajos, pertenecen a regiones más oscuras en el frame.

Ahora, considerando que para un frame se obtiene un conjunto de datos que existe en un espacio 3D, se obtiene su representación en el modelo 3D-EVM. Para visualizar el 3D-EVM resultante, se utiliza un software diseñado específicamente para editar y visualizar 3D-EVMs [46, 47]. En la

figura 5.4 se muestra, a través del software mencionado, la representación en el 3D-EVM del frame mostrado en la figura 5.3.



(a) 3D-EVM del frame con una vista frontal.

(b) 3D-EVM del frame con una vista en perspectiva.

Figura 5.4: Representación en el 3D-EVM del frame.

Considerando que cada pixel es representado mediante un nD-OPP, en este trabajo se proponen algoritmos para la formación de animaciones mediante una secuencia de frames. El algoritmo 5.1, presenta el procedimiento genérico para representar una secuencia de frames en el modelo nD-EVM. Este algoritmo considera como entrada un directorio de trabajo (*working directory*), en donde se encuentra la secuencia de frames. También, se considera que los nombres de archivo de los frames es *"frameX"*, donde *X* es el ordenamiento con respecto a la variable temporal.

Algoritmo 5.1: Algoritmo para generar una animación en el nD-EVM a partir de una secuencia de frames.

```
   Input : El directorio de los frames y el número del frame inicial y final
1  Procedure generateAnimation(String workingDirectory, integer initFrame,   integer endFrame)
2      EVM currentFrame, prevFrame, diffFrame;
3      string framePath;
4      prevFrame ← initEVM();
5      for i = initFrame to endFrame do
6          currentFrame ← initEVM();
7          framePath ← workingDirectory + ''frame'' + toString(i) + ''.image'';
8          currentFrame ← loadImage(framePath);
9          diffFrame ← getCouplet(prevFrame,currentFrame);
10         saveEVM(diffFrame,i);
11         prevFrame ← currentFrame;
12     saveEVM(prevFrame, endFrame +1);
```

El flujo de operación del algoritmo 5.1 es el siguiente:

- Primero se realiza la declaración de variables a utilizar , lo cual se realiza en la línea 2. En este caso se utilizan variables para el frame actual (*currentFrame*), el frame previo (*prevFrame*) y para almacenar el resultado del cálculo de diferencia entre frames consecutivos (*diffFrame*).

- Para cada frame se obtiene su representación en el nD-EVM, lo cual se logra mediante la lectura de la imagen correspondiente al frame y la función *loadImage*, véase las líneas 7 y 8.
- Posteriormente, se calcula la diferencia entre frames consecutivos mediante la operación XOR, con lo que se obtienen las proyecciones de los Couplets de la animación. Esto se realiza en la línea 9, y como se observa se utiliza el método del algoritmo 3.9, el cual calcula un Couplet que se encuentra entre dos Secciones consecutivas.
- Los Couplets son almacenados en archivos binarios, lo cual se realiza con la función *saveEVM* en la linea 10.
- Por último, en la línea 12 se almacena el último Couplet de la animación, el cual es idéntico al último frame, ya que en este contexto los frames son considerados como Secciones del nD-OPP de la animación.

Como se mencionó anteriormente, en el algoritmo 5.1 se requiere la función *loadImage*, la cual construye la extrusión de un frame y retorna su representación en el nD-EVM. Esta función realiza una doble iteración para obtener cada uno de los pixeles y su respectiva información de color. El algoritmo 5.2 muestra el procedimiento básico para obtener la extrusión de un frame.

Algoritmo 5.2: Algoritmo para obtener la representación de un frame en el modelo nD-EVM.

```
   Input : La ruta donde se encuentra el archivo del frame.
   Output: Un nD-EVM que es la extrusión del frame de entrada.
1  Procedure loadImage(string imagePath)
2      EVM p ;                                                    // El EVM resultante.
3      ImageData imageData;
4      integer colorCount;
5      integer array hyperPrismBase, hyperPrismLengths;
6      p ← InitEVM();
7      imageData ← readImageData(imagePath);
8      colorCount ← imageData.colors;
       /* Inicializar la base y longitud del hiper-prisma                                  */
9      for k = 0 to (colorCount − 1) do
10         hyperPrismBase [2 + k] ← 0;
11     hyperPrismLengths [0] ← 1;
12     hyperPrismLengths [1] ← 1;
13     for i = 0 to (imageData.height − 1) do
14         hyperPrismBase [1] ← i;
15         for j = 0 to (imageData.width − 1) do
16             hyperPrismBase [0] ← j;
               /* Obtener cada componente de color del pixel actual.                   */
17             for k = 0 to (colorCount − 1) do
18                 hyperPrismLengths [2 + k] ← imageData.pixelData(i, j).color(k) + 1;
19             populateHyperPrism(p, hyperPrismBase, 2+ colorCount, 0, hyperPrismLengths);
20     return p
```

En el algoritmo 5.2 se tiene una fase de declaración e inicialización de las variables a utilizar, véase las líneas 2-6. Posteriormente se realiza la lectura de información del frame, que en este caso es la información una imagen (líneas 7 y 8). Es importante considerar que, en el contexto de una extrusión para cada pixel en el frame se obtiene un hiper-prisma nD, para el cual se forma una base 2D relativa a las coordenadas del pixel y los componentes de color con el valor mínimo diferente de cero de acuerdo a la escala de color considerada.

Por lo anterior, en el algoritmo 5.2 se definen dos arreglos, *hyperPrismBase* y *hyperPrismLengths*, cuyo propósito es almacenar las coordenadas iniciales y longitudes del hiper-prisma respectivamente. En las líneas 9-12 se establece la configuración inicial de dichos arreglos, en donde se observa que la base de los componentes de color es el valor cero, y las longitudes del hiper-prisma en

la base 2D son las de una caja unitaria. No obstante, como se mencionó anteriormente el componente de color debe ser diferente de cero, por lo que en la línea 18 se toman los componentes de color y se agrega una unidad para la longitud del hiper-prisma. Esto con base en que si se tiene un hiper-prisma con componentes de color con valor cero no se obtiene un nD-OPP válido, entonces al agregar una unidad al componente de color se garantiza obtener un hiper-prisma apropiado.

Los ciclos *for* de las líneas 13 y 15 realizan la exploración de todos los píxeles del frame, y como se observa en las líneas 14 y 16 el arreglo *hyperPrismBase* es actualizado con base en las posiciones del píxel actual en el espacio 2D. Posteriormente, los arreglos *hyperPrismBase* y *hyperPrismLengths* se pasan como argumentos a la función *populateHyperPrism*, la cual obtiene todos los vértices que forman el hiper-prisma como tal.

Es importante considerar que para un hiper-prisma en el espacio nD se generan 2^n vértices, los cuales son vértices extremos. En este sentido, el nD-OPP de un frame consiste en la unión de todos los hiper-prismas que se forman de éste. También es importante notar que los hiper-prismas son disjuntos o cuasi-disjuntos, lo que significa que no hay superposición. Como se describe en [10, 11], cuando se tienen dos nD-OPPs p y q con estas propiedades, considerando su respectiva representación en el EVM, entonces la unión de éstos se realiza de la siguiente manera:

$$EVM_n(p \cup q) = EVM_n(p) \otimes EVM_n(q) \tag{5.1}$$

Como se observa, con las condiciones mencionadas la operación Unión es equivalente a la operación XOR, lo cual implica que los vértices comunes a ambos nD-OPPs serán eliminados. Con esto, en el algoritmo 5.3 se muestra el procedimiento para obtener un hiper-prisma a partir de los arreglos *hyperPrismBase* y *hyperPrismLengths*. Como se observa, se hace uso de la función *insertVertex* en la línea 3, la cual intenta insertar un vértice del hiper-prisma actual a un objeto nD-EVM p. Cabe mencionar que mediante dicha función, p será formado por la unión de todos los hiper-prismas que se obtengan del frame correspondiente.

En la función *insertVertex* se pueden presentar dos casos: si el vértice no existe, entonces éste es insertado en p; en caso contrario, el vértice es eliminado de p. Esta funcionalidad proporciona el mismo comportamiento que la operación XOR, ya que vértices comunes entre hiper-prismas serán eliminados.

Algoritmo 5.3: Algoritmo para obtener e insertar los vértices de un hiper-prisma.

```
   Input : El nD-EVM p, la base y longitudes del hiper-prisma, el número de dimensiones y la dimensión
           actual.
1  Procedure populateHyperPrism(nD-EVM p, integer array hyperPrismBase, integer dimensions, integer currentDim,
   integer array hyperPrismLengths)
2      if !(currentDim < dimensions) then
3          insertVertex(p,hyperPrismBase,dimensions);
4          return;
5      populateHyperPrism(p, hyperPrismBase, dimensions, currentDim +1, hyperPrismLengths);
6      hyperPrismBase [currentDim ] ← hyperPrismBase [currentDim ] + hyperPrismLengths [currentDim ];
7      populateHyperPrism(p, hyperPrismBase, dimensions, currentDim +1, hyperPrismLengths);
8      hyperPrismBase [currentDim ] ← hyperPrismBase [currentDim ] − hyperPrismLengths [currentDim ];
```

5.1.2. Cálculo de Couplets

En la sección 5.1.1, se muestra cómo se realiza la representación de frames en el modelo nD-EVM. Con ello, el siguiente paso a realizar, según el proceso de representación de una secuencia de frames ilustrado en la figura 5.1, es calcular las diferencias entre frames consecutivos.

Con lo anterior, se obtienen los Couplets que forman el nD-EVM resultante de la secuencia de frames. Para calcular las diferencias entre dos frames consecutivos representados en el nD-EVM,

se realiza la operación XOR entre éstos. Con ello, la operación XOR permite eliminar los vértices de los hiper-prismas nD que son comunes a ambos frames. Y por ende, solo se preservan los vértices que cambian de un frame a otro.

Para ejemplificar este proceso, se consideran dos frames de un video de vigilancia como se muestran en la figura 5.5. Como se observa, en estos frames el fondo de la escena es el mismo, ya que la cámara de vigilancia está estática. Consecuentemente, los cambios entre frames radican en los objetos en movimiento, en este caso se pueden tener personas o autos. Para este ejemplo se consideran dos frames en donde se encuentran dos personas caminando sobre el pavimento.

(a) Frame del primer operando para la operación XOR.

(b) Frame del segundo operando para la operación XOR.

Figura 5.5: Frames para realizar la operación XOR y obtener el Couplet correspondiente.

Ahora, la representación en el 3D-EVM de los frames de la figura 5.5 se muestran en las figuras 5.6a y 5.6b. Posteriormente, se obtienen las diferencias entre ambos 3D-EVMs y se forma el Couplet que se muestra en la figura 5.6c. Como se observa, las principales diferencias entre los frames consecutivos consisten en las dos personas que están caminando sobre el pavimento. Debido a que la parte del fondo de la escena es prácticamente la misma en ambos frames, ésta no es considerada en el Couplet.

En el Couplet de la figura 5.6c se observa que hay ciertos puntos y pequeñas regiones en donde los frames son diferentes. Estas diferencias se deben a que las cámaras tienden a hacer uso de métodos de compresión al procesar frames consecutivos, ya que éstos no cambian mucho y es posible reducir la cantidad de información necesaria para generarlos. Sin embargo, el uso de dichos métodos implica que ciertas regiones requieren menos bits de información, lo cual afecta en los detalles que se tienen en éstas. A esto se le conoce como artefactos (*Blocking Artifacts*) [8, 48].

No obstante, independientemente de la cámara que se use, siempre existe un cierto grado de difuminado (*Blur*) [8, 48]. Esto se debe principalmente al movimiento que se este llevando a cabo en la escena. Por ejemplo, en la figura 5.5 se tiene el caso de dos personas que están caminando por el pavimento, lo cual puede generar cierto difuminado debido a su movimiento. También, el movimiento de objetos puede generar cierto difuminado debido al desenfoque de la cámara.

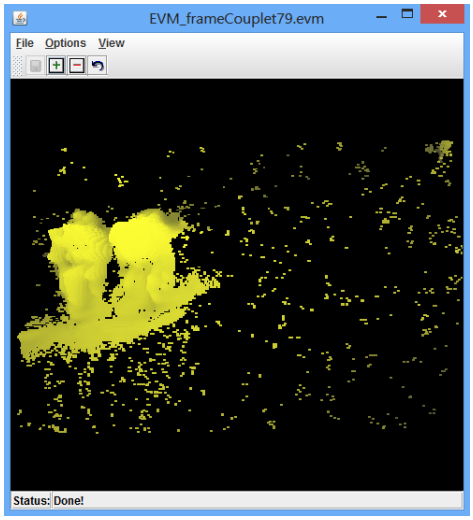
5.1.3. Archivos Binarios de los Couplets

El formato en que se almacenan los archivos de los Couplets es de tipo binario. La estructura del archivo consiste de dos partes: la cabecera y el contenido. La cabecera del archivo contiene la dimensionalidad del nD-EVM que se almacenó. Con ello se puede almacenar un nD-EVM para cualquier número de dimensiones. Un ejemplo de cabecera es el siguiente:

```
3D-EVM
```

(a) Primer frame representado en el 3D-EVM.

(b) Segundo frame representado en el 3D-EVM.



(c) Couplet obtenido de la operación XOR.

Figura 5.6: Visualización de los frames y el Couplet en el visualizador de 3D-EVMs.

Ahora, el contenido del archivo se basa en el contenido del nD-EVM. Como se ha mencionado, el EVM define que para un politopo dado solo se conservan los vértices extremos. Entonces, prácticamente un nD-EVM está formado por una lista de vértices en forma vectorial. Por ejemplo, considerando el caso para un hiper-cubo 3D unitario con su vértice inicial en el origen, se obtienen ocho vértices extremos:

$$[0,0,0],\ [0,0,1],\ [0,1,0],\ [0,1,1],\ [1,0,0],\ [1,0,1],\ [1,1,0],\ [1,1,1]$$

Debido a que se conoce la dimensionalidad del nD-EVM, también se conoce la cantidad de elementos que contiene cada vector, ya que dicha cantidad está denotada por n. Por lo tanto, se sabe cuántos datos hay que leer para formar un vértice, con ello la lectura y almacenamiento se realiza por bloques de información del tamaño especificado en la cabecera, y además la información está almacenada de manera secuencial.

También es importante considerar el tipo de dato utilizado en la implementación del nD-EVM, ya que de ello depende la cantidad de bytes para cada elemento de un vértice. En el caso de una implementación en *C++*, considerando un tipo de dato *unsigned int* se tienen 16 bits[1] por cada elemento en el arreglo, lo cual equivale a dos bytes de información. Para el ejemplo anterior del hiper-cubo 3D unitario, el archivo de su correspondiente 3D-EVM contiene cada una de las coordenadas de los vértices y se almacena como se muestra a continuación. Cabe aclarar que para efectos de visualizar lo que se almacena se muestra su valor entero, no obstante en el archivo se almacenan los respectivos valores binarios:

```
3D-EVM00000101001110010111 0111
```

5.2 Segmentación de Video usando Compacidad Discreta y Mapas Auto-organizados de Kohonen

Como se explicó en el capítulo 4, el proceso de segmentación requiere de un conjunto de datos de entrenamiento. Éste conjunto contiene las muestras que van a ser agrupadas y sus respectivas características. Sin embargo, hasta este punto solo se han presentado los fundamentos para la representación de una secuencia de frames en el nD-EVM. Así como el cálculo de Compacidad Discreta para un nD-OPP representado en el nD-EVM.

Ahora, el enfoque que se plantea en este trabajo, consiste en el uso de máscaras. En el área de procesamiento de imágenes, las máscaras se usan para la extracción y/o alteración de información de regiones particulares de una imagen. No obstante, en el contexto de la secuencia de frames representada en el nD-EVM (animación), las máscaras se utilizan para extraer información de regiones dentro de la animación. Por lo que una región de la animación es una *sub-animación*, ya que contiene una porción de ésta. Es decir, se tiene el cambio a través de una porción de tiempo, de una pequeña región 2D de los frames. En el apartado 5.2.1, se describe cómo se obtienen dichas sub-animaciones mediante máscaras.

Entonces, la segmentación consiste en que el SOM agrupe sub-animaciones similares dentro de una animación. Por lo que, el conjunto de datos de entrenamiento se forma con los valores de DC de cada una de las sub-animaciones dentro de la animación original, lo cual se describe en el apartado 5.2.2. En este sentido, lo que el SOM realiza es el agrupamiento de regiones en los frames que presentan un cambio similar a lo largo del tiempo.

5.2.1. Generacion de Máscaras

En este apartado se muestra la generación de máscaras nD para realizar la extracción de regiones dentro de una animación representada en el nD-EVM. En este contexto, una máscara consiste en un nD-OPP que contiene una región en el espacio $(tiempo, x, y, colores)$. Es decir, es una animación con longitudes configurables, con lo que se puede obtener una sub-animación en una región 2D (x,y) específica.

Para la profundidad de color se consideran tantos componentes de color como sean necesarios. Por ejemplo, para imágenes en escala de grises se utiliza solo un componente de color que pueden ser representados con un byte de información. Por otro lado, para imágenes con un esquema de color RGB, se tienen tres componentes de color que pueden ser representados con un byte de información cada uno. En este sentido, al momento de generar una máscara, se considera el valor máximo en cada componente de color según la cantidad de bytes necesarios para su representación. Por lo que se utiliza la siguiente ecuación para obtener el valor máximo de cada componente:

$$maxValue = 2^{Bytes*8}$$

[1] Véase los tipos de datos en http://en.cppreference.com/w/cpp/language/types

Las máscaras se utilizan para extraer sub-animaciones de la animación original, lo cual se logra mediante la operación intersección. Con ello se pueden extraer características de éstas sub-animaciones y así distinguirlas entre otras. Dichas características servirán para realizar el agrupamiento de las diferentes sub-animaciones que se extraen mediante una convolución de la máscara y la animación expresada en el nD-EVM, es decir, se realiza un barrido a lo largo de toda la animación sobre la variable temporal y el espacio 2D de los frames.

Para generar una máscara y representarla en el nD-EVM, se usa el algoritmo 5.3, ya que solo se necesita proporcionar su coordenada inicial y las longitudes de ella. Para ilustrar el proceso de intersección entre una máscara y la animación en el EVM, asumamos que una animación es representada en un 3D-EVM y los frames y Couplets se representan a través de 2D-EVMs. En la figura 5.7, se ilustra un ejemplo de una animación y una máscara ubicada en algún lugar dentro de la animación.

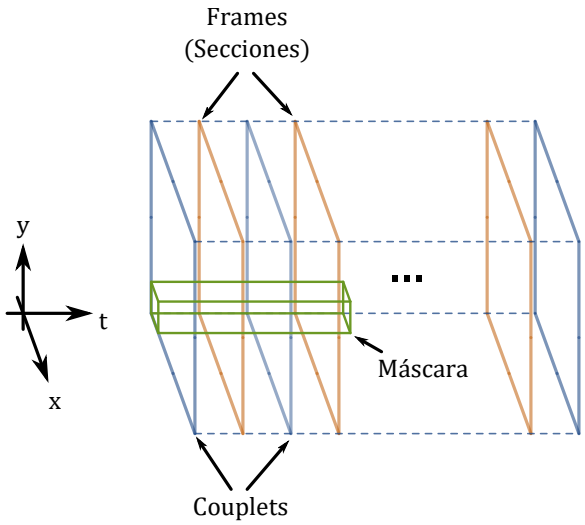


Figura 5.7: Ejemplo de una animación y una máscara para la extracción de sub-animaciones.

Una sub-animación, en este caso particular, puede ser vista como el 3D-EVM que contiene los cambios a través del tiempo de una pequeña región 2D de los frames. Para obtener todas las posibles sub-animaciones se realizan desplazamientos unitarios en la máscara para los ejes ordenados base x, y y t. Esto es realizado para todas las coordenadas posibles dentro de la animación, la única restricción es que la máscara no debe rebasar la frontera de la animación para cada uno de los ejes coordenados.

Para el caso de la animación representada en el nD-EVM mostrada en la figura 5.6, se realizó la extracción de una sub-animación utilizando una máscara con las siguientes dimensiones: una región 2D de 100x100, una longitud temporal de 5 frames y un componente de color con un byte para su representación. Como resultado de aplicar la operación intersección entre la animación original y la máscara, se obtuvo una sub-animación cuyos Couplets se muestran en la figura 5.8. Como se observa, se obtienen 6 Couplets en la sub-animación resultante, lo cual es razonable en el sentido que se estableció una longitud temporal de 5 frames, y se requieren 6 Couplets para englobar a 5 Secciones (frames). También se observa que los Couplets contienen las diferencias entre frames, pero si se obtiene la secuencia de Secciones de esta sub-animación, se obtendrán las regiones correspondientes de las Secciones originales.

5.2.2. Cálculo de Compacidad Discreta para Sub-animaciones

En el caso de una sub-animación, como se mencionó en la Sección 3.7.5, para obtener la compacidad discreta es necesario especificar las cotas L_{Min} y L_{Max} de la ecuación 3.6. Lo cual se efectúa considerando los siguientes dos casos:

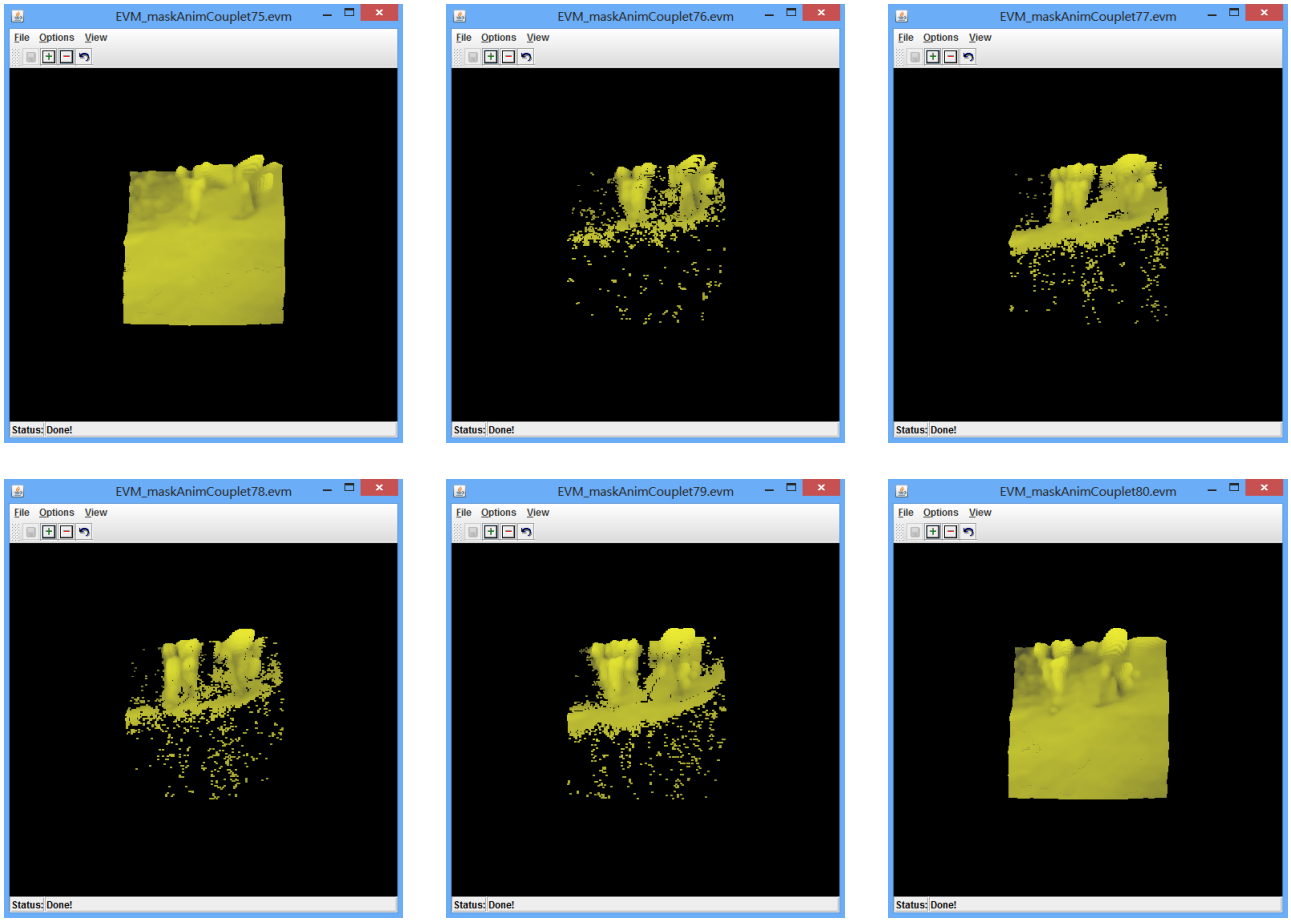


Figura 5.8: Couplets del resultado de la operación intersección entre la máscara y la animación.

- Para L_{Max} se consideran los contactos internos de la máscara original. Es decir, la máscara que cuenta con los componentes de color en sus valores máximos. Lo cual, por ejemplo en escala de grises puede ser visto como una sub-animación con frames con color blanco.
- Para L_{Min} se consideran todos los contactos internos de una máscara similar a la original, pero la diferencia radica en los valores máximos de los componentes de color. En este caso se consideran que los valores de los componentes de color son los mínimos. Lo cual, para el caso de un componente de color en escala de gris, se puede ver como una sub-animación con frames en negro.

Con lo anterior, ahora es posible obtener la compacidad discreta usando la ecuación 3.6. No obstante esta ecuación requiere el cálculo de contactos internos del nD-EVM de la sub-animación (L_c), lo cual se obtiene mediante el algoritmo 3.14.

5.2.3. Generación del Conjunto de Entrenamiento a partir de la Compacidad Discreta de Sub-animaciones

Para generar el conjunto de datos de entrenamiento para el SOM, para cada sub-animación extraída mediante una máscara se extrae su DC. El proceso de extracción de sub-animaciones consiste básicamente en una convolución, el cual se muestra en el algoritmo 5.4 y se describe a continuación:

- Primero se definen e inicializan las variables a utilizar, véase las líneas 2-9. Como se observa, este algoritmo requiere de una máscara como parámetro de entrada, la cual es una máscara con su coordenada base en el origen del espacio (x, y, t).
- La función *maskAnimSections* que se observa en las líneas 10 y 30, se utiliza para obtener una animación que contiene solo los frames que actualmente abarca la máscara. Debido a que los Couplets de la animación original se encuentran almacenados en disco duro, y para optimizar el uso de memoria RAM, solo se utilizan los Couplets necesarios para la intersección con la máscara.

  Lo anterior en el sentido que el algoritmo de Operaciones Booleanas (algoritmo 3.11) realiza el proceso de cálculo de Secciones de manera exhaustiva, lo cual resulta ineficiente ya que con cada desplazamiento de la máscara se deben obtener nuevamente todas las Secciones que le preceden. Por ejemplo, en la figura 5.7 se muestra una máscara que abarca dos frames (Secciones) de la animación original, entonces para realizar la operación intersección solo se requieren las dos primeras Secciones. Así cuando se aplique un desplazamiento a la máscara no será necesario calcular nuevamente dichas Secciones, como se indica en el algoritmo 3.11.
- Las estructuras de repetición *while* de las líneas 15, 16 y 17 realizan el barrido a lo largo de las longitudes de la animación en los ejes coordenados x, y y t.
- Posteriormente, se realiza la operación intersección entre la máscara actual y el objeto que contiene solo los frames que abarca la máscara (línea 18), con lo que se obtiene la sub-animación correspondiente.
- En la línea 19 se obtiene el valor de DC (*maskDC*) para la sub-animación actual *currentResult*. Los valores de DC son almacenados en el conjunto $DCValues$, en el cual se almacenan todos los valores de DC de las sub-animaciones (línea 20).
- En las líneas 21, 25 y 29 se efectúan los desplazamientos unitarios de la máscara para cada uno de los ejes coordenados antes mencionados. Como se observa, en la línea 30 se obtienen nuevamente los frames requeridos para la intersección de la animación con la máscara. Ello se debe a que en la línea 29 se efectúa un desplazamiento en el tiempo y por ende se deben obtener Secciones acordes con la máscara.
- Por último se retorna el conjunto de datos $DCValues$, el cual será utilizado posteriormente para la etapa de entrenamiento del SOM.

Como se observa en la línea 18 del algoritmo 5.4, para obtener la sub-animación correspondiente se emplea la función *maskIntersection*, la cual se utiliza específicamente para aplicar la operación intersección entre una máscara y una secuencia de Secciones (frames). En el algoritmo 5.5 se muestra el algoritmo de la función *maskIntersection* y ésta se describe a continuación:

- Como se observa, la etapa de declaración e inicialización de variables se efectúa en las líneas 2-8.
- En las líneas 9-11 se obtiene la Sección correspondiente a la máscara, que como se puede deducir de manera sencilla, la máscara tiene solo una Sección. Esto con base en que la máscara es una hiper-caja con longitudes específicas en los ejes coordenados (x, y, t) y en los ejes coordenados para los componentes de color se tiene como longitud el máximo correspondiente.
- La estructura de repetición de la línea 12 permite explorar las Secciones que contiene el párámetro *sectionSeq*, las cuales son los frames de la animación original que son necesarios para la operación intersección con la máscara.
- La operación intersección de las Secciones de *sectionSeq* y la Sección de la máscara se efectúa en la línea 15. Como se observa, se utiliza la función del algoritmo de Operaciones Booleanas convencional, ello debido a que a partir de este punto la operación intersección se aplican de manera usual.

Algoritmo 5.4: Algoritmo para el proceso de convolución de la máscara y generar el conjunto $DCValues$.

```
Input : El nD-EVM de la máscara, el frame final y el ancho y alto de los frames.
Output: El conjunto DCValues que contiene los valores del descriptor DC para cada posible
        sub-animación.
1  Procedure AnimConv(nD-EVM mask, integer endFrame, integer width, integer length)
2      EVM currentResult ;                                   // Resultado actual de la intersección
3      EVM frameSeq;
4      integer τ ← 0,maxTimeShift;
5      integer x ← 0,maxXShift;
6      integer y ← 0,maxYShift;
7      real maskDC;
8      Set of reals DCValues;
9      integer i ← 0 ;                                       // Contador de cada desplazamiento
10     frameSeq ← maskAnimSections(mask,endFrame);
11     maxTimeShift ← endFrame − mask.timeLength + 1;
12     maxXShift ← width − mask.xLength + 1;
13     maxYShift ← length − mask.yLength + 1;
14     DCValues ← ∅;
15     while τ ≤ maxTimeShift do
16         while y ≤ maxYShift do
17             while x ≤ maxXShift do
18                 currentResult ← maskIntersection(mask,frameSeq);
19                 maskDC_{τ,i} ← discreteCompactness(currentResult, mask.LcMin, mask.LcMax);
20                 DCValues.addDCValue(maskDC_{τ,i});
21                 mask.EVMTraslation(1, 1);
22                 i ← i + 1;
23                 x ← x + 1;
24             mask.dimReset(1);
25             mask.EVMTraslation(2, 1);
26             y ← y + 1;
27             x ← 0;
28         mask.dimReset(2);
29         mask.EVMTraslation(0, 1);
30         frameSeq ← maskAnimSections(mask,endFrame);
31         τ ← τ + 1;
32         y ← 0;
33         i ← 0;
34     return DCValues
```

- En la línea 16 se obtiene el correspondiente Couplet, y en las líneas 18-19 éste es agregado al resultado final. En la condición de la línea 21 se agrega el último Couplet de la sub-animación correspondiente.

Con base en lo anterior es importante establecer cómo está constituido el conjunto de datos *DCValues*, ya que está formado por los valores de compacidad discreta $maskDC$ de todas las posibles sub-animaciones. El conjunto $DCValues$ se organiza de la siguiente manera:

$$\begin{aligned} DCValues = \{ & maskDC_{0,1}, maskDC_{0,2}, \ldots, maskDC_{0,i}, \\ & maskDC_{1,1}, maskDC_{1,2}, \ldots, maskDC_{1,i}, \\ & \vdots \\ & maskDC_{\tau,1}, maskDC_{\tau,2}, \ldots, maskDC_{\tau,i} \} \end{aligned}$$

En donde:

- τ denota el desplazamiento en el tiempo de la máscara y la sub-animación correspondiente. Cabe mencionar que el valor máximo que puede tomar τ están en función de la cantidad de

Algoritmo 5.5: Algoritmo para la intersección de la máscara y una secuencia de Secciones.

```
   Input : La máscara y el nD-EVM que contiene una secuencia de Secciones.
   Output: EL resultado de la operación Intersección.
1  Procedure maskIntersection(EVM mask, EVM sectionSeq)
2      EVM animSection, maskSection, couplet;
3      EVM result, rPrevSection, rCurrentSection;
4      integer coord,dim;
5      dim ← dimDepth ;                                   // Obtener la dimensionalidad de la máscara
6      maskSection ← initEVM();
7      rCurrentSection ← initEVM();
8      result ← initEVM();
9      coord ← mask.getCoord();
10     couplet ← mask.readCouplet();
11     maskSection ← getSection(maskSection,couplet);
12     while !sectionSeq.endEVM() do
13         animSection ← sectionSeq.readCouplet();
14         rPrevSection ← rCurrentSection;
15         rCurrentSection ← booleanOperation(animSection,maskSection,"intersection",dim-1);
16         couplet ← getCouplet(rPrevSection,rCurrentSection);
17         if !couplet.isEmpty() then
18             couplet.setCoord(coord);
19             result.putCouplet();
20         coord ++;
21     if !rCurrentSection.isEmpty() then
22         rCurrentSection.setCoord(coord);
23         result.putCouplet(rCurrentSection);
24     return result;
```

frames y la longitud en el tiempo de la máscara. Ya que en el desplazamiento en el tiempo, la máscara no debe rebasar el límite de frames.

$$maxTimeShift = (endFrame - mask.timeLength + 1)\tau \qquad : 0, 1, \ldots, maxTimeShift$$

- i es el índice de cada sub-animación en el barrido en (x, y). Por lo que, al igual que en el caso anterior, es importante considerar los desplazamientos máximos en x y y. Ya que la máscara no debe rebasar los límites en el tamaño de los frames. Con esto se obtiene los desplazamientos máximos en x y y, y el rango de valores de i de la siguiente manera:

$$\begin{aligned} maxXShift &= frame.width - mask.xLength + 1 \\ maxYShift &= frame.length - mask.yLength + 1 \\ i &: 0, 1, \ldots, (maxXShift * maxYShift) \end{aligned}$$

Como se observa, este proceso es similar a la convolución que se realiza en el área de procesamiento de imágenes, en donde se hace el barrido de una máscara a lo largo de toda la imagen para realizar cambios en ella. Pero en este caso, la máscara no realiza cambio alguno, sino que extrae regiones de la animación para después obtener su correspondiente descriptor de DC.

5.2.4. Agrupamiento de Sub-animaciones con un Mapa Auto-Organizado de Kohonen

En él apartado 5.2.3 se describe el proceso de formación del conjunto de entrenamiento para el SOM. El proceso de entrenamiento de un SOM es el mismo que se presentó en el capítulo 4, y los vectores de pesos de las neuronas son $1D$. Esto debido a que solo se tiene un descriptor para cada sub-animación, que es el valor de DC.

En base al algoritmo de entrenamiento de un SOM que se describe en la sección 4.2.4, cuando éste finaliza se obtienen los vectores de pesos sinápticos finales. No obstante, hasta este punto solo se obtuvo el SOM entrenado. Ahora se deben formar los grupos de datos con base en la neurona ganadora.

Lo anterior se realiza mediante una presentación más del conjunto de entrenamiento. Pero en esta ocasión solo se obtienen los índices de la neurona ganadora para cada dato de entrada. Como se ha mencionado en el Capítulo 4, la neurona ganadora es aquella que tiene la menor distancia entre el peso sináptico y el valor de DC del dato de entrada.

Ahora, suponiendo que se tienen m neuronas, a partir de los datos de entrada se obtendrán m grupos, $cluster_1, cluster_2, \ldots, cluster_m$. En donde un $cluster_j$ es un conjunto que tiene los índices de los valores de DC ($maskDC_{\tau,i}$) de los cuales la neurona ganadora es la neurona j:

$$Sea\ cluster_j,\ en\ donde\ j = 1, 2, \ldots, m$$
$$cluster_j = \left\{(\tau, i) \mid maskDC_{\tau,i} \in DCValues,\ minDistanceNeuron(maskDC_{\tau,i}) = j\right\}$$

Como se observa, los grupos solo contienen los índices del valor de DC. Esto se debe a que lo que nos interesa es la sub-animación correspondiente a este valor de DC, no el valor de DC como tal. Ya que por cada grupo se debe formar el nD-EVM resultante, el cual está formado por la unión de todas las sub-animaciones del conjunto asociado a la correspondiente neurona.

Para obtener la sub-animación correspondiente a un índice de $maskDC$ dentro de un $cluster$, primero es importante generar la máscara con la que fue generada. Con ella es posible extraer la sub-animación mediante la operación intersección. Posteriormente se realiza la operación unión entre todas las sub-animaciones extraídas de la animación original, y con ello se obtiene el nD-EVM del $cluster$.

Para realizar lo anterior, primero se inicializa la máscara con las posiciones en (t, x, y) en el origen, es decir sin desplazamiento en el tiempo y en la posición $(0, 0)$ en (x, y). Posteriormente, se deben obtener los desplazamientos requeridos para extraer la sub-animación indicada en los índices (τ, i) de la siguiente manera:

- Se deben considerar los máximos desplazamientos posibles en las dimensiones (x, y) descritas en el algoritmo 5.4, *maxXShift* y *maxYShift*.
- Posteriormente, el desplazamiento en la dimensión y se obtiene de la siguiente manera: $yShift = i/maxXShift$.
- Y el desplazamiento en la dimensión x se obtiene de la siguiente manera:

$$xShift = i - yShift * maxXShift$$

Ahora, debido a que el orden de las dimensiones en un frame representado en el nD-EVM es el siguiente $(t, x, y, colores)$, en donde las dimensiones de color dependen del esquema de color de los frames, ya se puede formar la máscara para extraer la sub-animación correspondiente a (τ, i):

```
mask.Traslation(0,τ);
mask.Traslation(1,xShift);
mask.Traslation(2,yShift);
```

5.2.5. Estimación del Tiempo de Ejecución

Es importante realizar el análisis de tiempo de ejecución para el proceso de segmentación ya que de esta manera se proporciona un panorama general sobre la complejidad temporal. También es importante notar que los algoritmos propuestos en este capítulo emplean algoritmos básicos del nD-EVM que se describen en la sección 3.7, para los cuales en [11] se presenta una estimación de su complejidad temporal. Por ello, en este caso se consideran las estimaciones de las complejidades temporales de los algoritmos de Operaciones Booleanas para la intersección, cálculo de contenido y *mergeXOR*:

- La estimación de la complejidad temporal para la operación intersección es la siguiente:

$$t_{int}(x,n) = x^{1.1737} n^{1.0862} \tag{5.2}$$

  En donde x es la cardinalidad total de los dos nD-EVMs que son operados y n es la dimensionalidad.

- Para el algoritmo del cálculo de contenido se tiene que la estimación de la complejidad temporal es la siguiente:

$$t_{cont}(x,n) = x^{1.1894} n^{0.8390} \tag{5.3}$$

  En donde nuevamente x es la cardinalidad total de los dos nD-EVMs que son operados y n es la dimensionalidad.

- Para el algoritmo *mergeXOR* se tiene que la complejidad temporal es la siguiente:

$$t_{mergeXOR} = x \tag{5.4}$$

  Como se observa el tiempo de ejecución es lineal y es equivalente a la cardinalidad del nD-EVM.

Ahora, para el proceso de convolución y formación del conjunto $DCValues$ se obtiene su estimación del tiempo de ejecución de la siguiente manera:

- Primero se debe considerar la cantidad de las sub-animaciones que se pueden obtener de la animación representada en el 6D-EVM, esto es la cardinalidad del conjunto $DCValues$:

$$C_{DCValues} = maxXShift \cdot maxYShift \cdot maxTimeShift \tag{5.5}$$

  - Para obtener cada sub-animación se realiza la operación intersección entre la máscara y la animación original, para lo cual se utiliza la ecuación 5.2 y se obtiene el tiempo $t_{maskInt}(x,n)$, en donde la cardinalidad x consiste de la cardinalidad de la máscara y la máxima cardinalidad de los frames:

$$x = 2^5 + x_f \cdot y_f \cdot t_{mask} \cdot 2^5 \tag{5.6}$$

    La dimensionalidad en este caso es $n = 6$, x_f y y_f corresponden a las longitudes de los frames y t_{mask} es la longitud de la máscara en el eje del tiempo, 2^5 es la cantidad de vértices extremos que se obtienen para una hiper-caja en el espacio 5D. Se consideran los vértices de una hiper-caja en 5D debido a que un frame en RGB se representa en un 5D-EVM. A partir de este punto se consideran los peores escenarios en la representación de los frames, lo cual se presenta cuando las hipercajas de las representaciones no tienen vértices extremos en común.

  - Para cada sub-animación obtenida se calcula su descriptor de DC, en donde se consideran cuatro etapas principales con base a los algoritmos 3.13 y 3.14: reordenamientos de los ejes coordenados, cálculo de secciones, intersección de Secciones consecutivas y el cálculo de contenido.

Considerando el peor escenario, para el proceso de obtención de Secciones se debe procesar una cantidad de vértices extremos igual a:

$$x_{mask} \cdot y_{mask} \cdot t_{mask} \cdot C_{red} \cdot C_{green} \cdot C_{blue}$$

en donde x_{mask}, y_{mask} y t_{mask} son las longitudes de la máscara y C_x corresponde a las longitudes en cada componente de color, que para el caso del esquema RGB24 se tiene que $C_{red} \cdot C_{green} \cdot C_{blue} = 256^3$.

En cada obtención de Secciones se utiliza el algoritmo *mergeXOR*, lo cual tiene un tiempo de ejecución $t_{mergeXOR}(x)$, en donde $x = x_{mask} \cdot y_{mask} \cdot 2^5 \cdot 2$, con base en la ecuación 5.4.

El tiempo para obtener las intersecciones entre Secciones consecutivas se calcula mediante la ecuación 5.2 con lo que se obtiene $t_{secInt}(x,n)$, en donde $x = x_{mask} \cdot y_{mask} \cdot 2^5 \cdot 2$ y la dimensionalidad es $n = 5$.

Por último, el cálculo del contenido del resultado de la intersección se realiza considerando la ecuación 5.3 y se obtiene el tiempo $t_{cont}(x,n)$ en donde $x = x_{mask} \cdot y_{mask} \cdot 2^5$.

Con base en lo anterior, el tiempo para el cálculo de DC queda expresado de la siguiente manera:

$$t_{DC} = x_{mask} \cdot y_{mask} \cdot t_{mask} \cdot 256^3 \cdot (t_{mergeXOR} + t_{secInt} + t_{cont}) \tag{5.7}$$

Como se observa, el tiempo de ejecución para el proceso de convolución se calcula de la siguiente manera:

$$t_{conv} = C_{DCValues} \cdot (t_{maskInt} + t_{DC}) \tag{5.8}$$

Para el proceso de entrenamiento del SOM se calcula su tiempo de ejecución con base al número de presentaciones n_p del conjunto de entrenamiento $DCValues$, la búsqueda de la neurona ganadora y la actualización de la vecindad. Con lo anterior la estimación del tiempo de ejecución es la siguiente:

$$t_{training} = n_p \cdot C_{DCValues} \cdot (m + m) \tag{5.9}$$

en donde m es el número de neuronas en el SOM.

5.2.6. Cómputo Concurrente

En este apartado se establecen las condiciones necesarias para realizar el cálculo de DC para las sub-animaciones mediante cómputo concurrente. Esto se realiza con base en que es deseable aprovechar el máximo de la capacidad de procesamiento de una computadora, ya que en la actualidad existen procesadores con múltiples núcleos (*Multicore*) [49]. Debido a que la naturaleza del proceso de convolución para el eje del tiempo es independiente para diferentes desplazamientos de la máscara, éste puede ser dividido en tareas más pequeñas partiendo de la idea que una función externa realiza los desplazamientos de la máscara hasta alcanzar el final de la animación.

Ahora, considerando que el algoritmo 5.4 cuenta con una estructura *while* para realizar desplazamientos unitarios en el eje del tiempo (línea 15), esta estructura de repetición puede ser omitida y la convolución sobre los ejes x y y se realiza sin cambios. Con esto, es posible que el proceso externo realice el desplazamiento de la máscara, para lo cual se utiliza el método *mask.Traslation(0,τ)*. En el algoritmo 5.6 se establecen los cambios mencionados.

Algoritmo 5.6: Algoritmo para el proceso de convolución de la máscara y generar el conjunto $DCValues$.

```
Input : El nD-EVM de la máscara, el frame final y el ancho y alto de los frames.
Output: El conjunto DCValues que contiene los valores del descriptor DC para cada posible
        sub-animación.
Procedure AnimConv2(nD-EVM mask, integer endFrame, integer width, integer length)
    EVM currentResult ;                                   // Resultado actual de la intersección
    EVM frameSeq;
    integer x ← 0,maxXShift;
    integer y ← 0,maxYShift;
    real maskDC;
    Set of reals DCValues;
    integer i ← 0 ;                                       // Contador de cada desplazamiento
    frameSeq ← maskAnimSections(mask,endFrame);
    maxXShift ← width − mask.xLength + 1;
    maxYShift ← length − mask.yLength + 1;
    DCValues ← ∅;
    while y ≤ maxYShift do
        while x ≤ maxXShift do
            currentResult ← maskIntersection(mask,frameSeq);
            maskDC_i ← discreteCompactness(currentResult, mask.LcMin, mask.LcMax);
            DCValues.addDCValue(maskDC_{τ,i});
            mask.EVMTraslation(1, 1);
            i ← i + 1;
            x ← x + 1;
        mask.dimReset(1);
        mask.EVMTraslation(2, 1);
        y ← y + 1;
        x ← 0;
    return DCValues
```

Con base en lo anterior, se propone usar el enfoque de programación de múltiples hilos (*Multithreading*) [49], en donde se tendrá un proceso principal que se encarga de gestionar los desplazamientos de la máscara y ejecutar el proceso de convolución, en este caso dicho proceso se ha denominado *AnimConvLauncher*. El flujo de operación de este proceso se ilustra en la figura 5.9 y se describe a continuación:

- Como primer paso se debe obtener la configuración del video a procesar, la máscara y la cantidad de threads a utilizar. Para este caso se propone utilizar un archivo de texto en donde se almacena la configuración, a continuación se muestra un ejemplo de cómo está formado dicho archivo:

```
#Frames initFrame: 0 endFrame: 54 colors: 1 width: 240 length: 160
#Mask xLength: 3 yLength: 3 timeLength: 3
#Clustering clusters: 15 iterations: 200
#ThreadCount conv: 4 clusters: 4
```

- Posteriormente, se debe inicializar el sistema de ejecución de threads con base en el lenguaje de programación, en este caso la implementación del proceso *AnimConvLauncher* se realizó en el lenguaje de programación JAVA. Cabe mencionar que se generaron archivos ejecutables de las implementaciones en el lenguaje C++ tal como se describe en la sección 6.3, los cuales contienen los métodos del EVM, procesamiento de video y SOMs.
- El proceso *AnimConvLauncher* gestiona la ejecución del ejecutable de la implementación del proceso de convolución correspondiente al algoritmo 5.6, ello considerando la cantidad de threads que se halla elegido (figura 5.9).
- Por último, una vez alcanzado el máximo desplazamiento de la máscara, el proceso *AnimConvLauncher* finaliza su ejecución.

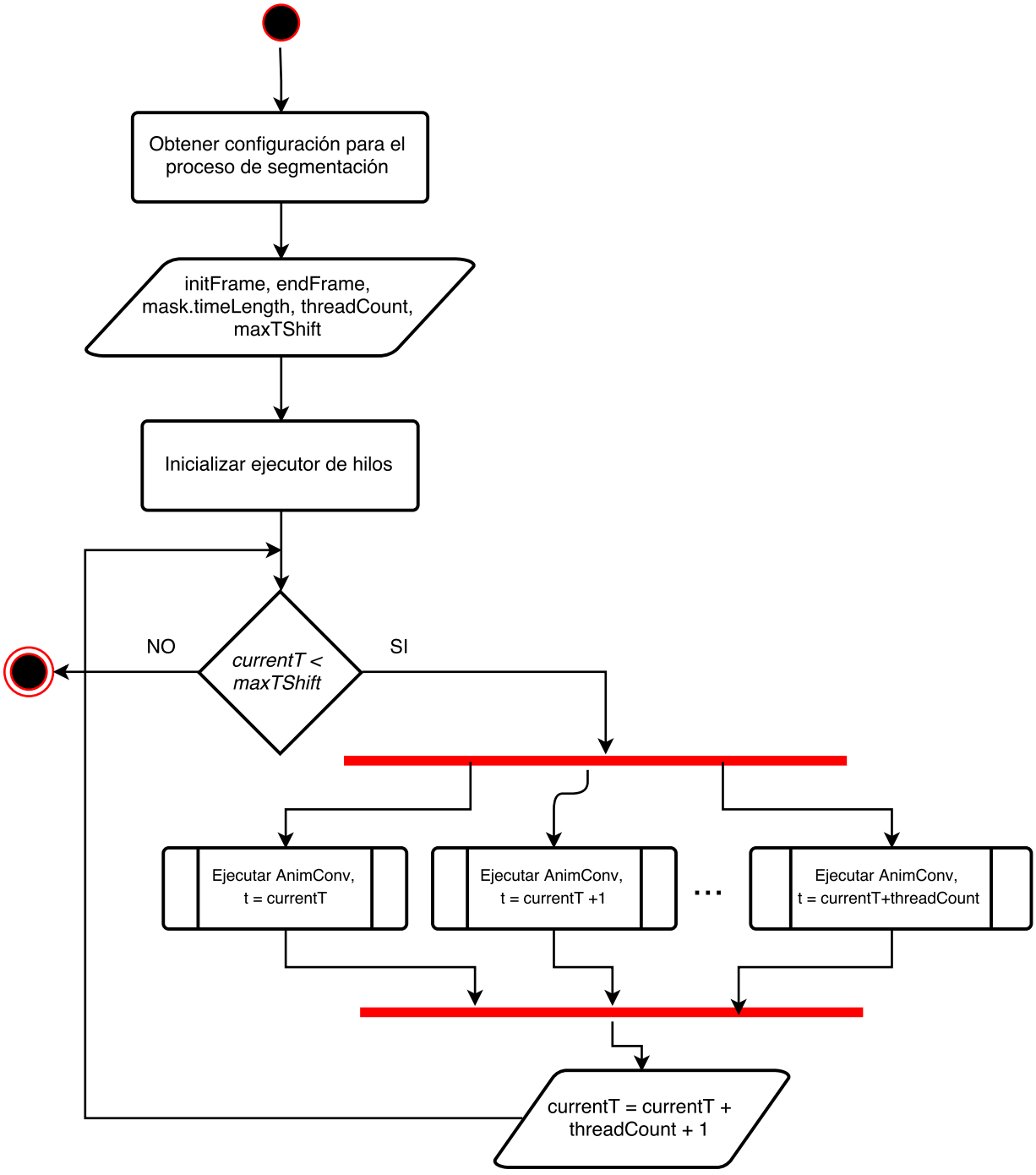


Figura 5.9: Diagrama de flujo del proceso de convolución usando el enfoque multithreading.

Capítulo 6

Implementación en C++

En este capítulo se presenta la implementación obtenida en el lenguaje C++ para el modelo nD-EVM, sus métodos básicos y los métodos para realizar el procesamiento de video. Es importante mencionar que la implementación del modelo nD-EVM considera que se cuenta con una estructura de datos para gestionar el almacenamiento de vértices extremos. En este caso para dicha estructura se utiliza un árbol Trie [11], del cual se muestra su implementación en el apéndice A.

6.1 Implementación de Métodos del nD-EVM

La implementación del modelo nD-EVM consiste en una clase que incluye un árbol Trie de la clase *TrieTree* (Apéndice A). En el código 6.1 se muestra la declaración de la clase nD-EVM. En donde se tiene un apuntador a un objeto de la clase TrieTree. En este sentido, un nDEVM tiene asociado un TrieTree (línea 7), el cual a su vez cuenta con un nodo raíz del árbol Trie. Las variables que se muestran en las líneas 9-11 se utilizan para el proceso de convolución descrito en el apartado 5.2.3 y el objeto de la clase SOM se utiliza para aplicar el agrupamiento de sub-animaciones.

código 6.1: Definición de la clase nDEVM

```
1  #include "TrieTree.h"
2
3  template<typename valueType>
4  class nDEVM {
5  private:
6  public:
7      TrieTree<valueType> *trieTree;
8      // -- Variables para uso exclusivo del proceso de convolucion y compacidad discreta.
9      valueType maskMax[3];
10     valueType maskMin[3];
11     valueType LcMin = 0, LcMax = 0;
12     // -- Objeto de la clase SOM.
13     SOM *som;
14
15     // -- Constructores
16     // -- Funciones
17 }
```

En las líneas 9-11 del código 6.1 se definen variables que se usan específicamente para el cálculo de DC de las sub-animaciones y para realizar la convolución de la máscara y la animación. Con esto se mejora el rendimiento de dichos procesos debido a que no será necesario realizar su cálculo de manera dinámica.

En esta sección, se presenta la implementación de los procedimientos básicos del nD-EVM, y en los apartados subsecuentes se presentan métodos de uso más específico. En el código 6.2 se muestra el método para verificar si el nD-EVM está vacío. Como se observa en este código, también se

implementa el equivalente a esta consulta en la clase TrieTree. Ya que para saber si el nD-EVM esta vacío, se debe consultar al nodo raíz del árbol Trie.

código 6.2: Método para verificar si el modelo nDEVM está vacío

```
template<typename valueType>
bool nDEVM<valueType>::isEmpty(){
    return trieTree->isEmpty();
}
```

El método del código 6.3 se utiliza para insertar un Couplet en un nD-EVM. Este método toma un (n-1)D-EVM, que se encuentra inmerso en un espacio nD y lo inserta en un nD-EVM. Esto quiere decir que el Couplet (n-1)D se le asocia una coordenada para la dimensión x_1, con lo que se vuelve un (n-1)D-EVM inmerso en un espacio nD.

código 6.3: Método para insertar un Couplet en un nD-EVM.

```
template<typename valueType>
void nDEVM<valueType>::putCouplet(nDEVM * couplet){
    trieTree->putCouplet(couplet->trieTree);
}
```

Como se observa, también se define el método para realizar la operación equivalente en la clase TrieTree. Esto porque de manera jerárquica, la operación se realiza a nivel de los árboles Trie.

En el código 6.4 se define el método para la lectura de un Couplet de un nD-EVM. Esta lectura se realiza de manera ordenada. Es decir, se obtiene el Couplet (n-1)D según el ordenamiento sobre la primera dimensión del nD-EVM.

código 6.4: Método para obtener un Couplet de un nD-EVM.

```
template<typename valueType>
nDEVM<valueType>* nDEVM<valueType>::readCouplet(){
    return new nDEVM<valueType>(trieTree->readCouplet());
}
```

No obstante, se define este método para la clase TrieTree. Ello debido a que para formar el modelo (n-1)D-EVM del Couplet (n-1)D, se debe obtener un TrieTree con el que se construye el objeto nDEVM. En este método se observa la funcionalidad del apuntador doble *coupletIndex*. Ya que inicialmente, dicho apuntador hace referencia al nodo raíz del árbol Trie, y cada vez que se realiza una lectura de Couplets, este apuntador se actualiza al siguiente nodo en la primera dimensión. Así se garantiza una exploración de Couplets de manera ordenada.

Con lo anterior, para verificar si se ha llegado al final del nD-EVM, durante la exploración de Couplets, se utiliza el método *endEVM()* (código 6.5). El cual indicará si el apuntador *coupletIndex* ha llegado al final del nDEVM.

código 6.5: Método para verificar si se llegó al final del nD-EVM.

```
template<typename valueType>
bool nDEVM<valueType>::endEVM(){
    return trieTree->endTrie();
}
```

Sin embargo, para evitar problemas en exploraciones futuras, al finalizar una exploración, el apuntador *coupletIndex* debe reiniciarse a su estado original, en donde se hace referencia al nodo raíz del árbol Trie. Ello se logra mediante el método del código 6.6.

código 6.6: Reestablecimiento del apuntador coupletIndex.

```
template<typename valueType>
void nDEVM<valueType>::resetCoupletIndex(){
    trieTree->resetCoupletIndex();
}
```

El método para establecer la coordenada en la primera dimensión para un Couplet (n-1)D y así quede inmerso en el espacio nD, se muestra en el código 6.7.

código 6.7: Método para agregar una coordenada en la primera dimensión

```
template<typename valueType>
void nDEVM<valueType>::setCoord(valueType coord){
    trieTree->setCoord(coord);
}
```

En este método, se inserta un nuevo nodo *trieNode* con el valor de la coordenada, el cual será el nodo raíz del árbol Trie. De esta manera se obtiene el Couplet (n-1)D inmerso en un espacio nD.

Por otro lado, para obtener la coordenada de la primera dimensión, asociada a un Couplet (n-1)D inmerso en un espacio nD, se utiliza el método del código 6.8.

código 6.8: Método para obtener la coordenada de la primera dimensión de un Couplet

```
template<typename valueType>
valueType nDEVM<valueType>::getCoord(){
    return trieTree->getCoord();
}
```

En el método del código 6.9, se realiza la operación XOR regularizada. Esto se realiza utilizando la implementación descrita en la sección A.6. De manera genérica se denomina a este método como *mergeXOR*, ya que realiza la operación XOR de manera directa sobre los vértices en el árbol Trie. El primer operando para el método mergeXOR es el objeto nD-EVM del cual se realiza la llamada, el segundo operando es el que se recibe como parámetro.

código 6.9: Método para realizar la operación XOR regularizada entre dos nD-EVMs

```
template<typename valueType>
nDEVM<valueType>* nDEVM<valueType>::mergeXOR(nDEVM* otherEVM){
    if(isEmpty() and !(otherEVM->isEmpty()))
        return otherEVM->cloneEVM();

    if(!isEmpty() and otherEVM->isEmpty())
        return cloneEVM();

    if(isEmpty() and otherEVM->isEmpty()){
        return new nDEVM();
    }

    TrieTree<valueType> *resultTrie = trieTree->mergeXOR(otherEVM->trieTree);
    nDEVM<valueType> *resultEVM = new nDEVM<valueType>(resultTrie);
    resultTrie = NULL;
    return resultEVM;
}
```

Como se observa, inicialmente se analizan tres posibles escenarios:

- Cuando el nD-EVM del primer operando está vacío y el segundo no lo esta (línea 3). En este caso se retorna un clon del segundo operando.
- Por otro lado, en el caso contrario se retorna un clon del nD-EVM del primer operando, véase la línea 6.
- Por último, cuando ambos nD-EVMs están vacíos, se retorna un nD-EVM vacío (línea 9).

Posteriormente, se realiza el procedimiento descrito en la sección A.6. Y con el TrieTree obtenido, se construye un nDEVM resultante, el cual es retornado.

6.1.1. Obtención de Secciones a partir de Couplets y viceversa

En el código 6.10 se muestran los métodos para obtener secciones a partir de Couplets y viceversa. Esto se realiza mediante la operación mergeXOR del código 6.9. Los parámetros de entrada para el método *getSection* son una Sección y el Couplet correspondiente. Por otro lado, para el método *getCouplet* sus parámetros de entrada son dos Secciones consecutivas.

código 6.10: Métodos para obtener Secciones a partir de Couplets y viceversa

```
1 template<typename valueType>
2 nDEVM<valueType>* nDEVM<valueType>::getSection(nDEVM* section, nDEVM *couplet){
3     return section->mergeXOR(couplet);
4 }
5
6 template<typename valueType>
7 nDEVM<valueType>* nDEVM<valueType>::getCouplet(nDEVM* section1, nDEVM *section2){
8     return section1->mergeXOR(section2);
9 }
```

6.1.2. Obtención de una Secuencia de Secciones y una Secuencia de Couplets

En esta sección se presenta la implementación del método que obtiene, a partir de un nD-EVM, un nD-EVM que contiene una secuencia ordenada de las Secciones del nD-EVM original. Lo cual se logra utilizando los métodos de la sección 6.10. Este método se muestra en el código 6.11, en donde se tiene un método de donde se efectúa la llamada principal (línea 2) y otro método que realiza la exploración de Couplets y obtención de Secciones (línea 9).

código 6.11: Obtención de la secuencia de Secciones para un nD-EVM

```
1  template<typename valueType>
2  nDEVM<valueType>* nDEVM<valueType>::EVMSectionSequence(){
3      nDEVM* sectionSequence = new nDEVM();
4      EVMSectionSequence(&sectionSequence);
5      return sectionSequence;
6  }
7
8  template<typename valueType>
9  void nDEVM<valueType>::EVMSectionSequence(nDEVM** sectionSequence){
10     nDEVM* currentCouplet = new nDEVM();
11     nDEVM* prevSection = new nDEVM();
12     nDEVM* currentSection = new nDEVM();
13     currentCouplet = readCouplet();
14     int i = 0;
15     while(!endEVM()){
16         currentSection = getSection(prevSection,currentCouplet);
17
18         // Procesamiento
19         if(i > 0){
20             prevSection->setCoord(i - 1);
21             (*sectionSequence)->putSection(prevSection);
22         }
23
24         currentSection->EVMFile("Section",i);
25         prevSection = currentSection;
26         currentCouplet = readCouplet();
27         i++;
28     }
29     currentSection->setCoord(i - 1);
30     (*sectionSequence)->putSection(currentSection);
31     (*sectionSequence)->resetCoupletIndex();
32     resetCoupletIndex();
33 }
```

Como se observa en el código 6.11, el contador de la línea 14 se utiliza para preservar el ordenamiento de las secciones. Este ordenamiento se efectúa sobre la primera coordenada, ya que para un nD-EVM, una Sección es un (n-1)D-EVM, por ello al insertar la coordenada en base al contador, se obtiene un (n-1)D-EVM inmerso en un espacio nD. De esta manera, el método *putSection* realiza la misma operación que la del método *putCouplet*.

Por otro lado, con el nD-EVM formado con la secuencia de Secciones, es posible obtener una secuencia ordenada de Couplets. Esto tiene utilidad para realizar una comparación entre la secuencia de Couplets obtenida y la secuencia de Couplets del nD-EVM original. De esta manera se verifica que todos los métodos implementados realicen su tarea de manera satisfactoria, además se proporciona el flujo de operación para otros métodos, los cuales requieren operar Secciones del nD-EVM.

En el código 6.12 se muestra el método para la obtención de una secuencia ordenada de Couplets, a partir de un nD-EVM que contiene una secuencia de Secciones.

código 6.12: Método para obtener la secuencia de Couplets

```
template<typename valueType>
nDEVM<valueType>* nDEVM<valueType>::EVMCoupletSequence(){
    nDEVM* coupletSequence = new nDEVM();
    EVMCoupletSequence(&coupletSequence);
    return coupletSequence;
}

template<typename valueType>
void nDEVM<valueType>::EVMCoupletSequence(nDEVM** coupletSequence){
    nDEVM* currentCouplet = new nDEVM();
    nDEVM* prevSection = new nDEVM();
    nDEVM* currentSection = readSection();
    int i = 0;
    while(true){
        currentCouplet = getSection(prevSection,currentSection);

        // Procesamiento
        currentCouplet->EVMFile("Couplet",i);
        currentCouplet->setCoord(i);
        (*coupletSequence)->putCouplet(currentCouplet);

        if(endEVM()){
            currentCouplet = currentSection->cloneEVM();
            currentCouplet->setCoord(i+1);
            (*coupletSequence)->putCouplet(currentCouplet);
            break;
        }
        prevSection = currentSection;
        currentSection = readSection();
        i++;
    }
    (*coupletSequence)->resetCoupletIndex();
    resetCoupletIndex();
}
```

6.1.3. Operaciones Booleanas

En este apartado se presenta la implementación en *C++* del algoritmo de Operaciones Booleanas 3.7.1, la cual se muestra en el código 6.13. Las Operaciones Booleanas implementadas son las siguientes: $\{\cup^*, \cap^*, -^*, \otimes^*\}$. Esta implementación considera los siguientes pasos básicos:

- Las operaciones se efectúan entre las secciones de cada nD-EVM. En este sentido, el ordenamiento es importante, por lo que se realiza la Operación Booleana entre secciones según el orden en la primera coordenada.
- Por lo anterior, se toma como primer Sección a la que se encuentre en una posición menor en la primera coordenada. Si ambas secciones se encuentran sobre la misma coordenada, entonces el orden es indistinto.
- Este procedimiento se realiza de manera recursiva, obteniendo las Operaciones Booleanas entre secciones hasta llegar al caso base, cuando se obtiene un 1D-EVM. En donde se operan segmentos 1D.
- Al retornar del caso base, la sección resultante es utilizada para construir el Couplet correspondiente al nD-EVM resultante. El cual es insertado en dicho nD-EVM resultante, hasta obtener la versión final de este resultado.

código 6.13: Método de Operaciones Booleanas.

```
template<typename valueType>
nDEVM<valueType>* nDEVM<valueType>::booleanOperation(nDEVM* evm2, string op){
    int dim = dimDepth();
    return booleanOperation(this,evm2,op,dim);
}

template<typename valueType>
nDEVM<valueType>* nDEVM<valueType>::booleanOperation(nDEVM *p, nDEVM *q, string op, int n){
    nDEVM<valueType> *pSection,*pPrevSection, *qSection,*qPrevSection, *couplet;
    nDEVM<valueType> *result, *rPrevSection, *rCurrentSection;
    bool fromP, fromQ;
    valueType coord;
    if(n == 1){
        return booleanOperation(p,q,op);
    }
    pSection = new nDEVM<valueType>();
    qSection = new nDEVM<valueType>();
    rCurrentSection = new nDEVM<valueType>();
    result = new nDEVM<valueType>();
    while(!(p->endEVM()) and !(q->endEVM())){
        nextObject(p,q,&coord,&fromP,&fromQ);
        if(fromP){
            couplet = p->readCouplet();
            pPrevSection = pSection;
            pSection = getSection(pSection,couplet);
            delete pPrevSection;
            delete couplet;
        }
        if(fromQ){
            couplet = q->readCouplet();
            qPrevSection = qSection;
            qSection = getSection(qSection,couplet);
            delete qPrevSection;
            delete couplet;
        }

        rPrevSection = rCurrentSection;
        rCurrentSection = booleanOperation(pSection, qSection,op,n-1);

        couplet = getCouplet(rPrevSection,rCurrentSection);

        if(!couplet->isEmpty()){
            couplet->trieTree->isCouplet = true;
            couplet->setCoord(coord);

            result->putCouplet(couplet);
            delete couplet;
        }

        delete rPrevSection;
    }
    while(!(p->endEVM())){
        nDEVM<valueType> *coupletCopy;
        if(putCoupletByOp(op,1)){
            coord = (*(p->trieTree->coupletIndex))->value;
            couplet = p->readCouplet();

            coupletCopy = couplet->cloneEVM();
            coupletCopy->trieTree->isCouplet = true;
            coupletCopy->setCoord(coord);

            result->putCouplet(coupletCopy);

            delete couplet;
            delete coupletCopy;
        }else
            break;
    }

    while(!(q->endEVM())){
        nDEVM<valueType> *coupletCopy;
        if(putCoupletByOp(op,2)){
            coord = (*(q->trieTree->coupletIndex))->value;
            couplet = q->readCouplet();

            coupletCopy = couplet->cloneEVM();
            coupletCopy->trieTree->isCouplet = true;
            coupletCopy->setCoord(coord);

```

```
80              result->putCouplet(coupletCopy);
81              delete couplet;
82              delete coupletCopy;
83          }else
84              break;
85      }
86      p->resetCoupletIndex();
87      q->resetCoupletIndex();
88      result->resetCoupletIndex();
89
90      delete pSection;
91      delete qSection;
92      delete rCurrentSection;
93
94      return result;
95  }
```

El método de Operaciones Booleanas del código 6.13, presenta el siguiente flujo de operación:

- La fase de declaración e inicialización de variables se presenta en las líneas 9-12 y líneas 16-19.
- La estructura de repetición de la línea 20 permite realizar la exploración de los Couplets y Secciones de ambos operandos hasta alcanzar el último Couplet de alguno de éstos.
- En las líneas 21-35 se evalúa y se obtienen las Secciones de ambos operandos en base a su ordenamiento sobre el primer eje coordenado.
- En la línea 38 se realiza la llamada recursiva a este método para operar las Secciones de ambos operandos. El caso base de la recursión es para 1D-EVMs como se ha mencionado en varias ocasiones, este caso base se procesa en la línea 14.
- Como se observa en las líneas 40-48, una vez obtenida la Seccion resultante de aplicar la Operación Booleana se obtiene un Couplet y se inserta en el EVM resultante *result*.
- En las estructuras de repetición de las líneas 52 y 70 se evalúa si se insertan o no en el resultado los Couplets restantes del operando que aún no termina su exploración, esto considerando que del otro operando se ha finalizado su exploración.
- Por último se inicializan los apuntadores *coupletIndex* de cada operando, esto para que se pueda realizar alguna otra operación.

El método *nextObject* que se muestra en la línea 21 del código 6.13, realiza una consulta para conocer qué sección es la siguiente en procesar. Esto se realiza en base al ordenamiento sobre la primera coordenada. Con ello se preserva el orden en las operaciones y se garantiza una construcción correcta del nD-EVM resultante. En el código 6.14 se muestra este método, y los casos que se evalúan son los siguientes:

- En la línea 3 se considera el caso en que la siguiente Sección del primer operando se encuentra en una coordenada menor que la Sección del segundo operando, esto con respecto al primer eje coordenado.
- En la línea 10 se muestra el caso en que la coordenada de la Sección del primer operando es mayor a la coordenada del segundo operando.
- Por último, en la línea 17 se considera el caso en que ambas Secciones se encuentran en la misma coordenada.

código 6.14: Método para obtener la siguiente Sección.

```
1  template<typename valueType>
2  void nDEVM<valueType>::nextObject(nDEVM *p, nDEVM *q,valueType *coord,bool *fromP, bool *fromQ){
3      if((*(p->trieTree->coupletIndex))->value < (*(q->trieTree->coupletIndex))->value){
4          *coord = (*(p->trieTree->coupletIndex))->value;
5          *fromP = true;
6          *fromQ = false;
7          return;
8      }
9
10     if((*(q->trieTree->coupletIndex))->value < (*(p->trieTree->coupletIndex))->value){
```

```
11          *coord = (*(q->trieTree->coupletIndex))->value;
12          *fromQ = true;
13          *fromP = false;
14          return;
15      }
16
17      if((*(p->trieTree->coupletIndex))->value == (*(q->trieTree->coupletIndex))->value){
18          *coord = (*(p->trieTree->coupletIndex))->value;
19          *fromQ = true;
20          *fromP = true;
21          return;
22      }
23  }
```

Ahora, la llamada al método de Operaciones Booleanas de la línea 14 en el código 6.13, hace referencia al caso base, en donde se obtiene el resultado de la operación para dos secciónes 1D. Este método consiste en un mapeo de una cadena que contiene el nombre de la operación a efectuar, y el método en sí. En el código 6.15, se muestra la implementación de este método.

código 6.15: Mapeo de Operaciones Booleanas para el caso 1D.

```
1  template<typename valueType>
2  nDEVM<valueType>* nDEVM<valueType>::booleanOperation(nDEVM *section1, nDEVM *section2, string op){
3      if(op.compare("union") == 0){
4          return unionOperation(section1,section2);
5      }
6
7      if(op.compare("intersection") == 0){
8          return intersectionOperation(section1,section2);
9      }
10
11     if(op.compare("difference") == 0){
12         return differenceOperation(section1,section2);
13     }
14
15     if(op.compare("xor") == 0){
16         return xorOperation(section1,section2);
17     }
18     cout<<"La operacion: "<<op<<", no existe...";
19     exit(-1);
20 }
```

6.1.4. Cálculo de Contenido

La implementación del algoritmo de cálculo de Contenido se presenta en al código 6.16, en donde se observa que en la línea 2 se define el método de la llamada principal y en la línea 9 se define el método general.

código 6.16: Código para el cálculo de contenido de un nD-EVM.

```
1  template<typename valueType>
2  valueType nDEVM<valueType>::content(){
3      int dim = dimDepth();
4      nDEVM<valueType> *p = this;
5      return content(&p, dim);
6  }
7
8  template<typename valueType>
9  valueType nDEVM<valueType>::content(nDEVM **p, int n){
10     valueType cont = 0, coordC1,coordC2;
11     nDEVM<valueType> *couplet;
12     nDEVM<valueType> *currentSection,*prevSection;
13
14     if( (*p)->isEmpty() ){
15         return 0;
16     }
17
18     if(n == 1){
19         return (*p)->length();
20     }
21
22     prevSection = new nDEVM<valueType>();
23
24     coordC1 = (*p)->getCoord();
```

```
25      couplet = (*p)->readCouplet();
26
27      while( !((*p)->endEVM()) ){
28          coordC2 = (*p)->getCoord();
29
30          currentSection = getSection(prevSection,couplet);
31
32          cont = cont + content(&currentSection,n-1)*(coordC2 - coordC1);
33
34          coordC1 = coordC2;
35          delete couplet;
36          couplet = (*p)->readCouplet();
37
38          delete prevSection;
39          prevSection = currentSection;
40      }
41      delete currentSection;
42      delete couplet;
43      (*p)->resetCoupletIndex();
44      return cont;
45  }
```

En el código 6.16 se observa que en las líneas 14 y 18 se consideran los casos base del algoritmo de Contenido. El primer caso base considera el caso en el que el nD-EVM dado se encuentra vacío, y el segundo caso se presenta cuando se calcula el contenido de un 1D-EVM, en donde se obtiene la suma de las longitudes de los segmentos 1D mediante la función *length*.

En la estructura de repetición de la línea 27 se exploran todos los Couplets del nD-EVM, y el cálculo de contenido de los Slices se efectúa en las líneas 30 y 32, en donde se realiza el producto del contenido de la correspondiente Sección y la distancia entre los Couplets obtenida mediante $coordC2$ y $coordC1$.

6.1.5. Cálculo de Compacidad Discreta

En la implementación del cálculo de Compacidad Discreta, es importante considerar que se realiza en varias etapas, la primera consiste en obtener los contactos internos de todos los Slices ordenados con respecto a la primera coordenada (función de la línea 2), posteriormente se obtiene un ordenamiento diferente de los ejes coordenados y se realiza nuevamente el paso anterior (función de la línea 40), y por último se aplica la ecuación 3.6 que corresponde al cálculo de Compacidad Discreta (línea 62).

código 6.17: Métodos para el cálculo de la Compacidad Discreta de un nD-EVM.

```
1   template<typename valueType>
2   valueType nDEVM<valueType>::internalContacts(nDEVM * p,int n){
3       nDEVM<valueType> * couplet;
4       nDEVM<valueType> * prevSection,*currentSection;
5       nDEVM<valueType> * sectionInt;
6       valueType c1Coord,c2Coord,nCoords;
7       valueType iContacts = 0;
8
9       prevSection = new nDEVM<valueType>();
10      c1Coord = p->getCoord();
11      couplet = p->readCouplet();
12      while( !(p->endEVM()) ){
13          currentSection = getSection(prevSection,couplet);
14
15          c2Coord = p->getCoord();
16          nCoords = c2Coord - c1Coord - 1;
17
18          iContacts = iContacts + nCoords * content(&currentSection,n-1);
19
20          sectionInt = prevSection->booleanOperation(currentSection,"intersection",n-1);
21          iContacts = iContacts + content(&sectionInt,n-1);
22
23          delete sectionInt;
24          delete prevSection;
25          delete couplet;
26
27          prevSection = currentSection;
28
29          c1Coord = c2Coord;
```

```
30          couplet = p->readCouplet();
31      }
32      p->resetCoupletIndex();
33      delete currentSection;
34      delete couplet;
35
36      return iContacts;
37  }
38
39  template<typename valueType>
40  valueType nDEVM<valueType>::totalInternalContacts(){
41      int dim = dimDepth();
42      valueType Lc = 0;
43      nDEVM<valueType> *p;
44      nDEVM<valueType> *prevP;
45
46      Lc = Lc + internalContacts(this,dim);
47
48      p = dimLeftShift();
49
50      for(int i = 0; i < (dim-1); i++){
51          Lc = Lc + internalContacts(p,dim);
52          prevP = p;
53          p = p->dimLeftShift();
54
55          delete prevP;
56      }
57      delete p;
58      return Lc;
59  }
60
61  template<typename valueType>
62  double nDEVM<valueType>::discreteCompactness(valueType lMin,valueType lMax){
63      valueType Lc = totalInternalContacts();
64
65      return ((double)(Lc - lMin))/((double)(lMax - lMin));
66  }
```

En la función *internalContacts* de la línea 2 del código 6.17 se obtienen los contactos internos de la siguiente manera:

- La estructura de repetición de la línea 12 permite explorar todos los Couplets del nD-EVM.
- Se obtiene la siguiente Sección con respecto al primer eje coordenado y las coordenadas de los Couplets correspondientes (líneas 13- 16).
- En la línea 18 se obtienen los contactos internos del Slice correspondiente, en la línea 20 se obtienen los contactos internos entre dos Slices consecutivos y en la línea 21 se realiza la suma acumulada de éstos.

La función *totalInternalContacts* de la línea 40 del código 6.17 obtiene los contactos internos totales, apoyándose en la función *internalContacts* y los diferentes ordenamientos de los ejes coordenados. Para obtener las versiones del nD-EVM con diferente ordenamiento se utiliza la función *dimLeftShift* de la línea 53, la cual realiza un ordenamiento de manera que se obtiene una versión donde los ejes ordenados están desplazados a la izquierda.

Por ejemplo si se tiene un 3D-EVM con el siguiente orden de ejes coordenados $\{x_1, x_2, x_3\}$, al aplicar la función *dimLeftShift* se obtendría un nD-EVM con el siguiente orden de ejes coordenados $\{x_2, x_3, x_1\}$, por lo que se requieren de un total de $3-1$ desplazamientos a la izquierda para explorar todos los ordenamientos necesarios (línea 50).

6.2 Implementación de los Métodos para Procesamiento de Video

6.2.1. Generación de una animación en el nD-EVM a partir de una secuencia de frames

En base al procedimiento general para obtener la representación de una secuencia de frames en el nD-EVM, como primer paso se debe obtener la representación en el nD-EVM de un frame. Se debe notar que el algoritmo 5.2 especifica el procedimiento genérico para obtener la representación en el nD-EVM de un frame con cualquier esquema de color. No obstante, para los fines de este trabajo de investigación solo se procesan frames con esquemas de de color en escala de grises y RGB. El método para representar un frame en un objeto nDEVM se presenta en el código 6.18.

código 6.18: Código para la generar la representación en el nD-EVM de un frame.

```
1  template<typename valueType>
2  void nDEVM<valueType>::loadImage(string fileName){
3      BMP bmpImage(fileName);
4      int colors = (int)bmpImage.header.bitsPerPixel/8;
5      valueType *hyperPrismLengths = new valueType[2+colors]; // - [X,Y,G|(R,G,B)]
6      valueType *hyperPrismBase = new valueType[2+colors]; // - [X,Y,G|(R,G,B)]
7      unsigned int dataIdx = 0;
8
9      cout<<"Loading Frame: "<<fileName<<endl;
10     hyperPrismLengths[0] = 1;
11     hyperPrismLengths[1] = 1;
12     for(int k = 0; k < colors; k++){
13         hyperPrismBase[2+k] = 0;
14     }
15
16     for(int i = 0; i < bmpImage.header.height; i++){
17         hyperPrismBase[1] = i;
18
19         for(int j = 0; j < bmpImage.header.width*colors; j+= colors){
20             hyperPrismBase[0] = (int)(j/colors);
21
22             for(int k = 0; k < colors; k++){
23                 hyperPrismLengths[2+k] = (valueType)(bmpImage.pImageData[dataIdx] + 1 );
24                 dataIdx++;
25             }
26             populateFrameHyperPrism(&hyperPrismBase,2+colors,0,&hyperPrismLengths);
27         }
28     }
29     cout<<"Frame loaded..."<<endl;
30     delete [] hyperPrismLengths;
31     delete [] hyperPrismBase;
32 }
33
34 template<typename valueType>
35 void nDEVM<valueType>::populateFrameHyperPrism(valueType **hyperPrismBase,int frameDim,int
       currentDim,
36         valueType ** hyperPrismLengths){
37     if(!(currentDim < frameDim)){
38         insertVertex(*hyperPrismBase,frameDim);
39         return;
40     }
41     populateFrameHyperPrism(hyperPrismBase,frameDim,currentDim+1,hyperPrismLengths);
42     (*hyperPrismBase)[currentDim] = (*hyperPrismBase)[currentDim] + (*hyperPrismLengths)[currentDim
       ];
43     populateFrameHyperPrism(hyperPrismBase,frameDim,currentDim+1,hyperPrismLengths);
44     (*hyperPrismBase)[currentDim] = (*hyperPrismBase)[currentDim] - (*hyperPrismLengths)[currentDim
       ];
45 }
```

En el método *loadImage* del código 6.18 se realiza una exploración de todos los pixeles dentro del frame, con ello se obtienen los correspondientes hiper-prismas cuya unión formará el nD-EVM final. En las líneas 5 y 6 se observa que las variables *hyperPrismBase* y *hyperPrismLengths* se utilizan para proporcionar la base y longitud del hiper-prisma correspondiente al pixel actual en la exploración del frame.

La inicialización de los arreglos *hyperPrismBase* y *hyperPrismLengths* se efectúa en las líneas 10-14, en donde se establece que las longitudes en el plano (x, y) corresponden a la caja unitaria y la

base de los hiper-prismas se establecen al mínimo para el esquema de color. Por otro lado, en las líneas 16 y 19 se realiza la exploración de los pixeles del frame. En las líneas 17 y 20 se ajustan las coordenadas iniciales de los hiper-prismas en base a las coordenadas de los pixeles. Por último la función *populateFrameHyperPrism* que se observa en la línea 26 y se define en la línea 35 obtiene todos los vértices del hiper-prisma en base a la configuración establecida anteriormente.

Con lo anterior, la función *generateAnimation* del código 6.19 presenta el flujo para obtener la representación de una secuencia de frames en el nD-EVM. En la línea 8 se observa la estructura de repetición para explorar los frames considerando el frame inicial 0 y el frame final está dado por el parámetro de entrada *endFrame*. Como se observa en la línea 15, se obtiene la diferencia entre un frame previo y el frame actual, con lo que se forma el Couplet correspondiente y éste es almacenado en disco duro en forma de archivo binario (línea 16).

código 6.19: Generación de una animación en el nD-EVM a partir de una secuencia de Frames.

```
1  template<typename valueType>
2  void nDEVM<valueType>::generateAnimation(string framePrefix, valueType endFrame){
3      nDEVM<valueType> *currentFrame, *prevFrame,*diffFrame;
4      int time;
5      prevFrame = new nDEVM<valueType>();
6      string frameName;
7
8      for(int i = 0; i <= endFrame; i++){
9          currentFrame = new nDEVM<valueType>();
10         time = i;
11
12         frameName = framePrefix + to_string(i)+".bmp";
13
14         currentFrame->loadImage(frameName);
15         diffFrame = getCouplet(prevFrame,currentFrame);
16         diffFrame->saveEVM("frameCouplet",time);
17
18         delete prevFrame;
19         delete diffFrame;
20         prevFrame = currentFrame;
21     }
22     prevFrame->saveEVM("frameCouplet",time+1);
23     delete prevFrame;
24 }
```

6.2.2. Generación de máscaras nD

En el código 6.20 se muestra el código para generar el nD-EVM de una máscara nD, como se observa en la función de la línea 2 se inicializa la máscara en base a la configuración dada, por otro lado la función de la línea 41 obtiene una máscara con una configuración similar a la original pero con los componentes de color en su valor mínimo. Como se mencionó en el apartado 5.2.2 dicha máscara es necesaria para calcular el parámetro L_{Min} de la ecuación de DC.

código 6.20: Generación del nD-EVM de una máscara nD

```
1  template<typename valueType>
2  void nDEVM<valueType>::maskInit(int xLength, int yLength, int timeLength,
3          int colorComponents, int colorCompSize){
4      maskMin[0] = 0;
5      maskMax[0] = timeLength;
6      maskMin[1] = 0;
7      maskMax[1] = xLength;
8      maskMin[2] = 0;
9      maskMax[2] = yLength;
10
11     valueType colorCompMax = pow(2,colorCompSize*8);
12     valueType * maskHyperPrism = new valueType[3+colorComponents];
13     valueType * maskLengths = new valueType[3+colorComponents];
14
15     maskHyperPrism[0] = 0;
16     maskLengths[0] = timeLength;
17
18     maskHyperPrism[1] = 0;
19     maskLengths[1] = xLength;
20
21     maskHyperPrism[2] = 0;
```

```
22      maskLengths[2] = yLength;
23
24      for(int i = 0; i < colorComponents; i++){
25          maskHyperPrism[3+i] = 0;
26          maskLengths[3+i] = colorCompMax;
27      }
28      populateMask(&maskHyperPrism,3+colorComponents,0,&maskLengths);
29
30      LcMax = totalInternalContacts();
31
32      nDEVM<valueType> *otherMask = new nDEVM<valueType>();
33      otherMask->minMask(xLength,yLength,timeLength,colorComponents);
34      LcMin = otherMask->totalInternalContacts();
35      delete [] maskHyperPrism;
36      delete [] maskLengths;
37      delete otherMask;
38  }
39
40  template<typename valueType>
41  void nDEVM<valueType>::minMask(int xLength, int yLength, int timeLength,
42          int colorComponents){
43      valueType * maskHyperPrism = new valueType[3+colorComponents];
44      valueType * maskLengths = new valueType[3+colorComponents];
45
46      maskHyperPrism[0] = 0;
47      maskLengths[0] = timeLength;
48
49      maskHyperPrism[1] = 0;
50      maskLengths[1] = xLength;
51
52      maskHyperPrism[2] = 0;
53      maskLengths[2] = yLength;
54
55      for(int i = 0; i < colorComponents; i++){
56          maskHyperPrism[3+i] = 0;
57          maskLengths[3+i] = 1;
58      }
59      populateMask(&maskHyperPrism,3+colorComponents,0,&maskLengths);
60      delete [] maskHyperPrism;
61      delete [] maskLengths;
62  }
```

En las líneas 11-27 del código 6.20, se realiza la inicialización de los parámetros de la máscara. En particular se establece la configuración de los vectores *maskHyperPrism* y *maskLengths*, que considerando que la máscara puede ser vista como un hiper-prisma, estos vectores contienen la coordenada inicial y las longitudes del hiper-prisma de la máscara. Las coordenadas iniciales de la máscara se establecen en el origen, por otro lado las longitudes se establecen en base a la configuración dada en los parámetros y por los componentes de color.

Como se observa en las líneas 30 y 34, en el objeto nDEVM de la máscara se almacenan los valores correspondientes a L_{Max} y L_{Min} que son utilizados para el cálculo de DC (ecuación 3.6), esto se realiza para evitar su cálculo de manera dinámica, lo cual sería ineficiente. Véase el código 6.1.

6.2.3. Convolución de una máscara y una animación en el nD-EVM

En este apartado se presenta la implementación del proceso de convolución descrito en el apartado 5.2.3, en donde se obtiene el conjunto de datos $DCValues$. Esta implementación difiere al algoritmo original de convolución, ya que la cantidad de sub-animaciones que se pueden obtener de una animación puede ser muy elevado, por lo que el manejo del arreglo $DCValues$ en memoria RAM sería ineficiente u ocasionaría mal funcionamiento de los equipos de cómputo.

El enfoque empleado para la implementación del algoritmo de convolución, radica en almacenar una serie de archivos binarios que contienen de manera ordenada los valores de DC. El ordenamiento de los valores de DC se basa en el orden de exploración de las sub-animaciones, con lo que es posible recuperar la configuración de la máscara que originalmente se utilizó para extraer cada sub-animación.

Por otro lado, para aplicar la operación intersección entre la máscara y la animación se utiliza un enfoque diferente al que normalmente se aplica con el algoritmo de Operaciones Booleanas. En

este caso se obtienen las Secciones (frames en el nD-EVM) de la animación que son suficientes para aplicar la operación intersección con la máscara. Con lo anterior dichas Secciones son almacenadas en memoria RAM y así no se requiere calcular éstas de manera dinámica, lo cual sería muy costoso computacionalmente hablando, debido a que se tienen que aplicar desplazamientos unitarios en la máscara hasta explorar toda la animación, y por ende por cada desplazamiento se tendría que calcular nuevamente las Secciones según el algoritmo de Operaciones Booleanas. En el código 6.21 se muestra la implementación del proceso de convolución.

código 6.21: Método que realiza el proceso de convolución entre una máscara y una animación representada en el nD-EVM.

```
template<typename valueType>
void nDEVM<valueType>::maskAnimConv(nDEVM * mask,int endFrame,valueType _xMax,
        valueType _yMax){
    nDEVM<valueType> *currentResult;
    nDEVM<valueType> *sectionSeq = new nDEVM<valueType>();

    string fileName = "";
    string partName = "";
    double *dcPtr = new double;
    unsigned int i = 0;
    unsigned int dcFile = 0;
    unsigned int dcPart = mask->getCoord();

    if(mask->maskMax[0] > (endFrame + 1)){
        cout<<"Se ha llegado al final de la animacion..."<<endl;
        return;
    }

    sectionSeq = maskAnimSections(mask,endFrame+1);

    fileName = "..\\dcFiles\\Part"+to_string(dcPart)+"\\dcFiles.txt";

    ofstream dcFiles( fileName );
    if ( ! dcFiles.is_open() ){
        cout << "El archivo "<<fileName<<" no se pudo abrir!" << '\n';
        return;
    }

    partName = "..\\dcFiles\\Part"+to_string(dcPart)+"\\dcFile"+to_string(dcFile)+".dc";
    ofstream outputFile( partName.c_str(),ios_base::out|ios_base::binary );
    if ( ! outputFile.is_open() ){
        cout << "El archivo: "+partName+" no se pudo abrir!!" << '\n';
        return;
    }

    dcFiles<<partName<<'\n';
    cout<<"Computing Part"<<dcPart<<"..."<<endl;
    // - Recorrido en y
    while(mask->maskMax[2] <= _yMax){
        // - Recorrido en x
        while(mask->maskMax[1] <= _xMax){
            currentResult = maskIntersection(mask,sectionSeq);

            *dcPtr = currentResult->discreteCompactness(mask->LcMin,mask->LcMax);
            delete currentResult;
            outputFile.write((char *) dcPtr,sizeof(double));

            mask->EVMTraslation(1,1); // - Desplazamiento en X
            i++;
            if(i >= 10000){
                outputFile.close();
                dcFile++;
                cout<<partName<<" ... DONE!!"<<endl;
                partName = "..\\dcFiles\\Part"+to_string(dcPart)+"\\dcFile"+to_string(dcFile)+".dc"
    ;
                outputFile.open( partName.c_str(),ios_base::out|ios_base::binary );
                if ( ! outputFile.is_open() ){
                    cout << "El archivo: "+partName+" no se pudo abrir!!" << '\n';
                    return;
                }
                dcFiles<<partName<<'\n';
                i = 0;
            }
        }
        mask->maskDimReset(1);
        mask->EVMTraslation(2,1);
    }
```

```
67      outputFile.close();
68      dcFiles.close();
69      mask->maskDimReset(1);
70      mask->maskDimReset(2);
71
72      delete dcPtr;
73      delete sectionSeq;
74  }
```

Para el código 6.21, en la línea 14 se muestra la condición para evaluar si se ha llegado al final de la animación con respecto a la variable temporal. En la línea 19 se obtiene la secuencia de Secciones requerida para la intersección mediante la función *maskAnimSections*, es importane notar que dicha función requiere la máscara original para determinar las Secciones necesarias, ello en base a la configuración y desplazamientos de dicha máscara. La operación intersección se efectúa mediante una versión modificada del algoritmo de Operaciones Booleanas del código 6.13, esta es la función *maskIntersection* de la línea 42.

En las líneas 21-27 y 29-34 se establecen los archivos iniciales para almacenar el conjunto $DCValues$. El archivo de la línea 21 contiene los nombres de los archivos binarios que se van a generar, esto se realiza debido a que los archivos binarios de DC almacenan una cantidad máxima de 10000 (línea 50), y no para todas las configuraciones de máscara y dimensiones de los frames se obtiene la misma cantidad de archivos de DC. Para este caso se realiza un barrido en los ejex x y y como se muestra en las líneas 39 y 41, ya que se considera que los desplazamientos en el tiempo por un método externo. En las líneas 44 y 46 se calcula y almacena el valor de DC obtenido para la sub-animación actual. Los desplazamientos unitarios de la máscara se realizan mediante la función *EVMTraslation* como se observa en las líneas 48 y 65.

En el código 6.22 presenta el método *maskAnimSections* que se utiliza para obtener la secuencia de Secciones requerida para la intersección de la animación y la máscara. Como se observa, este método es una versión modificada del método de extracción de la secuencia de Secciones del código 6.11. Las diferencias radican en que en este caso se realiza la lectura de los Couplets de la animación desde disco duro (línea 19), en general la exploración de Couplets en la animación se detiene cuando la máscara llega a su Couplet final (línea 14), y en las líneas 37-41 se almacenan las Secciones que se van a utilizar.

código 6.22: Obtención de las Secciones necesarias para la intersección de la animación y la máscara.

```
1  template<typename valueType>
2  nDEVM<valueType> * nDEVM<valueType>::maskAnimSections(nDEVM* mask,int endCouplet){
3      int iCouplet = 0;
4
5      nDEVM<valueType> *animSection,*animPrevSection, *couplet;
6      nDEVM<valueType> *sectionSequence;
7      bool fromAnim, fromMask,saveSection = false;
8      valueType coord;
9
10     animSection = new nDEVM<valueType>();
11
12     sectionSequence = new nDEVM<valueType>();
13
14     while(iCouplet <= endCouplet and !(mask->endEVM())){
15         animNextObject(iCouplet,mask,&coord,&fromAnim,&fromMask);
16
17         if(fromAnim){
18             couplet = new nDEVM<valueType>();
19             couplet->readEVM("frameCouplet" + to_string(iCouplet));
20             animPrevSection = animSection;
21             animSection = getSection(animPrevSection,couplet);
22             delete couplet;
23             delete animPrevSection;
24         }
25
26         if(fromMask){
27             couplet = mask->readCouplet();
28             if(!mask->endEVM()){
29                 saveSection = true;
30             }else{
31                 saveSection = false;
32             }
```

```
33              delete couplet;
34          }
35
36          if(saveSection){
37              nDEVM<valueType> *section;
38              section = animSection->cloneEVM();
39              section->trieTree->isCouplet = true;
40              section->setCoord(iCouplet);
41              sectionSequence->putCouplet(section);
42
43              delete section;
44          }
45
46          iCouplet++;
47      }
48      mask->resetCoupletIndex();
49      sectionSequence->resetCoupletIndex();
50
51      delete animSection;
52      return sectionSequence;
53  }
```

El método *maskIntersection* del código 6.23, presenta una versión modificada del algoritmo de Operaciones Booleanas, el cual aplica la operación intersección entre el nD-EVM que contiene la secuencia de Secciones y la máscara. Es sencillo deducir que la máscara solo contiene una Sección, ello en base a que una máscara es una hiper-caja con longitudes definidas en los ejes coordenados (x, y, t) así como para los componentes de color, dicha Sección se obtiene de manera directa en las líneas 17-18. Entonces la estructura de repetición de la línea 21 finaliza cuando se alcanza la última Sección del objeto *sectionSeq*. Debido a que el objeto *sectionSeq* contiene solo Secciones, no es necesario calcularlos como normalmente se realiza en el algoritmo de Operaciones Booleanas, tal como se muestra en la línea 22.

Para las Operaciones Booleanas subsecuentes, es decir entre las Secciones de la máscara y las Secciones de la animación, se utiliza el algoritmo de Operaciones Booleanas convencional (línea 25), ya que ahora los operandos presentan una configuración apropiada. En las líneas 30-33 se obtienen los Couplets y son insertados en el resultado de la operación intersección. Por último en las líneas 43-48 se obtiene el último Couplet del resultado.

código 6.23: Método para la operación intersección entre la máscara y la secuencia de Secciones.

```
1  template<typename valueType>
2  nDEVM<valueType> * nDEVM<valueType>::maskIntersection(nDEVM* mask, nDEVM* sectionSeq){
3
4      nDEVM<valueType> *animSection,*maskSection,*couplet;
5      nDEVM<valueType> *result, *rPrevSection, *rCurrentSection;
6
7      valueType coord;
8
9      int dim = mask->dimDepth();
10
11     maskSection = new nDEVM<valueType>();
12     rCurrentSection = new nDEVM<valueType>();
13
14     result = new nDEVM<valueType>();
15
16     coord = mask->getCoord();
17     couplet = mask->readCouplet();
18     maskSection = getSection(maskSection,couplet);
19     delete couplet;
20
21     while(!sectionSeq->endEVM()){
22         animSection = sectionSeq->readCouplet();
23
24         rPrevSection = rCurrentSection;
25         rCurrentSection = booleanOperation(animSection, maskSection,"intersection",dim-1);
26
27         couplet = getCouplet(rPrevSection,rCurrentSection);
28
29         if(!couplet->isEmpty()){
30             couplet->trieTree->isCouplet = true;
31             couplet->setCoord(coord);
32
33             result->putCouplet(couplet);
```

```
34              delete couplet;
35          }
36
37          delete rPrevSection;
38          delete animSection;
39
40          coord++;
41      }
42
43      if(!rCurrentSection->isEmpty()){
44          rCurrentSection->trieTree->isCouplet = true;
45          rCurrentSection->setCoord(coord);
46
47          result->putCouplet(rCurrentSection);
48      }
49
50      mask->resetCoupletIndex();
51      result->resetCoupletIndex();
52      sectionSeq->resetCoupletIndex();
53
54      delete maskSection;
55      delete rCurrentSection;
56
57      return result;
58  }
```

6.3 Generación de Ejecutables

Para facilitar la ejecución de los métodos implementados, se generaron archivos ejecutables de los métodos utilizados para el framework de segmentación. En este caso se cuentan con dos tipos de ejecutables: Primero se tienen los ejecutables obtenidos de los códigos en C++ del modelo nD-EVM y sus métodos, los métodos para segmentación de video y del SOM; por último, se cuentan con los ejecutables para la gestión de la ejecución de los anteriores. Es importante considerar que estos ejecutables obtienen la configuración de los frames, la máscara y el SOM de un archivo de texto, tal como se describe en la sección 5.2.6, de esta manera se facilita la ejecución en general.

A continuación se presenta la lista de los ejecutables generados a partir de los códigos en C++, y se proporciona la descripción de su funcionalidad:

- *AnimLoad.exe*: Obtiene la representación de una secuencia de frames en el modelo nD-EVM. En esta representación se obtienen solo los archivos correspondientes a los Couplets de la animación, en donde se considera lo establecido en la sección 5.1.3.
- *AnimConv.exe*: Una vez obtenidos los Couplets de la animación, este método permite realizar el proceso de convolución, en donde se consideran las condiciones establecidas en la sección 5.2.6 y el algoritmo 5.6. Con este ejecutable se generan archivos binarios que contienen los descriptores de DC para todas las sub-animaciones, es decir que se genera el conjunto *DCValues*.
- *Clustering.exe*: Realiza el proceso de agrupamiento de los valores de DC de las sub-animaciones. Al finalizar su ejecución se obtiene, para cada cluster, un archivo binario que contiene los índices de las sub-animaciones agrupadas.
- *ClusterContentNC.exe*: Con los archivos de los índices de las sub-animaciones generados con el ejecutable *Clustering.exe*, se obtiene el contenido de un cluster, es decir, se obtiene el 3D-EVM del cluster.
- *ClusterContent.exe*: Este ejecutable obtiene la información de color correspondiente a las regiones contenidas en el 3D-EVM generado para un cluster. De esta manera se obtiene la secuencia de Secciones (con información de color) de un cluster.
- *ClusterFrame.exe*: Permite generar una frame (imagen) de salida para cada Sección (con información de color) que contiene un cluster, y de esta manera visualizar su contenido.

Por otro lado, a continuación se muestra la lista de los archivos ejecutables generados en JAVA, los cuales gestionan la ejecución de los ejecutables mencionados anteriormente:

- *AnimConvLauncher.jar*: Gestiona la ejecución del archivo *AnimConv.exe* bajo el enfoque multithreading. La funcionalidad de este ejecutable se basa en lo descrito en la sección 5.2.6, en donde cada thread procesa la convolución de un desplazamiento en t de la máscara.
- *ClusterContentNCLauncher*: Realiza la gestión de la ejecución del archivo *ClusterContentNC.exe*, con lo cual se permite obtener los 3D-EVMs de los clusters de manera concurrente.
- *ClusterContentLauncher.jar*: Gestiona la ejecución del archivo *ClusterContent.exe* para obtener la secuencia de Secciones del EVM de cada cluster incluyendo la información de color de la animación original. El gestionamiento se realiza bajo el enfoque multithreading, con lo cual se permite procesar un cluster por cada thread.
- *ClusterFrameLauncher.jar*: Realiza la gestión de la ejecución del archivo *ClusterFrame.exe* bajo el enfoque multithreading, con lo cual se permite obtener de manera concurrente la secuencia de frames (imágenes) para cada cluster.
- *SOMClustering.jar*: Este ejecutable permite gestionar el proceso de clustering de manera simple, y se debe ejecutar posterior al ejecutable *AnimConvLauncher.jar*. Este ejecutable realiza, de manera interna, la llamada a los archivos *Clustering.exe*, *ClusterContentNCLauncher*, *ClusterContentLauncher.jar* y *ClusterFrameLauncher.jar*.

Capítulo 7

Pruebas y Resultados

En este capitulo se presentan las pruebas realizadas con algunos videos. Para cada video, las pruebas consisten básicamente en lo siguiente:

- Como primer paso, es necesario que se obtenga la secuencia de frames para cada video.
- Se obtiene la representación de la secuencia de frames en el EVM, en donde se almacenan solo los Couplets en forma de archivos binarios, tal como se describe en la sección 5.1.
- De manera general, la configuración seleccionada para las longitudes de la máscara son para una malla de tamaño 3x3 en el plano 2D (x, y) y 3 frames como longitud en el eje coordenado del tiempo. Cabe mencionar que los componentes de color van en función del esquema de color que se este utilizando, para estas pruebas se procesaron videos en escala de grises y RGB.
- Se ejecuta el algoritmo de convolución, con lo que se extrae el conjunto de valores de DC para cada posible sub-animación, véase el sección 5.2.3.
- Para la etapa de agrupamiento mediante un SOM y el modelo de Kohonen (capítulo 4) se consideraron tres configuraciones de neuronas para cada video: 5 neuronas, 10 neuronas y 15 neuronas. Se seleccionaron estos casos debido a que con ello se obtiene un panorama general de cómo el SOM realiza el agrupamiento para cada video, y también se observan patrones similares en cada configuración.
- Con base en lo anterior, se toma el conjunto de datos $DCValues$ para el proceso de entrenamiento del SOM, el cual consiste en 200 presentaciones del conjunto de datos de entrenamiento, y con ello se obtienen los vectores de pesos sinápticos, con los cuales se realiza el agrupamiento de los valores de DC.
- Después se forman los 3D-EVM resultantes correspondiente a cada neurona del SOM. Estos 3D-EVMs contienen las regiones correspondientes a cada cluster, las cuales presentan patrones similares con respecto al tiempo y al plano 2D de los frames, por ello solo se consideran los ejes coordenados (x, y, t).

  Con estos resultados se muestra que efectivamente se logra una compactación de video mediante el framework de segmentación desarrollado en este trabajo.
- Por último, para el 3D-EVM de cada cluster formado mediante el SOM, se obtienen su secuencia de frames correspondiente, en donde para efectos de visualizar el contenido de éstos, se obtiene la información de color correspondiente a las regiones de la animación original. No obstante también se presentan los 3D-EVMs para analizar su contenido.
- Es importante notar que, los resultados de agrupamiento que se presentan y son analizados, son aquellos que desde nuestro punto de vista son relevantes con respecto al contenido

del video. Los clusters que forman no cuentan con un etiquetado (véase la sección 1.7), ya que esta etapa se encuentra fuera del alcance de este proyecto. Por lo tanto solo se da una descripción de cada grupo formado en cada prueba.

7.1 Pruebas con un Video de Vigilancia Aérea

En esta prueba se realizó el procesamiento de un video de vigilancia aérea, el cual toma lugar en una reserva ecológica [50]. En la figura 7.1 se muestran algunos frames de dicho video, en donde se observa que se encuentra una *jirafa* en el centro de la escena, posteriormente la cámara se desplaza rodeándola. Con esto se tiene el caso en el que prácticamente toda la escena presenta cambios con respecto a los frames que se generan, ya que el movimiento de la cámara es continuo. Para estas pruebas, la animación en el 6D-EVM que representa la secuencia de frames tiene un total de 42430134 vértices extremos, considerando que los frames tienen un esquema de color RGB.

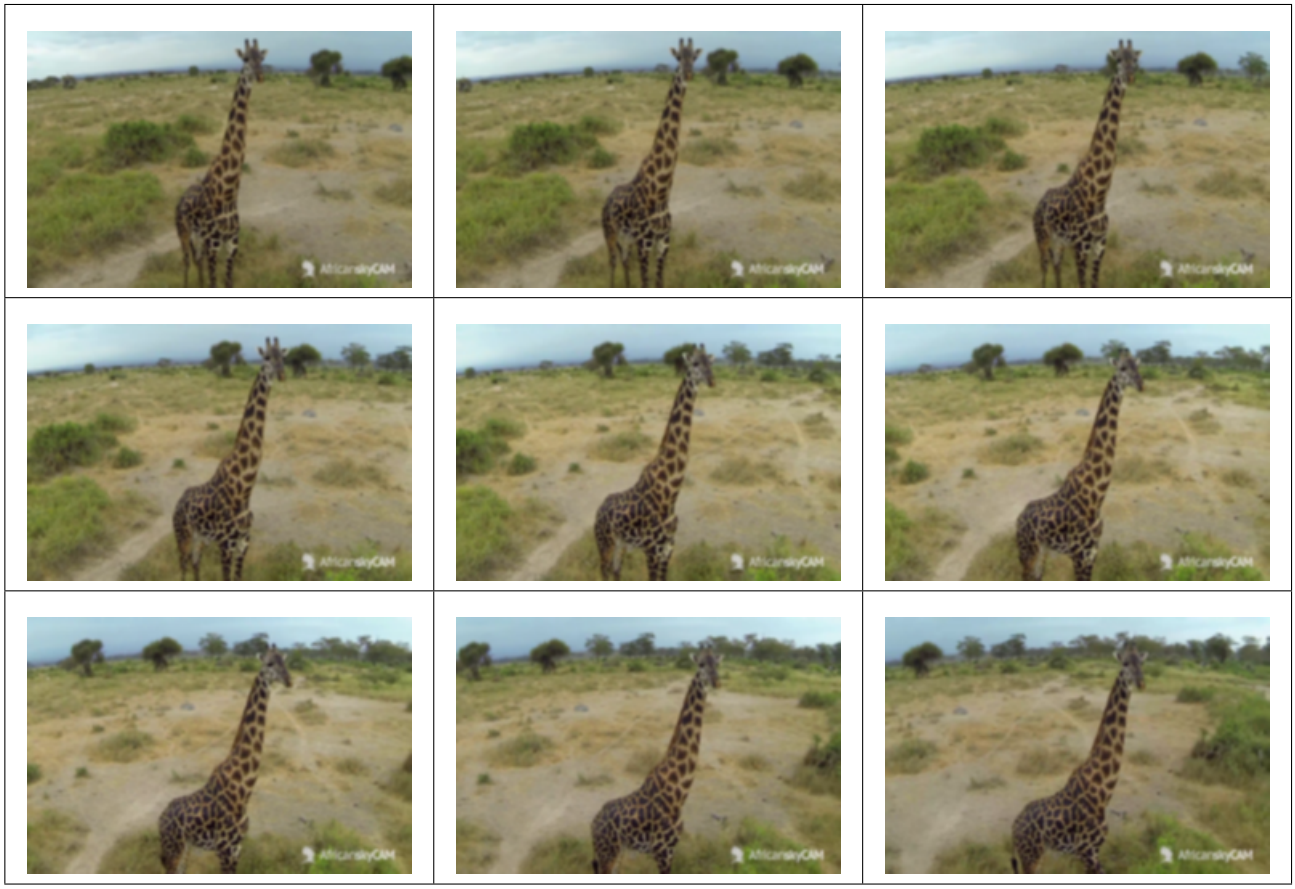


Figura 7.1: Algunos frames del video de vigilancia aérea.

7.1.1. Pruebas con 5 neuronas

En este apartado se describe el resultado de agrupamiento para 5 neuronas en el SOM. En la tabla 7.1 se muestra el tamaño de los 3D-EVMs generados por cada cluster, en donde se observa que el total de vértices extremos de los 3D-EVMs generados mediante el SOM y el descriptor de DC de las sub-animaciones es de 59258. La cantidad total de vértices extremos es mucho menor en comparación con los vértices extremos de la animación original, con ello se logra una tasa de compresión de 716.02. Es importante notar que los 3D-EVMs generados por el SOM no contienen información de color, ya que las sub-animaciones son agrupadas con base en la neurona ganadora.

Tabla 7.1: Tamaño de los 3D-EVMs generados por cada cluster para el video de vigilancia aérea y 5 neuronas.

Cluster	Tamaño del 3D-EVM
0	29032
1	18262
2	1408
3	3768
4	6788
Total	59258

Con este resultado se obtiene la compactación de video en el proceso de segmentación, ya que a las sub-animaciones se asigna como clase el número del cluster correspondiente. En la figura 7.2 se muestra la gráfica con la cantidad de sub-animaciones que contiene cada cluster, de esta manera se obtiene un panorama de la cantidad de información agrupada en ellos. Como se observa, los clusters que contienen más información son los clusters 0 y 1, y por ende posteriormente se analiza su contenido.

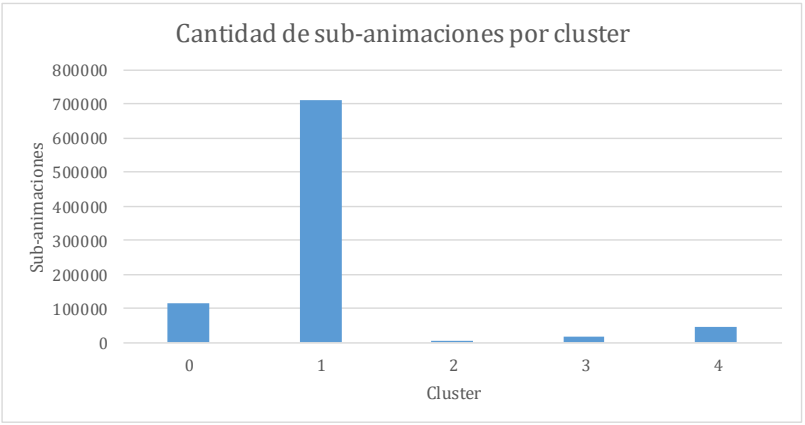


Figura 7.2: Gráfica con la cantidad de sub-animaciones para cada cluster para el video de vigilancia aérea y 5 neuronas.

Con base en lo anterior, se analiza el contenido de los clusters 0 y 1, los cuales se describen a continuación:

- En la figura 7.3 se muestra la secuencia de frames obtenida mediante la exploración de las sub-animaciones agrupadas en este cluster, no obstante para visualizar éstos frames, para cada sub-animación se obtiene la información de color respecto de la animación original. Como se observa en la figura 7.3, este cluster contiene la información de regiones en el horizonte con color azul, y también contiene información de regiones de color *uniforme* (o que varía poco), por ejemplo la parte del suelo que contiene pasto seco.

El correspondiente 3D-EVM que se obtiene mediante el SOM contiene regiones sin información de color, se muestra en la figura 7.4. Para comprender el objeto 3D que se genera, se debe considerar que los ejes coordenados correspondientes al plano 2D de los frames es paralelo con respecto al plano 2D al momento de leer este documento, y la tercera dimensión de profundidad corresponde al eje del tiempo y es perpendicular a dicho plano 2D.

Como se observa en la figura 7.4 se presentan desplazamientos en las regiones 2D de los frames correspondientes al cluster. En la parte superior del 3D-EVM se alcanza a observar la parte del horizonte (color azul en la figura 7.3), y en la parte central se aprecia la silueta de la jirafa.

- Por otro lado, para el cluster 1 se tiene como resultado la secuencia de frames que se muestran en la figura 7.5, en donde se observa que prácticamente se tienen las regiones de los frames que forman parte de la escena principal. Se observa que se agrupan regiones en donde se encuentra la jirafa, la parte del suelo y los arbustos que se encuentran en el fondo.

  El 3D-EVM que se genera a partir de las regiones del cluster pero sin información de color se muestra en la figura 7.6. En este 3D-EVM se observa que el politopo es muy similar a un cubo, esto con base en que las regiones de los frames que se están agrupando se mantienen casi constantes a lo largo del tiempo. No obstante se pueden apreciar algunos espacios vacíos en la parte derecha tal como se muestra en los frames.

Como se puede observar en los resultados, el cluster 1 contiene una gran cantidad de información, ello debido a que se tiene una cantidad pequeña de neuronas en el SOM, lo cual implica que no haya una separación adecuada de las sub-animaciones. No obstante, es posible realizar un post-procesamiento de este cluster, y considerarlo como una nueva animación para ser procesada nuevamente mediante un SOM, de esta manera se puede realizar una segmentación para clusters con una gran cantidad de sub-animaciones.

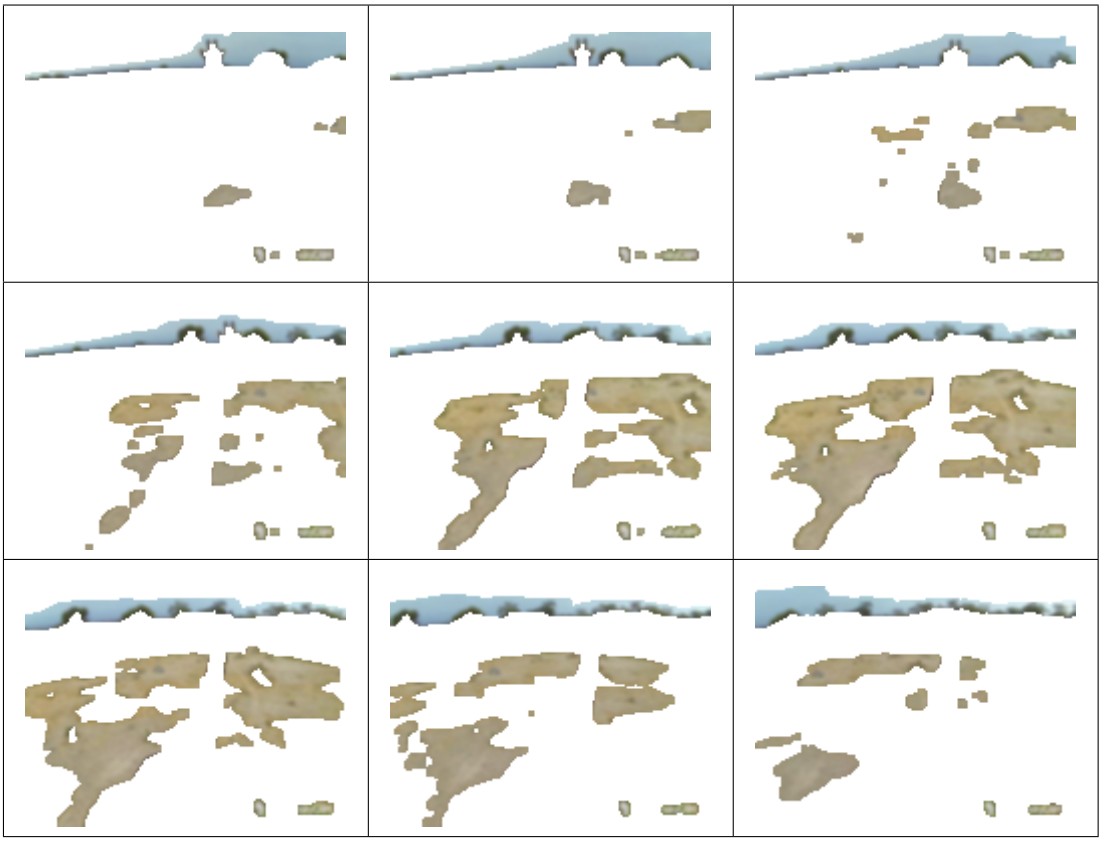


Figura 7.3: Video de vigilancia aérea y 5 neuronas, resultados de agrupamiento del cluster 0.

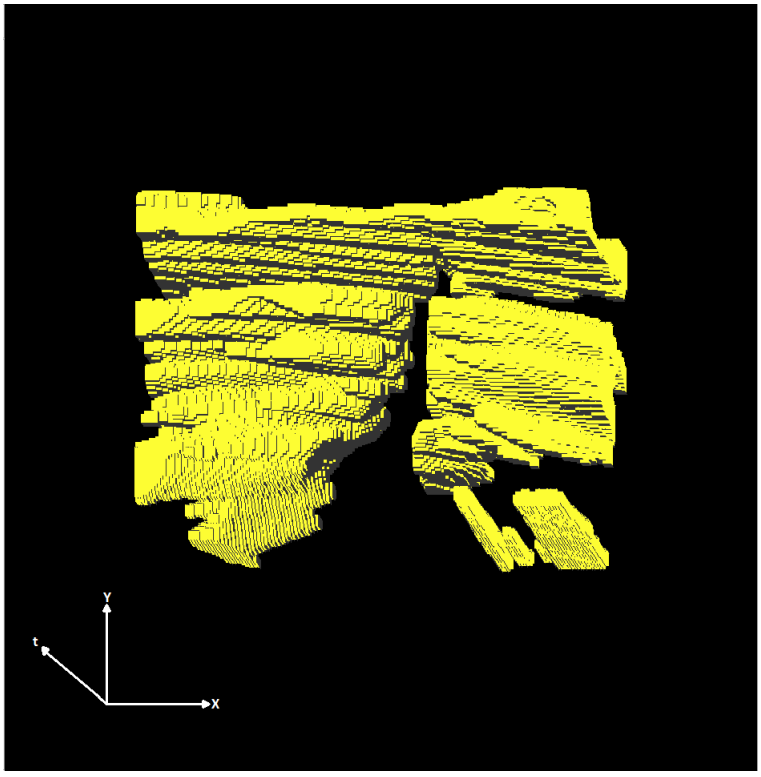


Figura 7.4: 3D-EVM del cluster 0 para el video de vigilancia aérea y 5 neuronas.

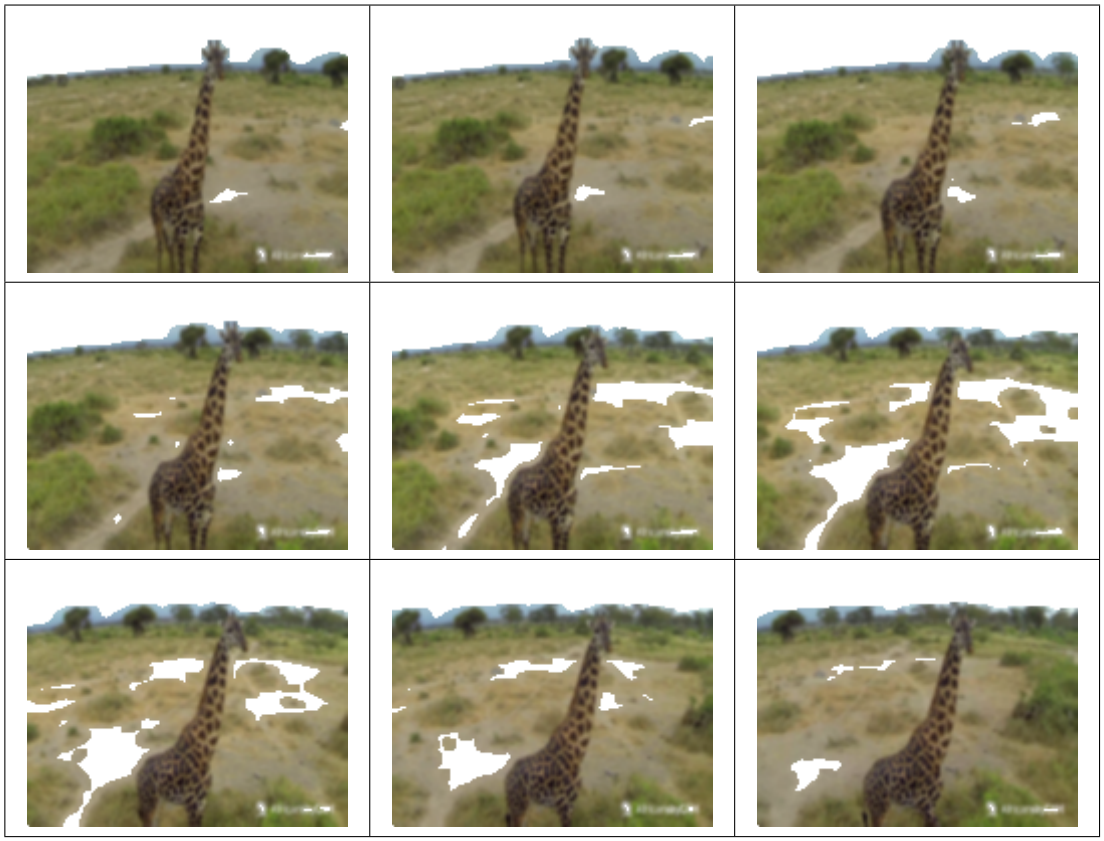


Figura 7.5: Video de vigilancia aérea y 5 neuronas, resultados del cluster 1.

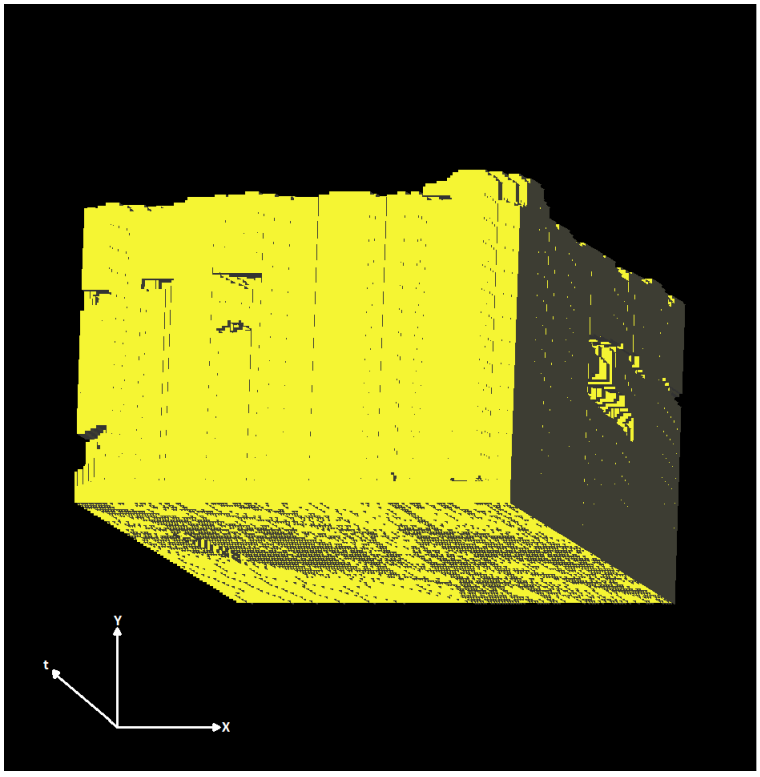


Figura 7.6: 3D-EVM del cluster 1 para el video de vigilancia aérea y 5 neuronas.

7.1.2. Pruebas con 10 neuronas

En este apartado se presentan los resultados de agrupamiento para la secuencia de frames de la figura 7.1 usando 10 neuronas en el SOM. En la tabla 7.2 se muestran los tamaños de los 3D-EVMs generados por cada cluster. Nuevamente es importante notar que se obtiene una reducción significativa en la cantidad de vértices extremos, ya que en total se tienen 185528 vértices extremos con este resultado, con lo que logra la compactación de la animación original, y la tasa de compresión es de 228.69.

Tabla 7.2: Tamaño de los 3D-EVMs generados por cada cluster para el video de vigilancia aérea y 10 neuronas.

Cluster	Tamaño del 3D-EVM
0	1064
1	16400
2	5144
3	33112
4	43616
5	7862
6	65698
7	6142
8	2520
9	3970
Total	185528

En la figura 7.7 se muestra la gráfica de la cantidad de sub-animaciones para cada cluster. Como se observa, se tienen clusters con una densidad mayor con respecto a las regiones que contienen, en este caso, los clusters 3, 4 y 6 contienen la mayor parte de las regiones de la animación original, mientras que el resto contiene una cantidad mucho menor de éstas. Los grupos con menor cantidad de sub-animaciones basan su contenido en que dentro de los frames se pueden hallar pequeñas regiones muy aisladas o que contienen información de color diferente al resto de la escena.

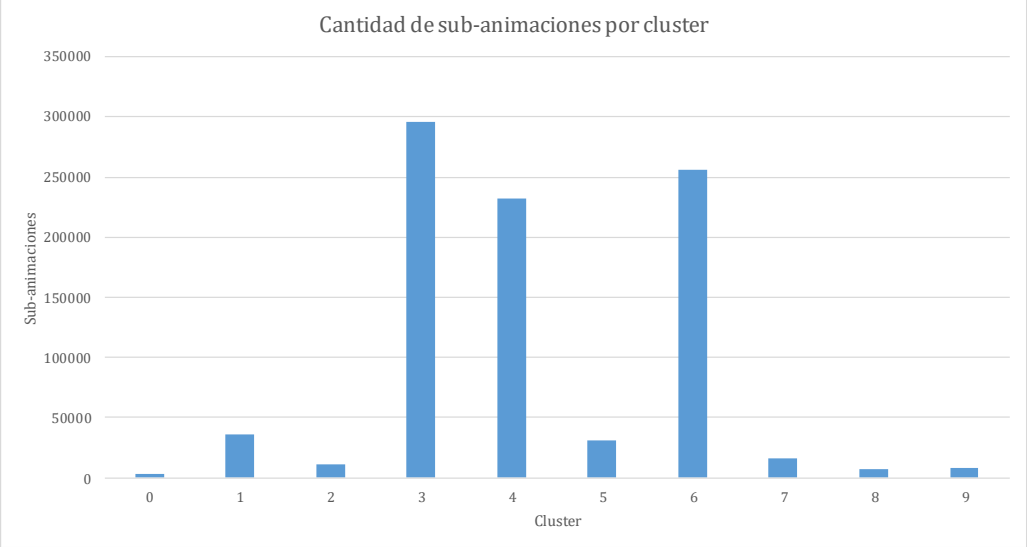


Figura 7.7: Gráficas con la cantidad de sub-animaciones por cada cluster para el video de vigilancia aérea y 10 neuronas.

Con base en lo anterior y considerando la gráfica que se ilustra en la figura 7.7, se analiza el contenido de los clusters 3, 4 y 6, los cuales se describen a continuación:

- En la figura 7.8 se muestra la secuencia de frames para el cluster 3, esto se obtiene al obtener la información original del color de las sub-animaciones en la animación original. Como se observa, este cluster contiene regiones en donde se encuentra la jirafa y regiones en donde hay arbustos. Esto tiene sentido debido a que la jirafa cuenta con manchas obscuras, y los arbustos también pueden ser vistos como pequeñas manchas de color más obscuro al del pasto del suelo.

  Ahora, en la figura 7.9 se muestra el 3D-EVM correspondiente a las regiones contenidas en el cluster 3 (figura 7.8). Se alcanza a apreciar que las regiones de la jirafa y el resto de regiones se encuentran agrupadas de manera adecuada, ya que se observa como una región sólida debido a que éstas son conservadas en el cluster aún cuando hay movimiento de la cámara.

- En la figura 7.10, se presenta la secuencia de frames obtenida del agrupamiento para el cluster 4 y la información de color de la animación original. Como se observa, las regiones que se encuentran en este cluster son aquellas que contienen información de color uniforme o que no tiene cambios bruscos en el color. En este caso, la mayoría de regiones en el cluster son aquellas en donde se encuentra pasto seco o de color pálido. También se puede observar que la jirafa prácticamente ha sido descartada de la escena y solo se tiene su silueta.

  Con base en los frames del cluster 4 de la figura 7.10, se forma el 3D-EVM que contiene las regiones del cluster pero sin información de color. Este 3D-EVM se muestra en la figura 7.11, en donde se observa la silueta de la jirafa, tal como se obtienen en los frames, y en la parte superior se tienen las regiones con información del horizonte de la escena.

- Por último, se analizan los resultados del cluster 6, y en la figura 7.12 se muestra la secuencia de frames, en donde se observa que se agrupan regiones en donde hay un cambio en el color. Por ejemplo, para la silueta de la jirafa se pueden apreciar sus bordes, ya que de manera general en esas zonas (donde hay bordes) hay cambios bruscos de color. No obstante se tienen regiones donde hay una tendencia hacia un cambio de color, tal como las zonas donde hay un cambio entre el pasto seco al pasto verde.

  En la figura 7.13, se observa el 3D-EVM correspondiente al cluster 6, el cual contiene las regiones que se muestran en los frames de la figura 7.12 pero sin información de color.

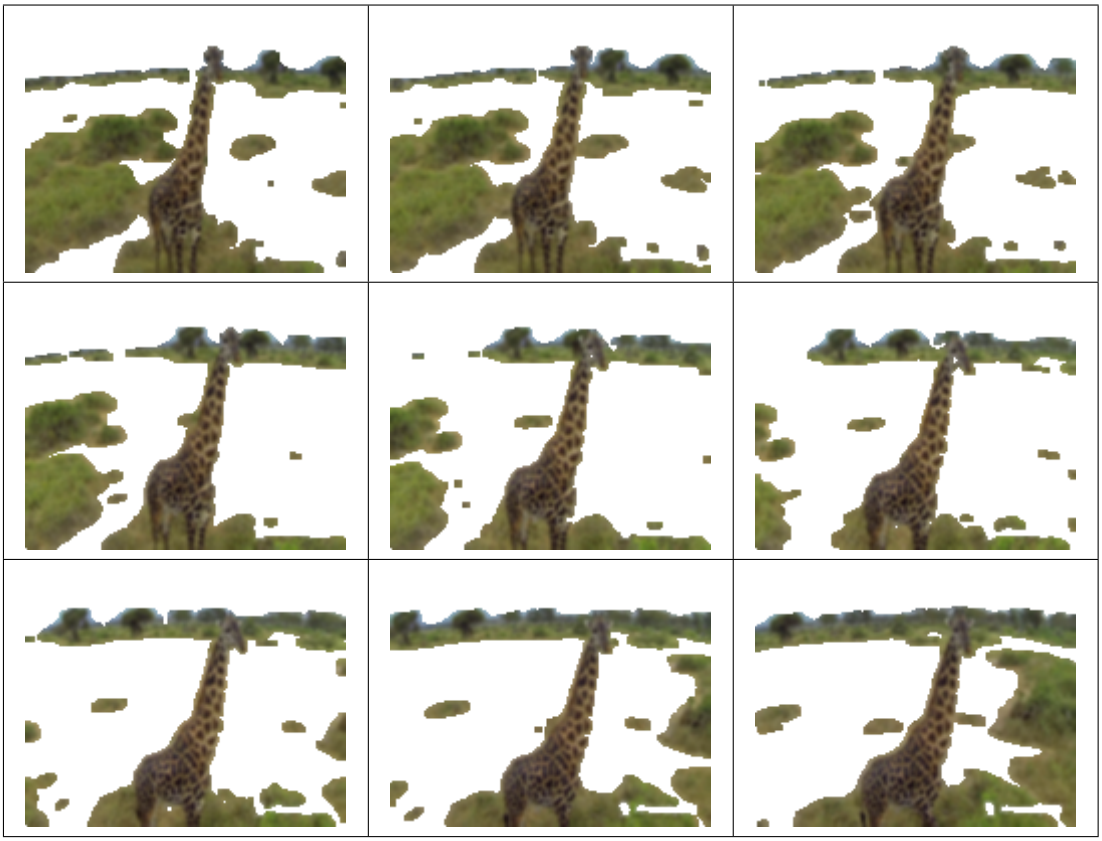


Figura 7.8: Video de vigilancia aérea y 10 neuronas, resultados de agrupamiento del cluster 3.

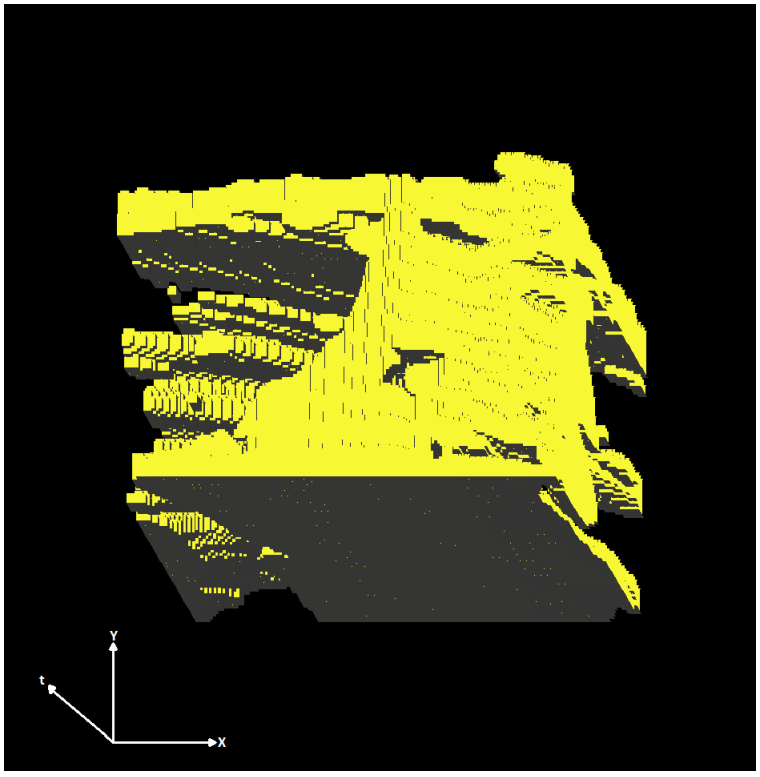


Figura 7.9: 3D-EVM del cluster 3 para el video de vigilancia aérea y 10 neuronas.

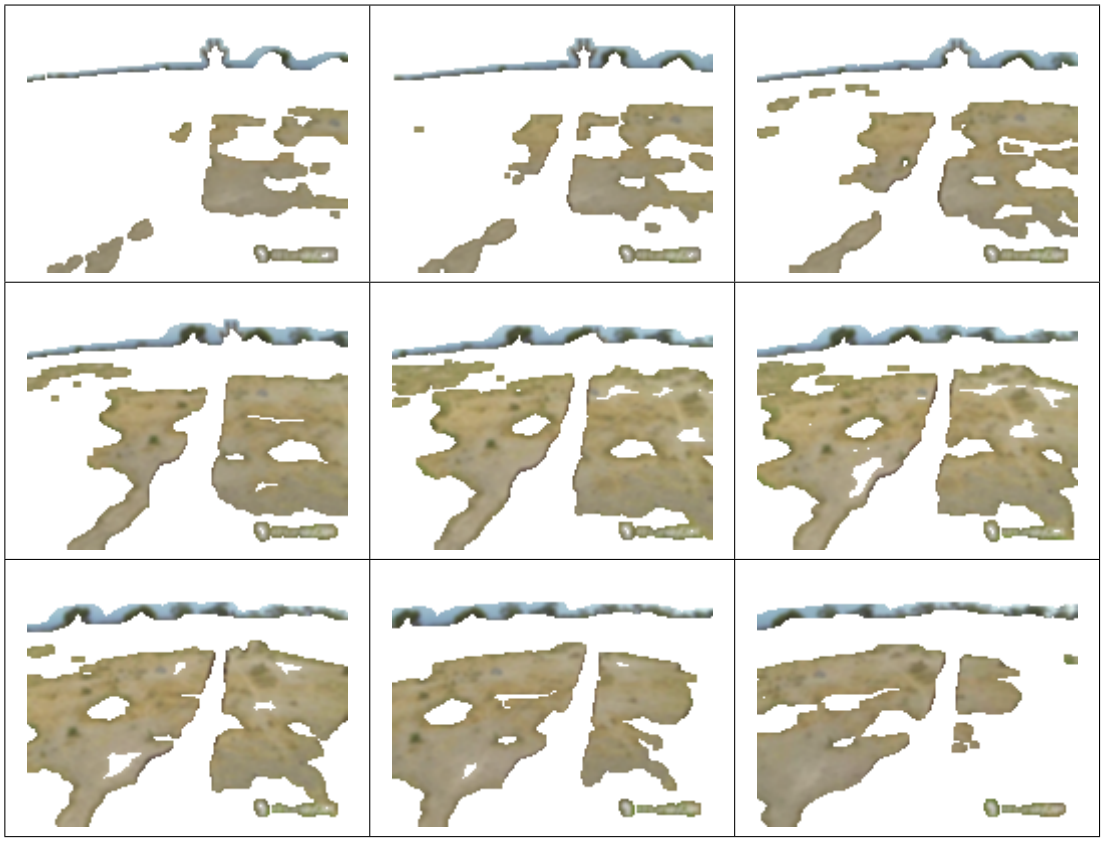


Figura 7.10: Video de vigilancia aérea y 10 neuronas, resultados del cluster 4.

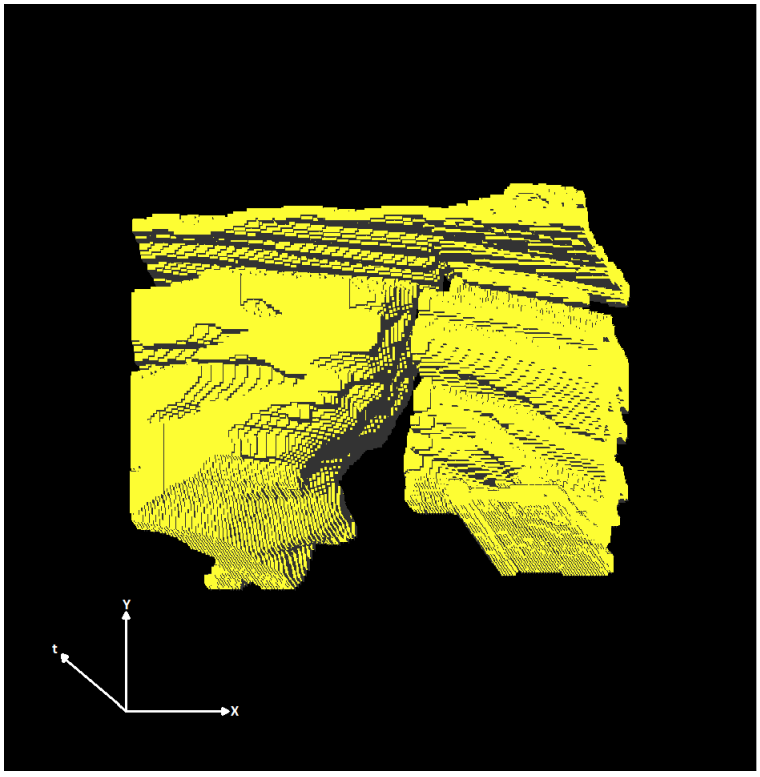


Figura 7.11: 3D-EVM del cluster 4 para el video de vigilancia aérea y 10 neuronas.

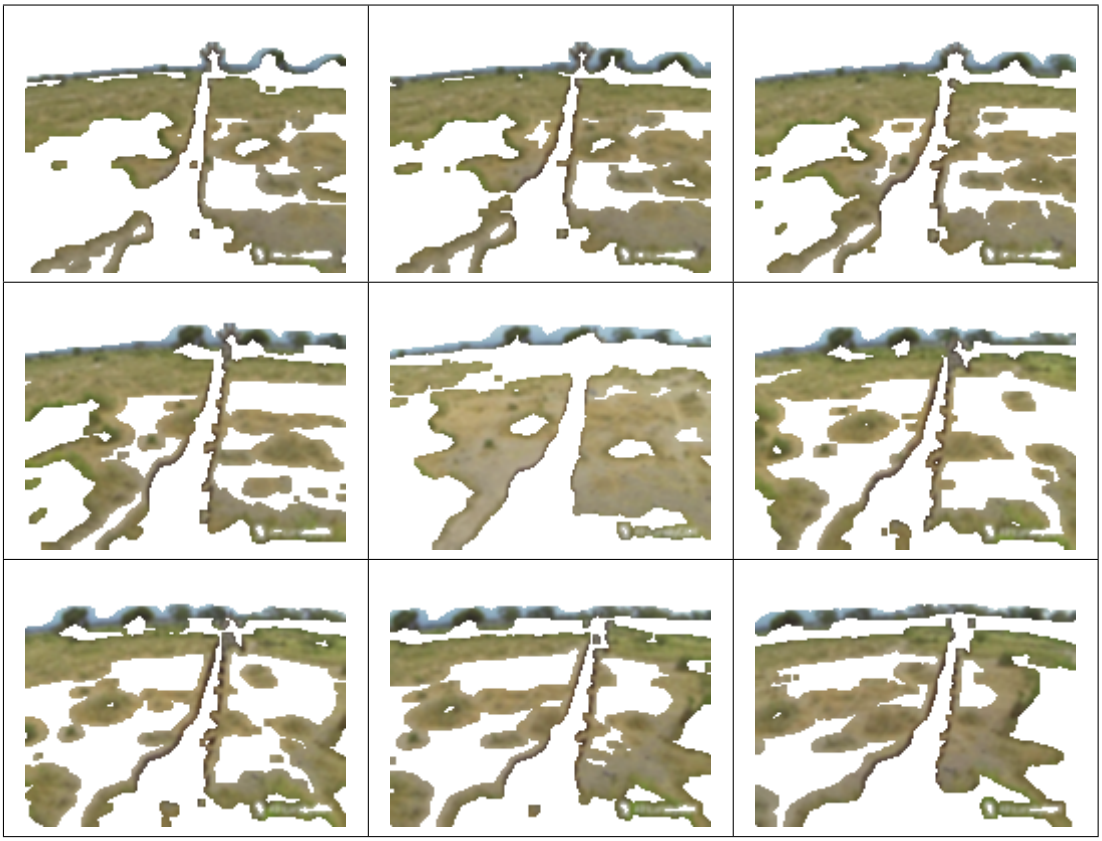


Figura 7.12: Video de vigilancia aérea y 10 neuronas, resultados del cluster 6.

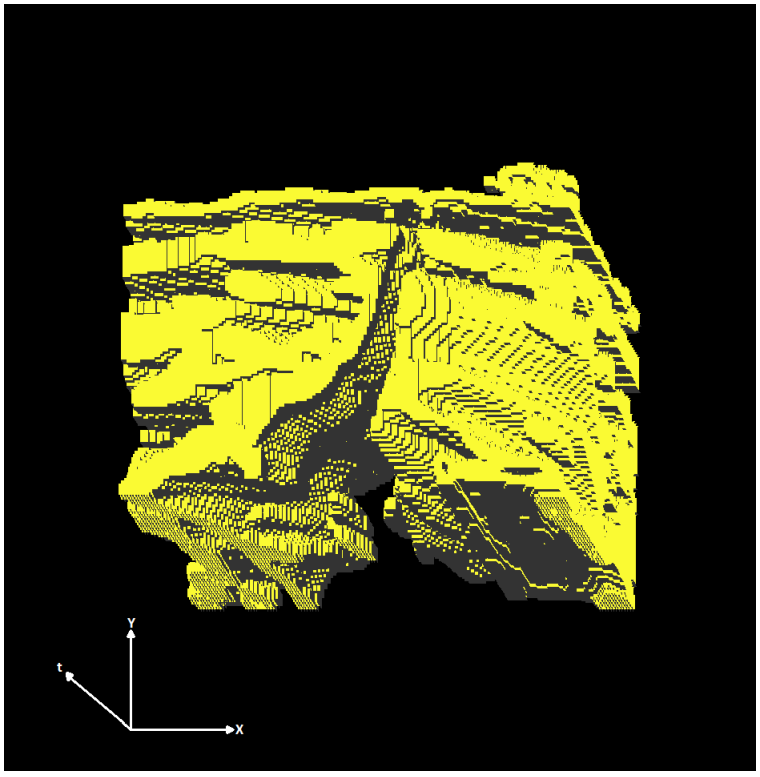


Figura 7.13: 3D-EVM del cluster 6 para el video de vigilancia aérea y 10 neuronas.

7.1.3. Pruebas con 15 neuronas

En este apartado se presentan las pruebas de segmentación con 15 neuronas para el SOM. En la tabla 7.3 se muestra la cantidad de vértices extremos para los 3D-EVMs correspondientes a cada cluster, en donde se observa que se tiene un total de 181376 vértices extremos, lo cual en comparación con la animación original se logra la compactación de ésta, ya que la etiqueta asignada a las regiones del cluster corresponde al número del cluster. La tasa de compresión para este caso es de 233.93.

Tabla 7.3: Tamaño de los 3D-EVMs generados por cada cluster para el video de vigilancia aérea y 15 neuronas.

Cluster	Tamaño del 3D-EVM
0	66430
1	5308
2	28956
3	896
4	6216
5	4374
6	7104
7	3276
8	10816
9	48000
10	2066
11	8034
12	65926
13	48920
14	24802
Total	181376

En la gráfica de la figura 7.14 se presenta la cantidad de sub-animaciones agrupadas en los clusters. Como se observa, los clusters que contienen más información son el cluster 0, 2, 9, 12 y 13. Por ello se describen éstos para analizar y observar su contenido.

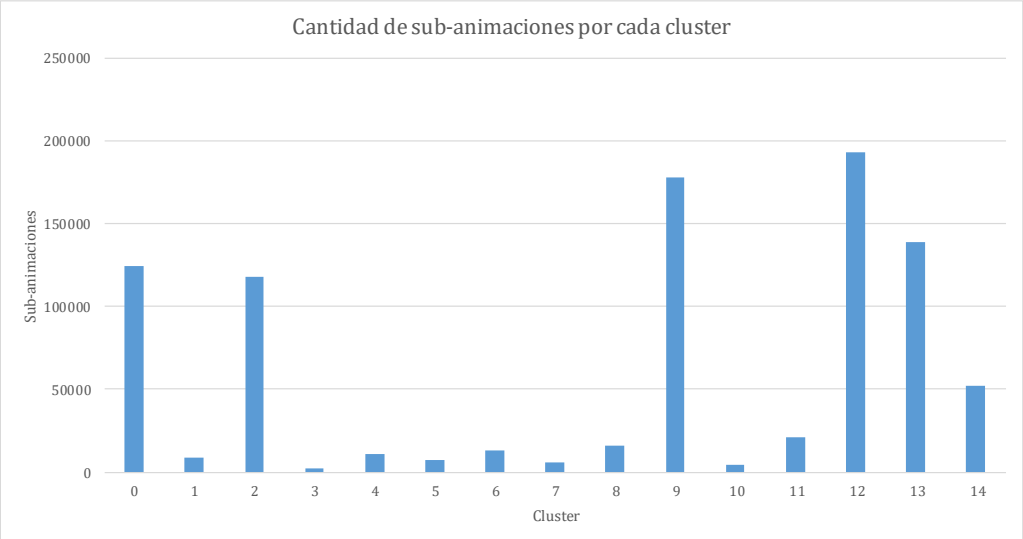


Figura 7.14: Gráfica con la cantidad de sub-animaciones para cada cluster para el video de vigilancia aérea y 15 neuronas.

Con base en lo anterior, a continuación se da la descripción de los clusters mencionados anteriormente:

- Para el cluster 0 se muestran sus frames en la figura 7.15, en donde se observa que las regiones agrupadas son aquellas en donde se presentan cambios de color. En este caso se puede observar el borde del cuerpo de la jirafa, lo cual tiene sentido porque en las regiones donde hay bordes se presentan cambios de color de manera brusca. También se pueden observar regiones en donde hay bordes de los arbustos, ya que en ellas hay un cambio de color entre los abustos y las zonas del pasto verde o seco.

  En la figura 7.15 se muestra el 3D-EVM generado a partir de las regiones del cluster 0, y considerando que no se tiene la información de color. Como se observa en la figura 7.16, para la silueta de la jirafa se tiene información sobre los bordes de ésta. No obstante, debido al movimiento de la cámara se observan que las regiones agrupadas también presentan movimiento en el 3D-EVM.

- En el cluster 2 se tienen los frames que se muestran en la figura 7.17. Como se observa, en este cluster prácticamente se agrupa solo las regiones del cuerpo de la jirafa, no obstante también se agrupan regiones con colores obscuros, tal como los arbustos. Otro patrón que se observa en este cluster es que con el movimiento de la cámara, las regiones de arbustos y del cuerpo de la jirafa se preservan aunque no en la misma posición espacial en el frame.

  En la figura 7.18 se muestra el 3D-EVM generado a partir de las regiones de los frames del cluster, pero sin considerar los componentes de color. Como se observa, se tiene el cuerpo de la jirafa a lo largo de todo el 3D-EVM y también se observa que las regiones que corresponden a los arbustos se desplazan a lo largo del eje del tiempo.

- Para el cluster 9 se muestran sus frames en la figura 7.19, en donde se observa que las regiones agrupadas corresponden a zonas con un color uniforme o que cambia poco. En este caso se puede notar que las regiones de los frames que se están agrupando, corresponden a zonas con pasto seco.

  Por otro lado, en la figura 7.20 se muestra el 3D-EVM para el cluster 9, en donde se observa la silueta de la jirafa, y las regiones correspondientes al pasto seco presentan movimiento con base en la rotación de la cámara.

- El cluster 12 cuyos frames se muestran en la figura 7.21, contiene información que presenta tendencias hacia bordes de objetos, ya que por ejemplo se agrupan regiones en el borde de la silueta de la jirafa. Por otro lado también se alcanza apreciar que hay regiones de los bordes entre el pastoy arbustos.

  En la figura 7.22 se muestra el 3D-EVM generado a partir de los frames, en donde se observa que la región de la jirafa está vacía, pero se preservan regiones de los bordes. También se puede apreciar que las regiones del pasto presenta movimiento con respecto al tiempo.

- Por último, el cluster 13 contiene los frames que se muestran en la figura 7.23, en donde se observa que se tienen regiones que pertenecen a zonas con pasto con color suave y color más oscuro. También se agrupan ciertas regiones del cuerpo de la jirafa, aunque no se tiene toda como tal. Este comportamiento es razonable en el sentido que ahora se utilizan más neuronas en el SOM, lo cual produce más regiones y que algunas contengan información similar. Para configuraciones con un número menor de neuronas, se podría esperar que este cluster forme parte de otro con mayor información (similar). En la figura 7.24 se muestra el 3D-EVM correspondiente a este cluster.

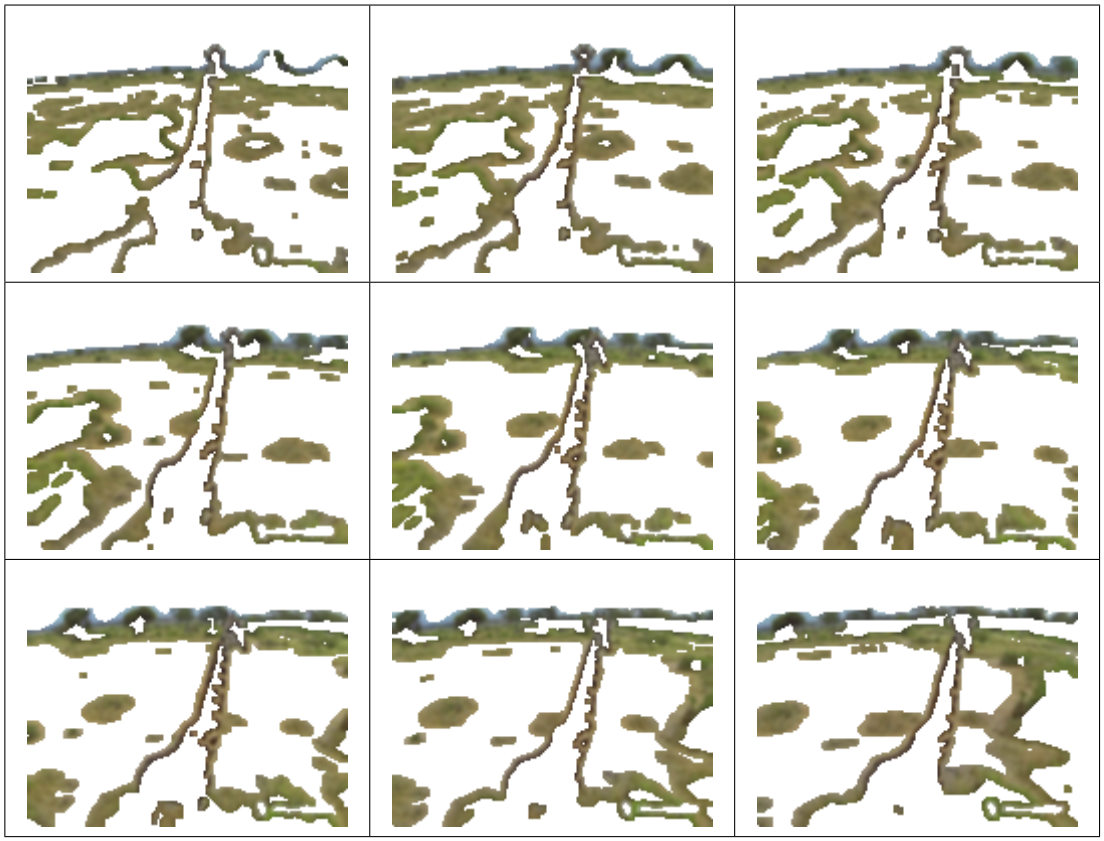


Figura 7.15: Video de vigilancia aérea y 15 neuronas, resultados de agrupamiento del cluster 0.

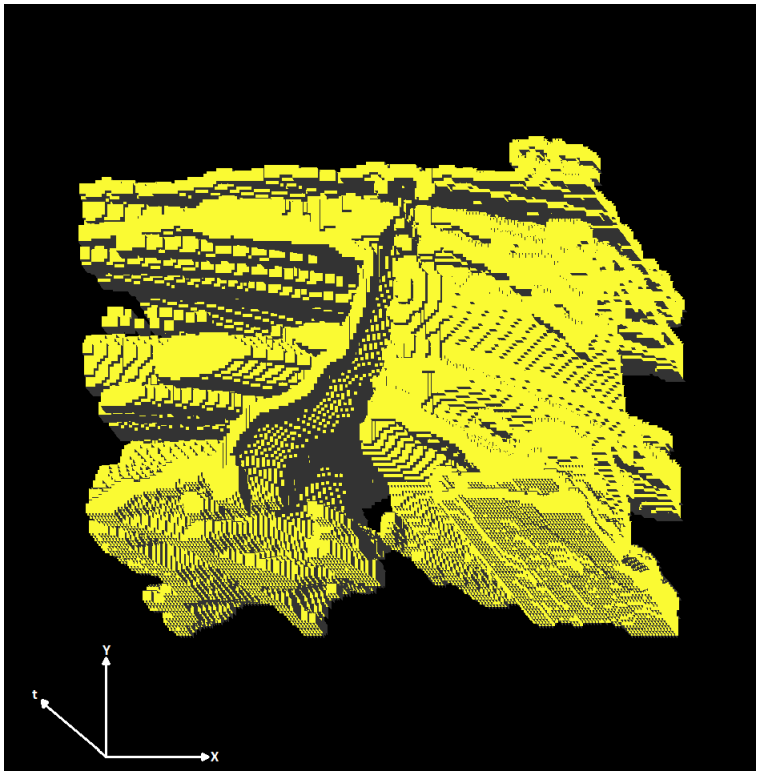


Figura 7.16: 3D-EVM del cluster 0 para el video de vigilancia aérea y 15 neuronas.

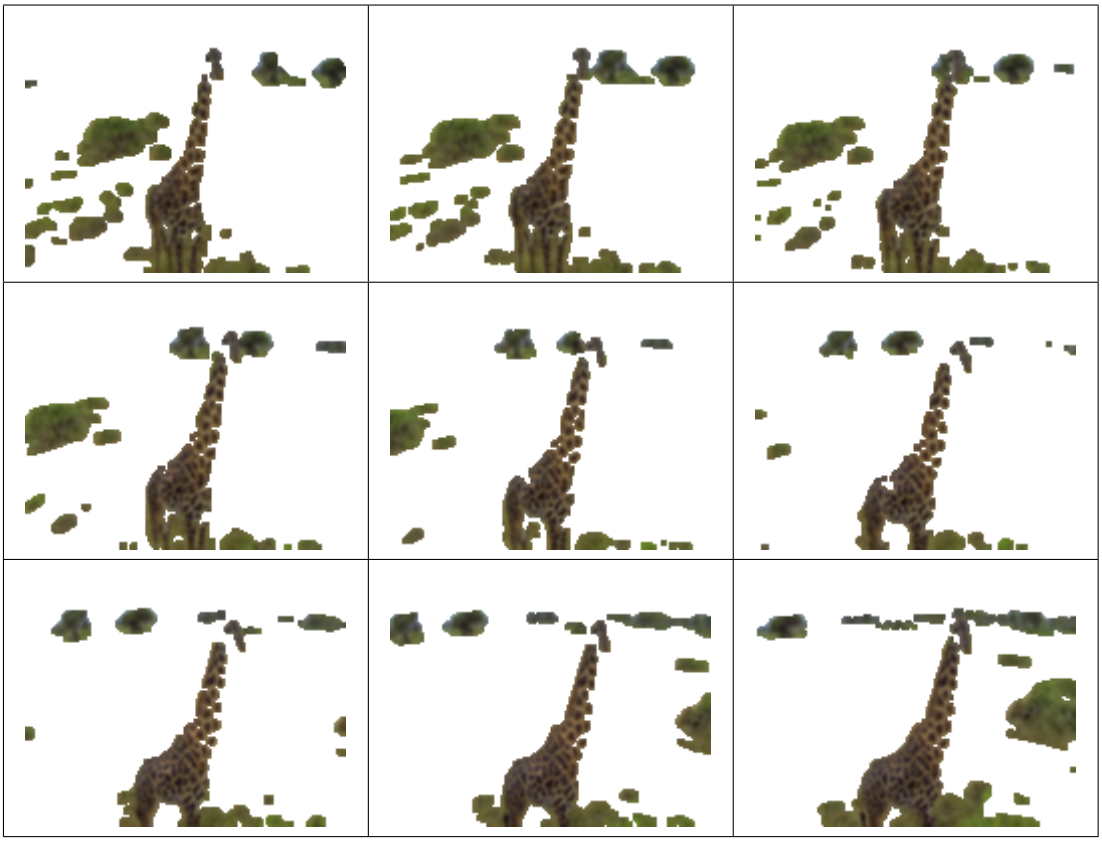


Figura 7.17: Video de vigilancia aérea y 15 neuronas, resultados de agrupamiento del cluster 2.

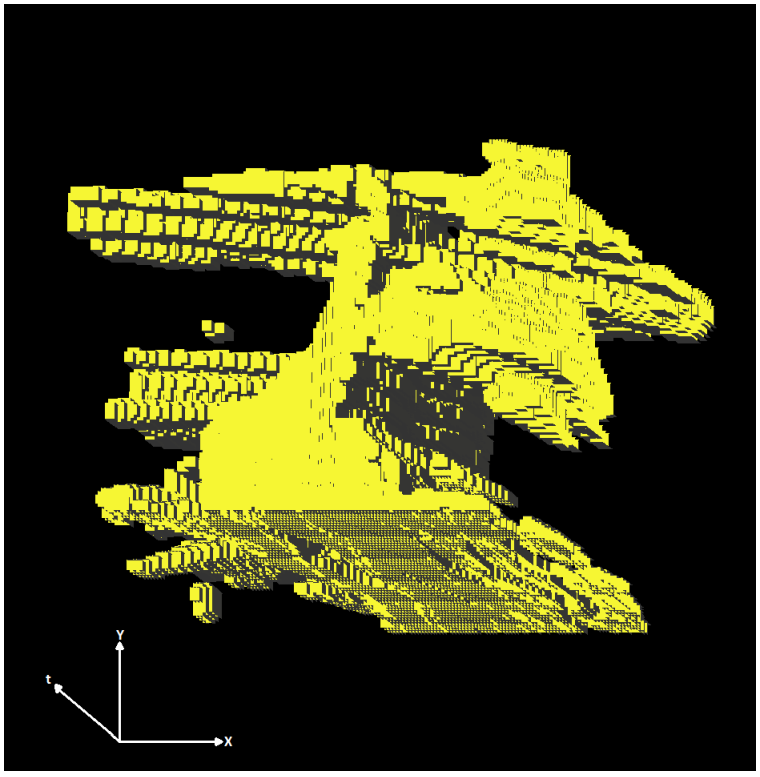


Figura 7.18: 3D-EVM del cluster 2 para el video de vigilancia aérea y 15 neuronas.

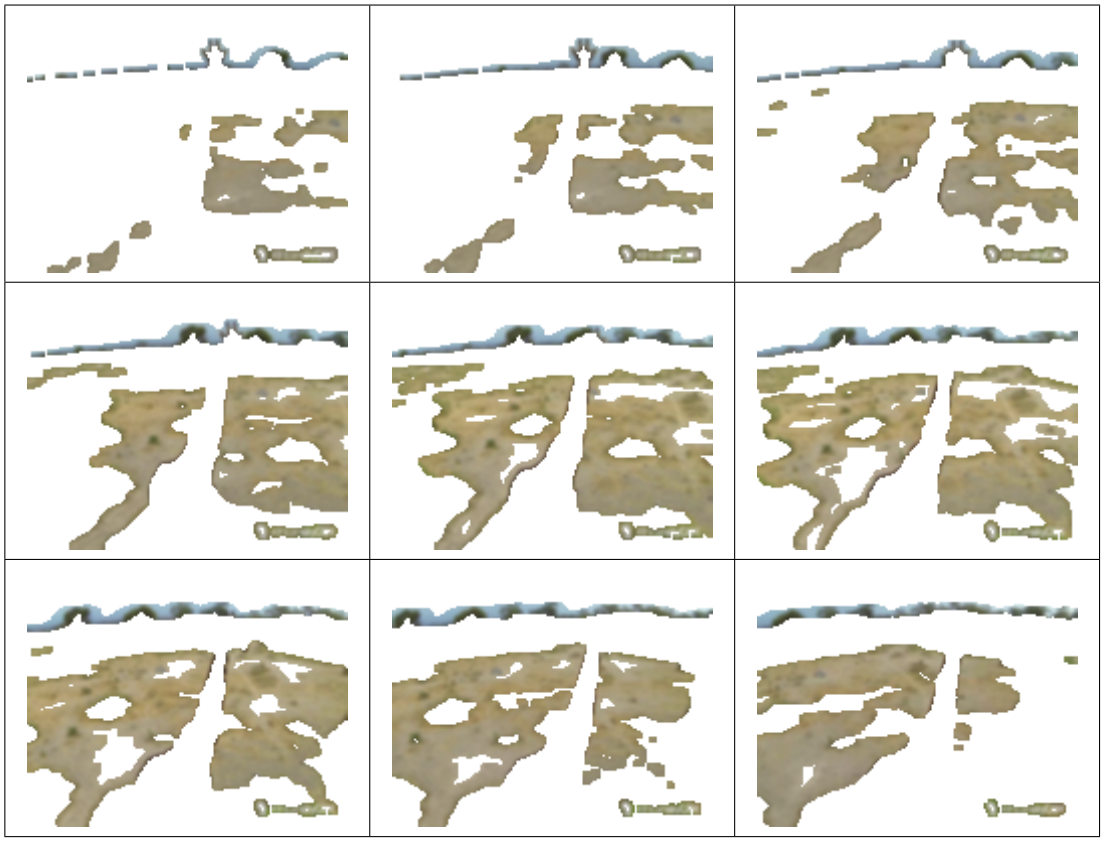


Figura 7.19: Video de vigilancia aérea y 15 neuronas, resultados de agrupamiento del cluster 9.

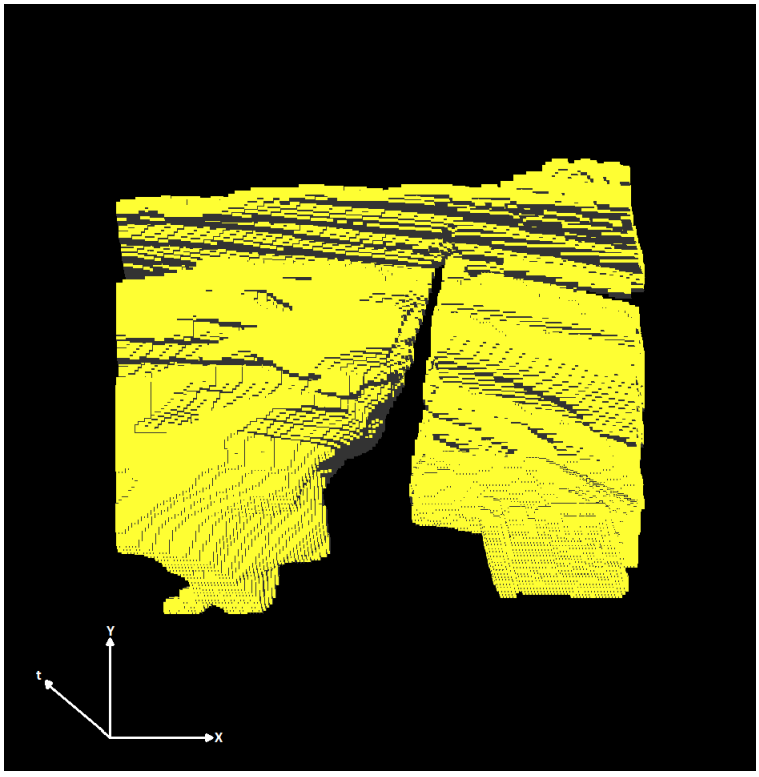


Figura 7.20: 3D-EVM del cluster 9 para el video de vigilancia aérea y 15 neuronas.

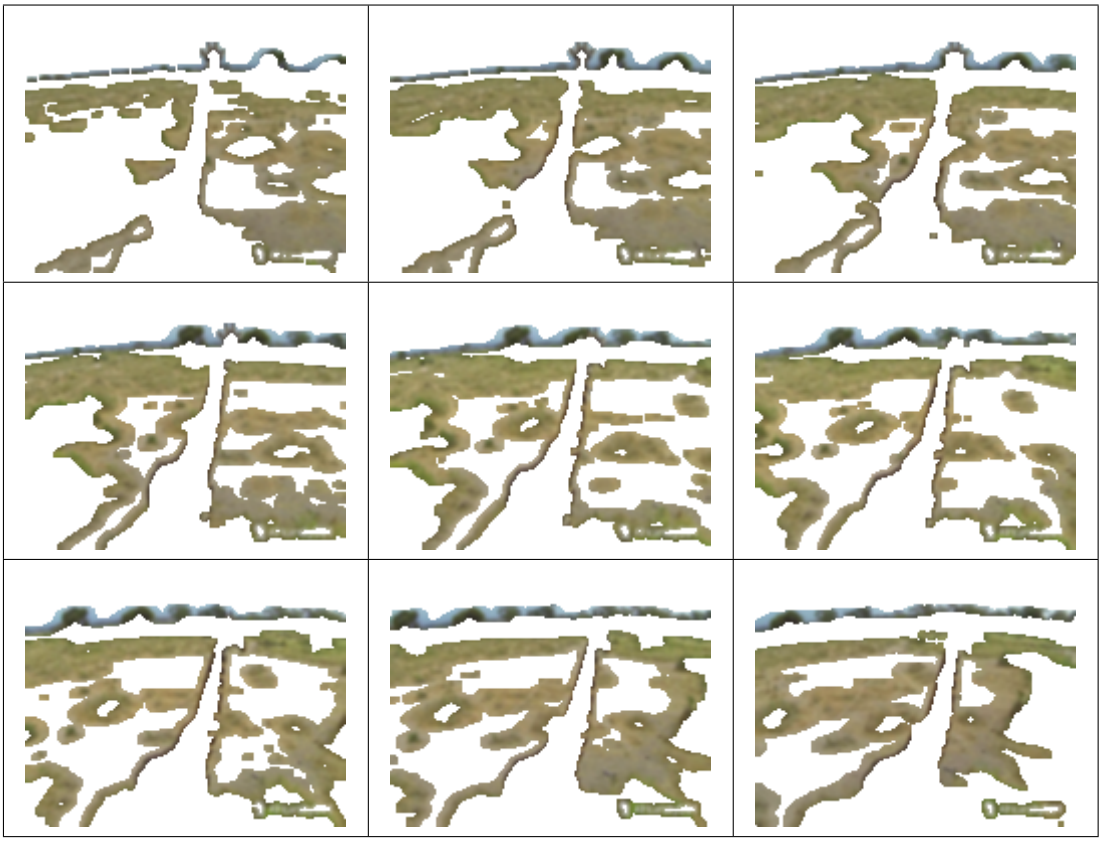


Figura 7.21: Video de vigilancia aérea y 15 neuronas, resultados de agrupamiento del cluster 12.

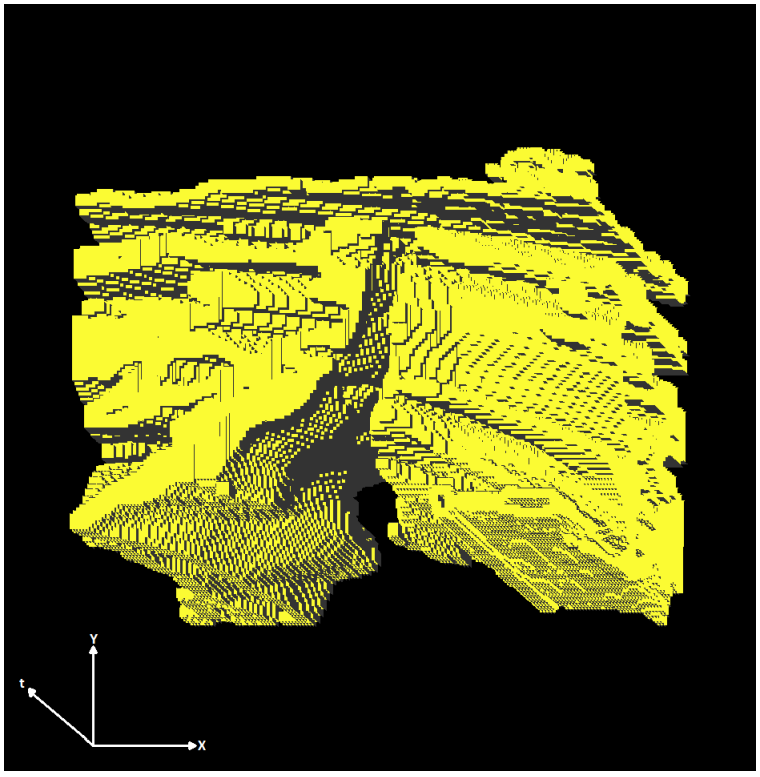


Figura 7.22: 3D-EVM del cluster 12 para el video de vigilancia aérea y 15 neuronas.

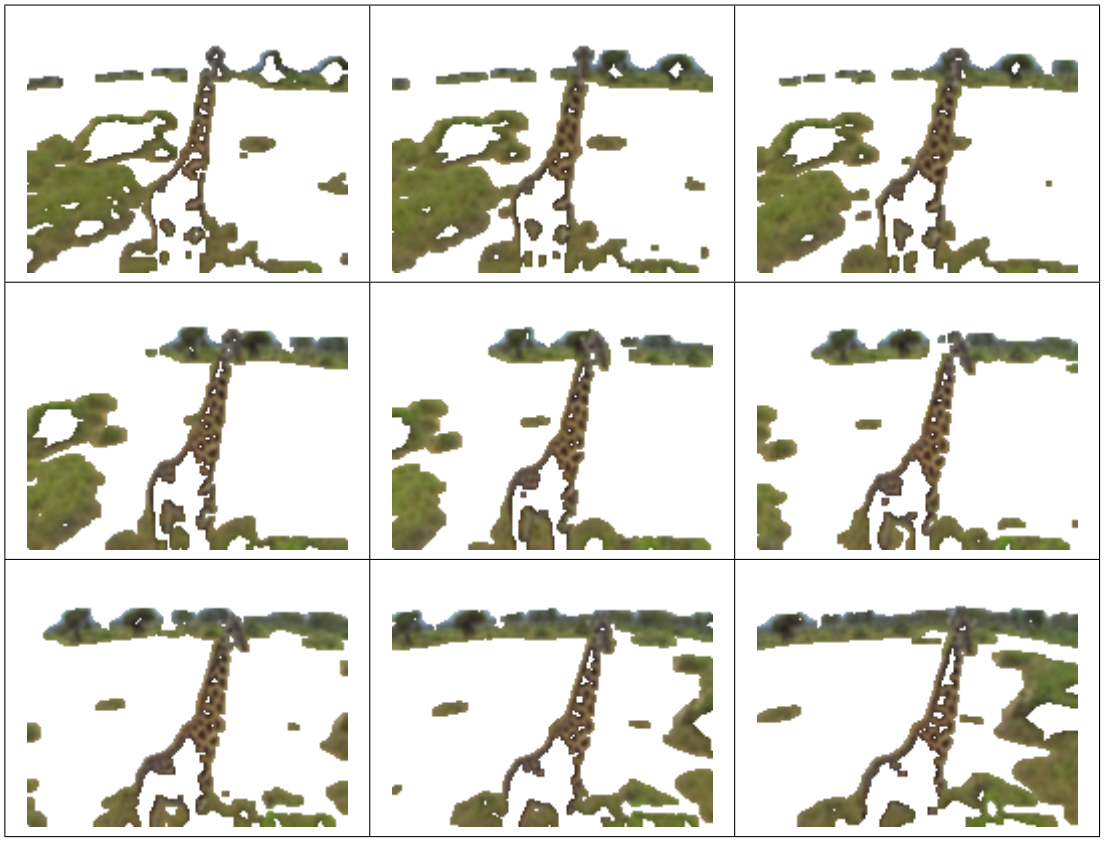


Figura 7.23: Video de vigilancia aérea y 15 neuronas, resultados de agrupamiento del cluster 13.

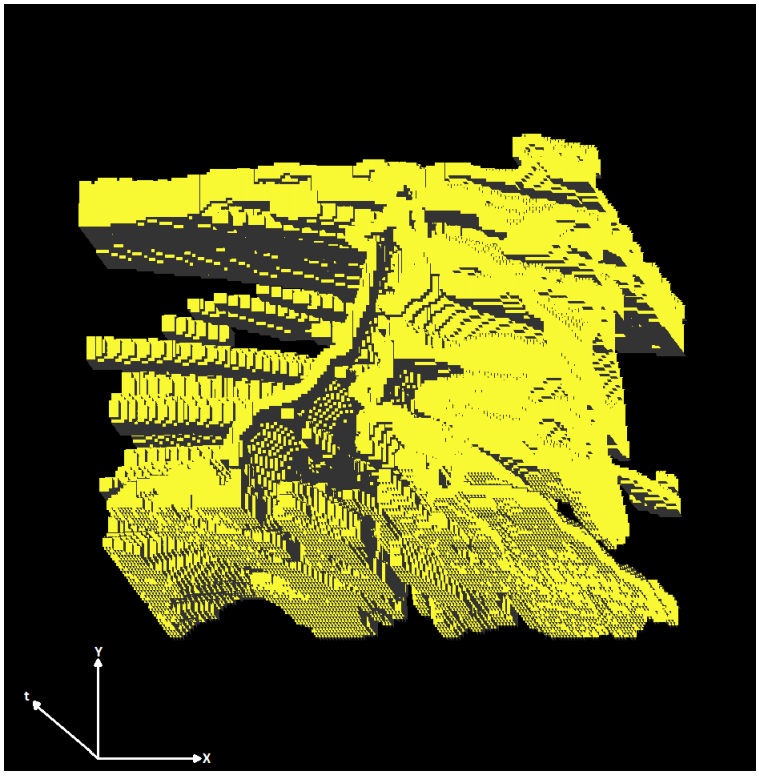


Figura 7.24: 3D-EVM del cluster 13 para el video de vigilancia aérea y 15 neuronas.

7.2 Pruebas con un Video de Control de Tráfico

En este apartado se analizan los resultados obtenidos de las pruebas con un video de control de tráfico [56], del cual se muestran algunos frames en la figura 7.25. Como se observa, el video fue tomado mediante una cámara estática y se captura el movimiento de los autos sobre la autopista. En este caso, debido a que se tiene una cámara estática los cambios entre frames consecutivos radican en el movimiento de los autos o por artefactos ocasionados por el entorno.

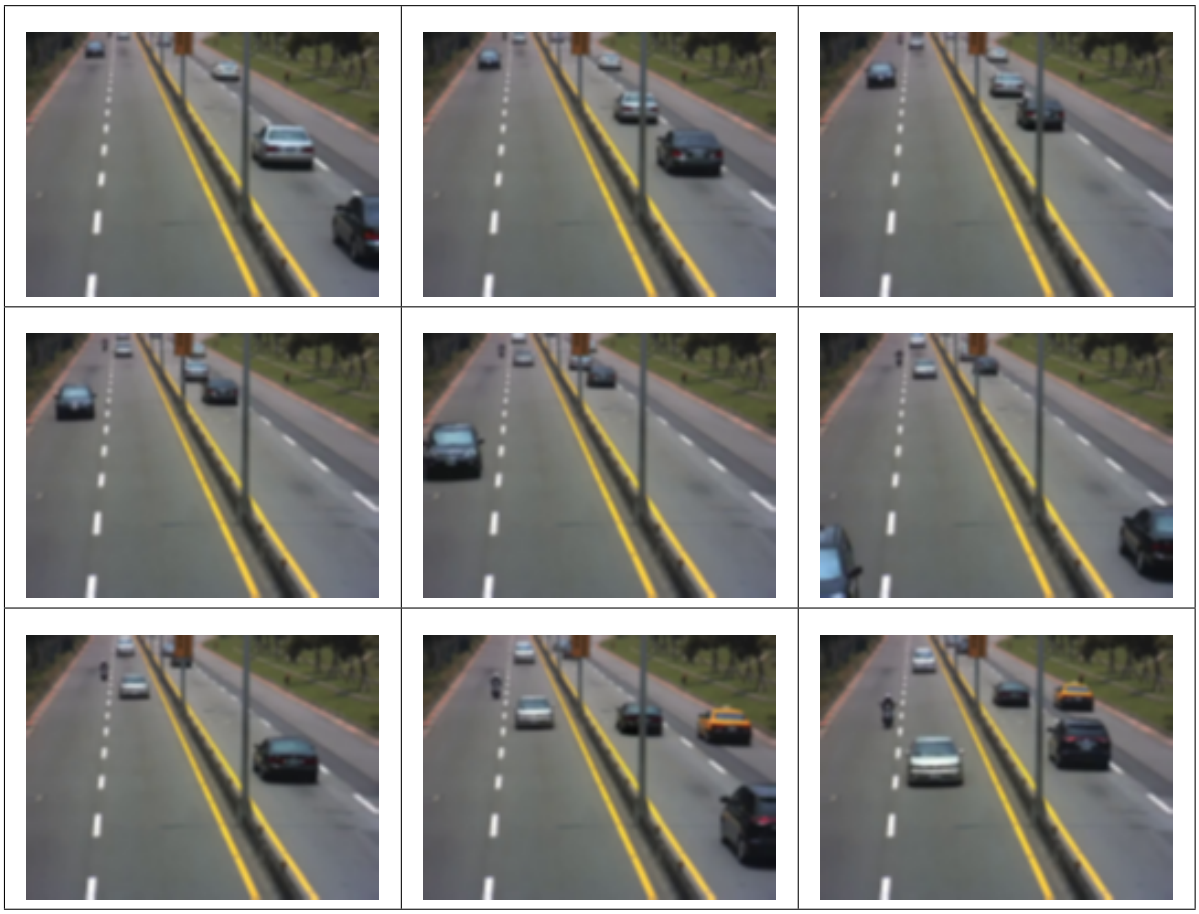


Figura 7.25: Algunos frames del video de control de tráfico.

De manera similar al caso de prueba descrito en la sección 7.1, los frames de este video tienen un tamaño de 160x120 y se tienen 60 frames consecutivos. En base al flujo del proceso de representación de una secuencia de frames, se obtuvieron los Couplets de la animación en el 6D-EVM, y en general la representación de la secuencia de frames contiene un total de 17058774 vértices extremos, considerando que se utiliza el esquema de color RGB. En los siguientes apartados se presentan los resultados de segmentación utilizando 5, 10 y 15 neuronas en el SOM.

7.2.1. Pruebas con 5 neuronas

En este apartado se analizan los resultados de agrupamiento mediante un SOM con 5 neuronas. Para los 3D-EVMs correspondientes a los clusters, en la tabla 7.4 se muestra la cantidad de vértices extremos para cada uno de ellos. Como se observa, se tiene un total de 80128 vértices extremos, lo cual en comparación con los vértices extremos del 6D-EVM en la animación original se obtiene una compresión de aproximadamente 212.89. Con lo anterior se valida que la compactación efectivamente se realiza gracias al agrupamiento del SOM y los valores de DC así como de la representación mediante el 3D-EVM para las sub-animaciones.

Tabla 7.4: Tamaño de los 3D-EVMs generados por cada cluster para el video de control de tráfico y 5 neuronas.

Cluster	Tamaño del 3D-EVM
0	28968
1	1528
2	24674
3	6308
4	18650
Total	80128

En la figura 7.26 se muestra la gráfica con la cantidad de sub-animaciones agrupadas en los clusters, en donde se observa que los clusters que contienen mayor información son el cluster 0, 2 y 4. Por lo anterior posteriormente se realiza el análisis del contenido de dichos clusters.

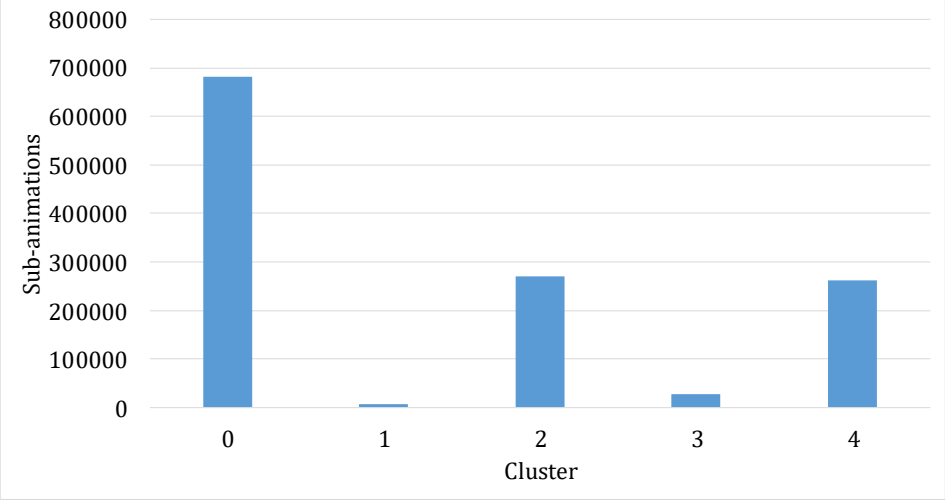


Figura 7.26: Gráfica con la cantidad de sub-animaciones por para cada cluster para el video de control de tráfico y 5 neuronas.

Con base en lo anterior, a continuación se proporciona la descripción de cada uno de los clusters seleccionados:

- Para el cluster 0 se muestra su secuencia de frames en la figura 7.27, en donde se observa que se agrupa información relacionada con las regiones oscuras del pavimento, también se agrupan las regiones de los automóviles que pasan a través de dichas regiones.

  En la figura 7.28 se muestra el 3D-EVM generado para el cluster 0, en donde se observan las regiones por donde pasan los autos y la parte del centro de la carretera que divide los carriles principales. No obstante, se observa que este cluster contiene una cantidad considerable de información, ello con base en que se está utilizando una cantidad reducida de neuronas

para el SOM, lo cual aparentemente no permite separar adecuadamente las regiones de la animación.

- En la figura 7.29 se muestra la secuencia de frames obtenida para el 3D-EVM del cluster 2 y con la información de color obtenida del 6D-EVM de la animación original. Se observa que se agrupan regiones más claras del pavimento, no obstante también se agrupan las regiones que se generan con el paso de los vehículos por dichas zonas.

  En la figura 7.30 se muestra el 3D-EVM generado para el cluster 2, Como se puede apreciar, se tienen regiones que se desplazan sobre el eje del tiempo, las cuales corresponden a los autos que se desplazan sobre el pavimento.

- Por último, para el cluster 4 se muestra su secuencia de frames en la figura 7.31, en donde se observa que se obtiene una separación de las regiones que corresponden a los autos que se desplazan en el pavimento. Sin embargo, también se agrupan regiones que corresponden a las zonas verdes que se encuentran en las orillas de la carretera. En este caso se obtuvo una mejor separación de las regiones de los autos con respecto al pavimento, lo cual es muy deseable para aplicaciones de monitoreo y vigilancia.

  En la figura 7.32 se muestra el 3D-EVM correspondiente al cluster 4, en donde se observan claramente las regiones de los autos y sus desplazamientos.

Figura 7.27: Video de control de trafico y 5 neuronas, resultados de agrupamiento del cluster 0.

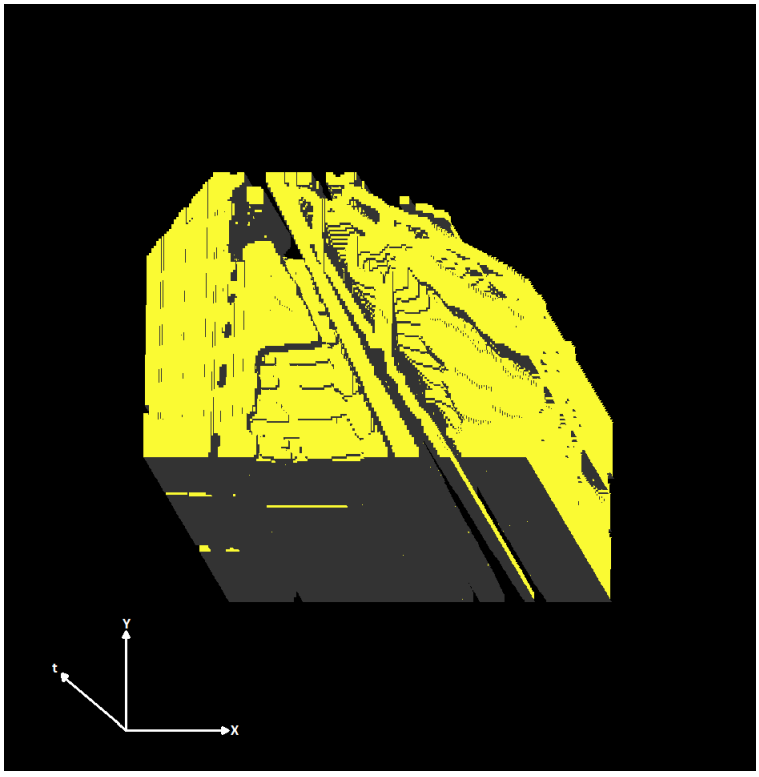


Figura 7.28: 3D-EVM del cluster 0 para el video de control de trafico y 5 neuronas.

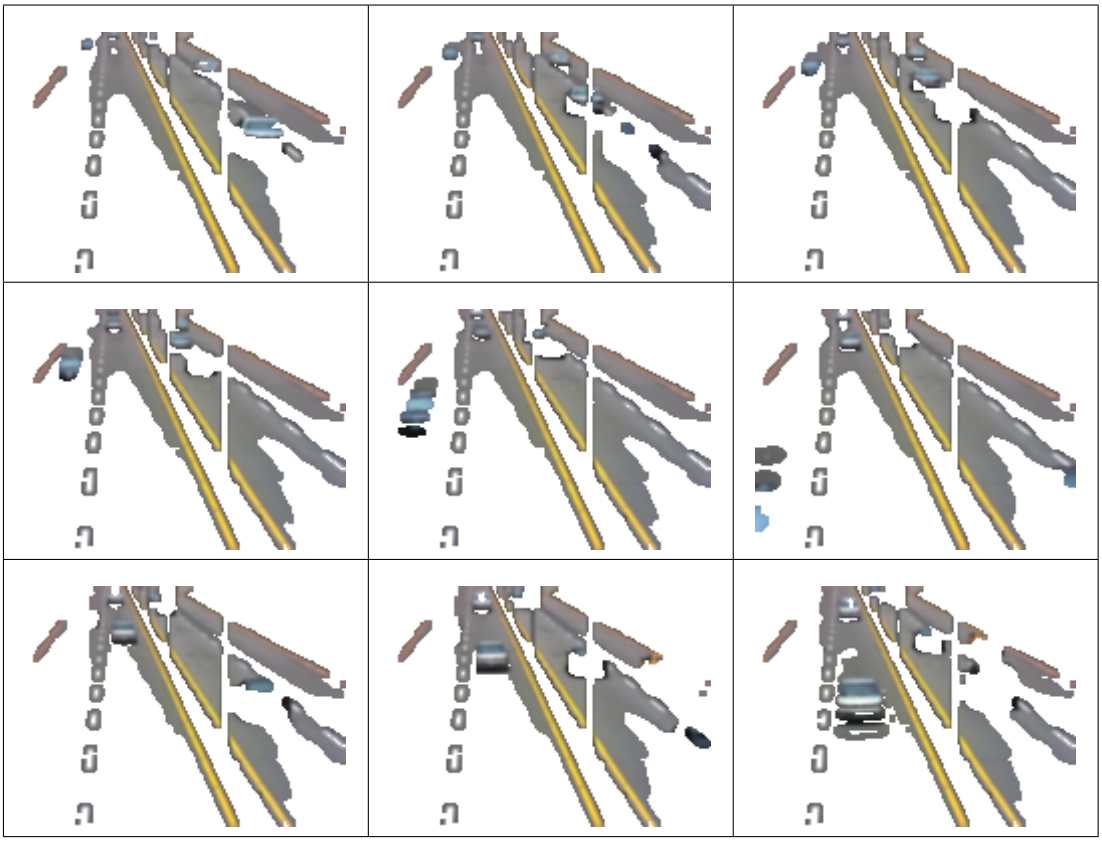


Figura 7.29: Video de control de trafico y 5 neuronas, resultados de agrupamiento del cluster 2.

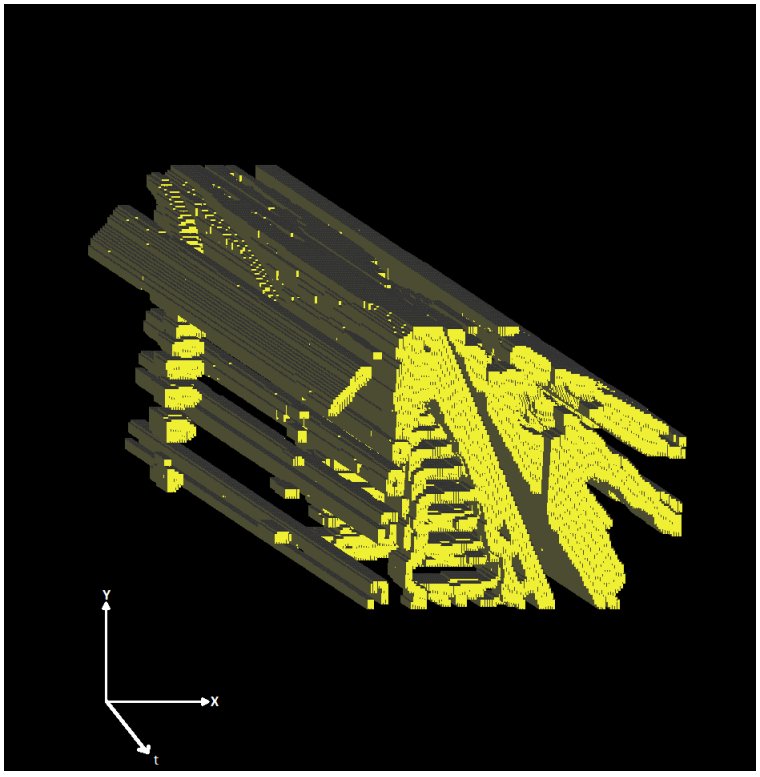


Figura 7.30: 3D-EVM del cluster 2 para el video de control de trafico y 5 neuronas.

Figura 7.31: Video de control de trafico y 5 neuronas, resultados de agrupamiento del cluster 4.

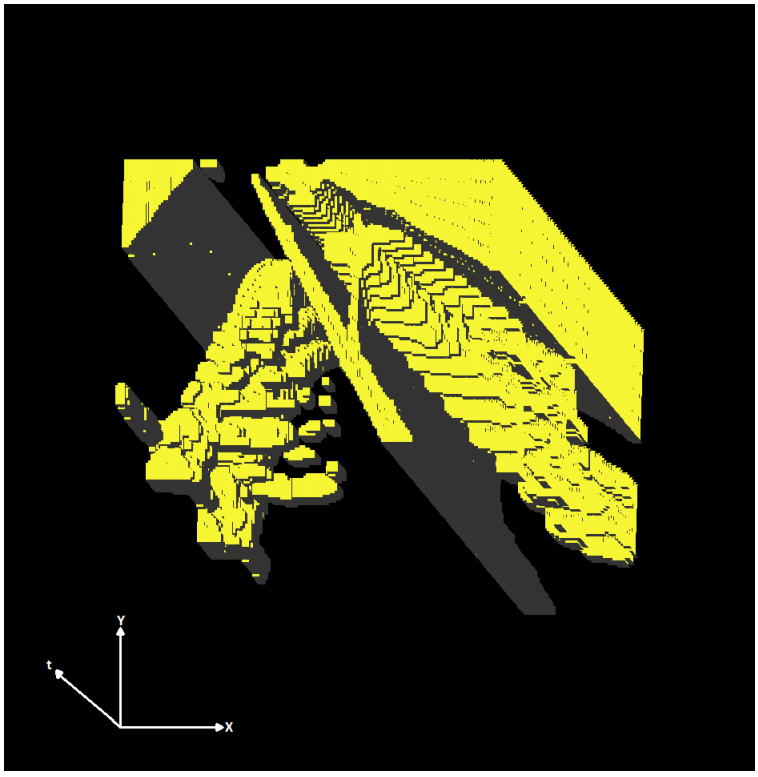


Figura 7.32: 3D-EVM del cluster 4 para el video de control de trafico y 5 neuronas.

7.2.2. Pruebas con 10 neuronas

En este apartado se analizan los resultados de agrupamiento de sub-animaciones mediante el SOM y 10 neuronas. En la tabla 7.5 se muestra la cantidad de vértices extremos obtenidos en los 3D-EVMs formados por cada cluster. Como se observa, se obtiene un total de 156806 vértices extremos, lo cual en comparación con la cantidad de vértices extremos del 6D-EVM de la animación original se obtiene una tasa de compresión de aproximadamente 108.78. Con lo anterior se obtiene una compactación de información mediante el agrupamiento de un SOM y los valores de DC para las sub-animaciones.

Tabla 7.5: Tamaño de los 3D-EVMs generados por cada cluster para el video de control de tráfico y 10 neuronas.

Cluster	Tamaño del 3D-EVM
0	13668
1	38192
2	26348
3	9678
4	272
5	2948
6	36070
7	23618
8	922
9	5090
Total	156806

En la figura 7.33 se muestra la cantidad de sub-animaciones agrupada por cada cluster, en donde se observa que los clusters con una mayor cantidad de información son el cluster 1, 2 y 6. Por ende, éstos clusters son analizados para observar su contenido.

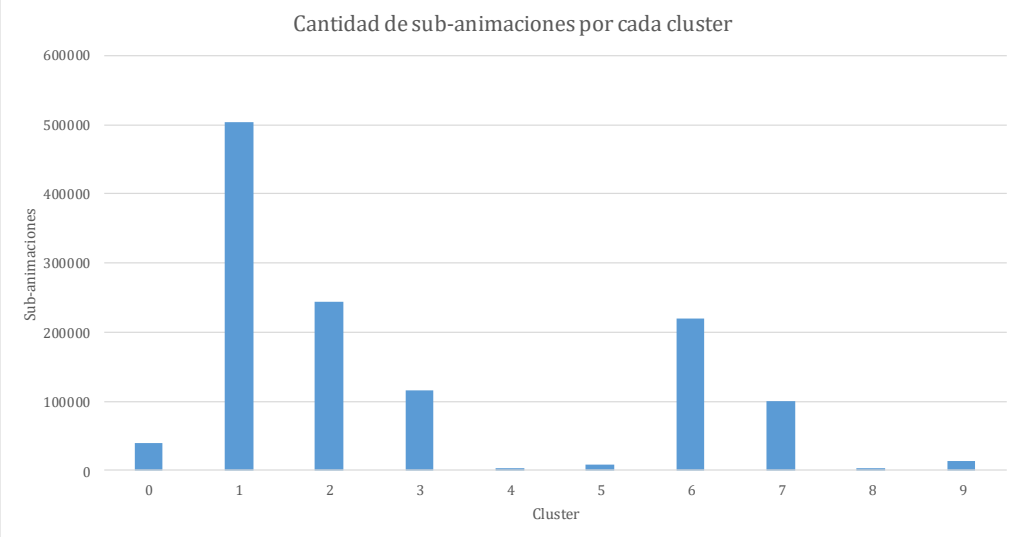


Figura 7.33: Gráfica con la cantidad de sub-animaciones por cada cluster para el video de control de tráfico y 10 neuronas.

Con base en lo anterior, a continuación se proporciona una descripción de los clusters mencionados:

- En la figura 7.34 se muestra la secuencia de frames obtenida del 3D-EVM generado en el cluster, esto considerando que se consulta la animación original para obtener información de

color. Como se observa en la figura 7.34, se agrupa regiones asociadas a las zonas más oscuras en el pavimento, no obstante también se está agrupando el movimiento de autos sobre éste.

En la figura 7.35 se muestra el 3D-EVM generado en el cluster 1, en donde se observa que se tiene una cantidad considerable de regiones. Y también se alcanza a apreciar que se registra el movimiento de los autos.

- En la figura 7.36 se muestra la secuencia de frames obtenida para el 3D-EVM del cluster 2 y la información de color para las regiones contenidas en él, en donde se observa que el agrupamiento permite prácticamente separar los autos en movimiento de las regiones donde se encuentra el pavimento, y también se observa que se tienen regiones asociadas a las zonas verdes en el video. Este resultado es importante, ya que el SOM halla patrones de movimiento en las sub-animaciones.

  En la figura 7.37 se muestra el 3D-EVM generado para el cluster 2, en donde se observa que se tiene la evolución con respecto al tiempo de las regiones correspondientes a los autos.

- Por último, en la figura 7.38 se muestran los resultados de agrupamiento para la neurona 6. En este resultado se aprecia que las regiones agrupadas corresponden a zonas claras del pavimento, no obstante se agrupa también el movimiento que se registra debido a los autos en movimiento. En la figura 7.39 se muestra el 3D-EVM obtenido para el cluster 6, en donde se observa las regiones correspondientes al resultado en los frames.

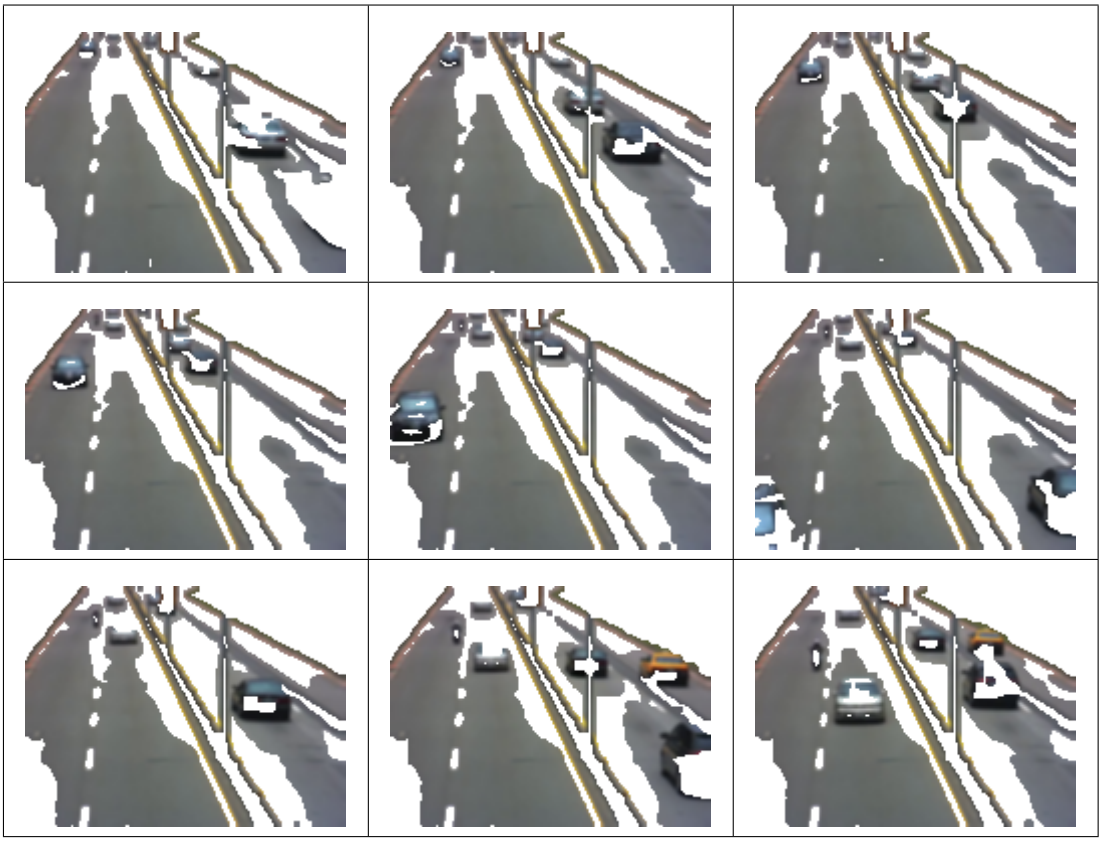


Figura 7.34: Video de control de trafico y 10 neuronas, resultados de agrupamiento del cluster 1.

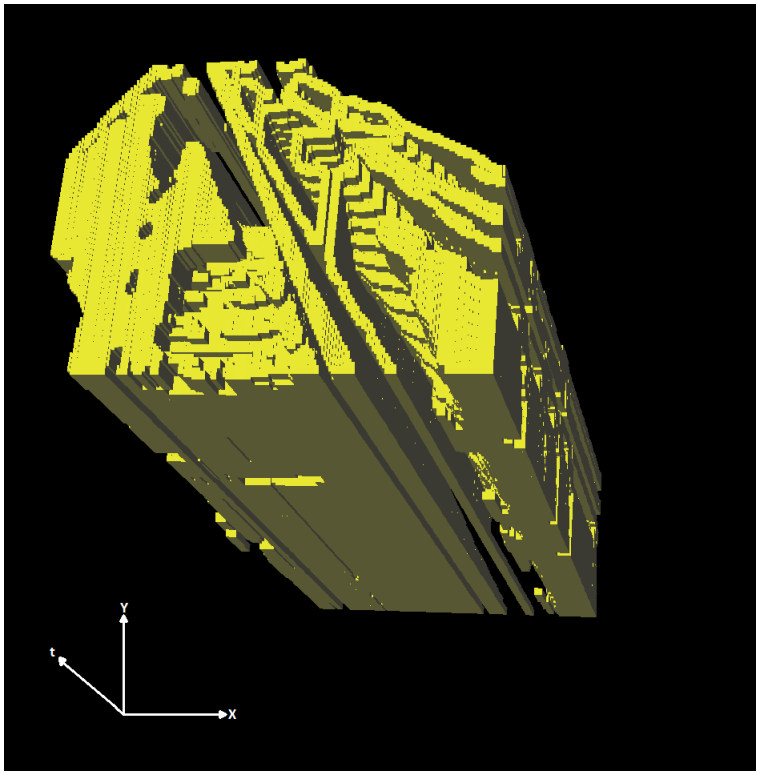


Figura 7.35: 3D-EVM del cluster 1 para el video de control de trafico y 10 neuronas.

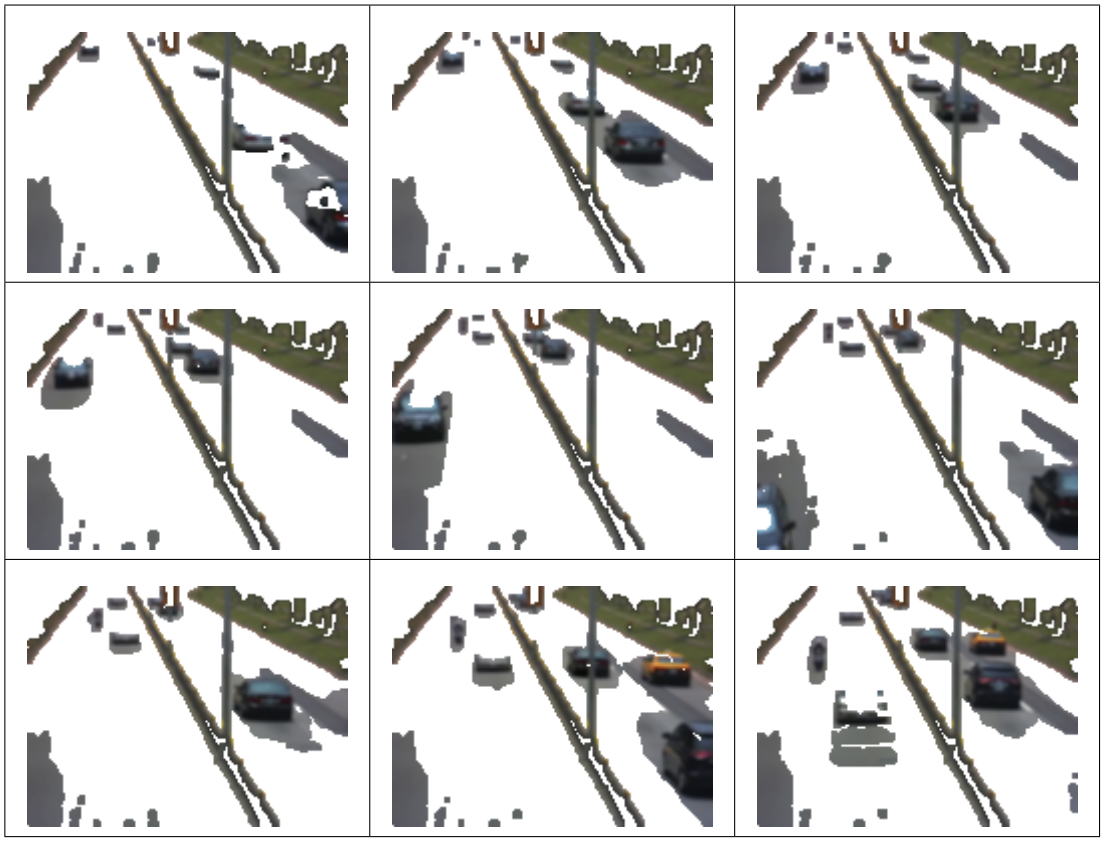


Figura 7.36: Video de control de trafico y 10 neuronas, resultados de agrupamiento del cluster 2.

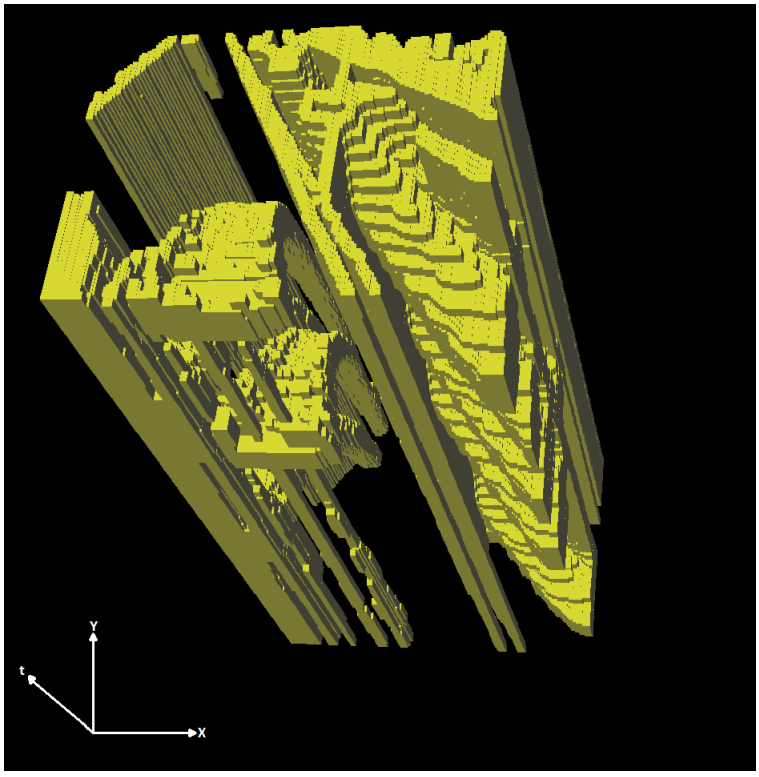


Figura 7.37: 3D-EVM del cluster 2 para el video de control de trafico y 10 neuronas.

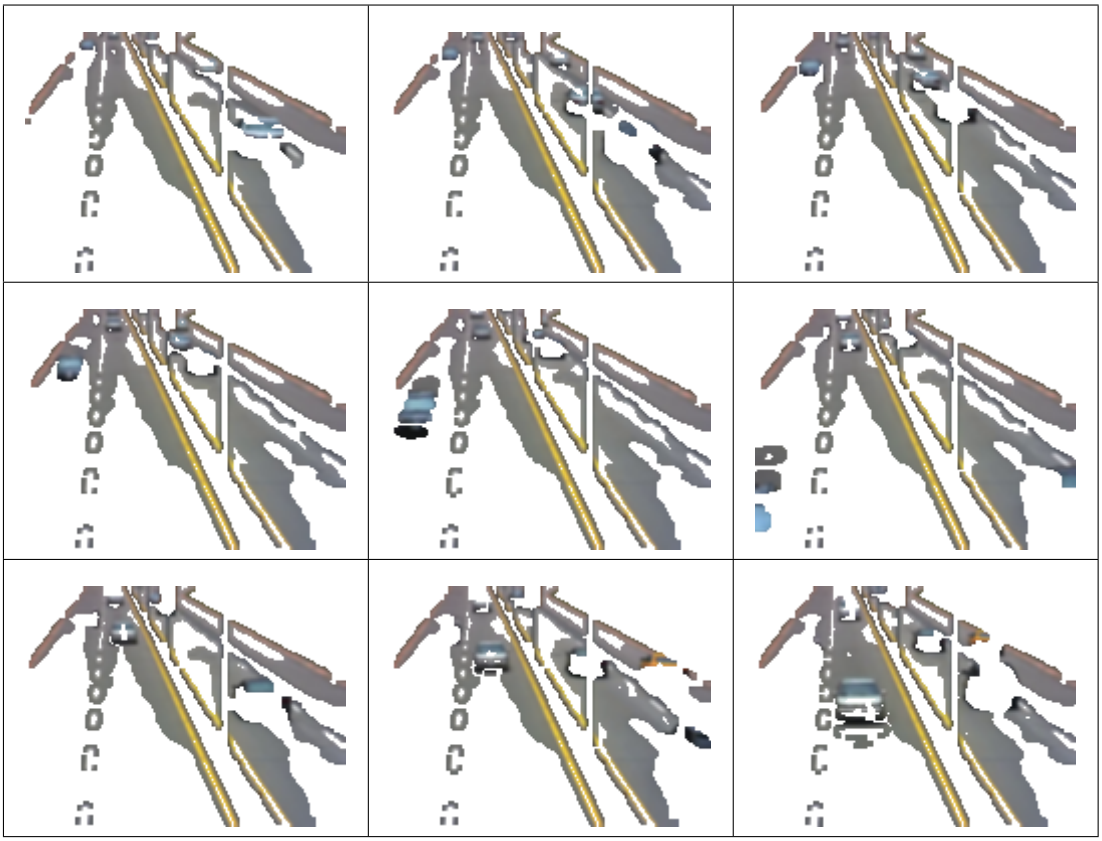


Figura 7.38: Video de control de trafico y 10 neuronas, resultados de agrupamiento del cluster 6.

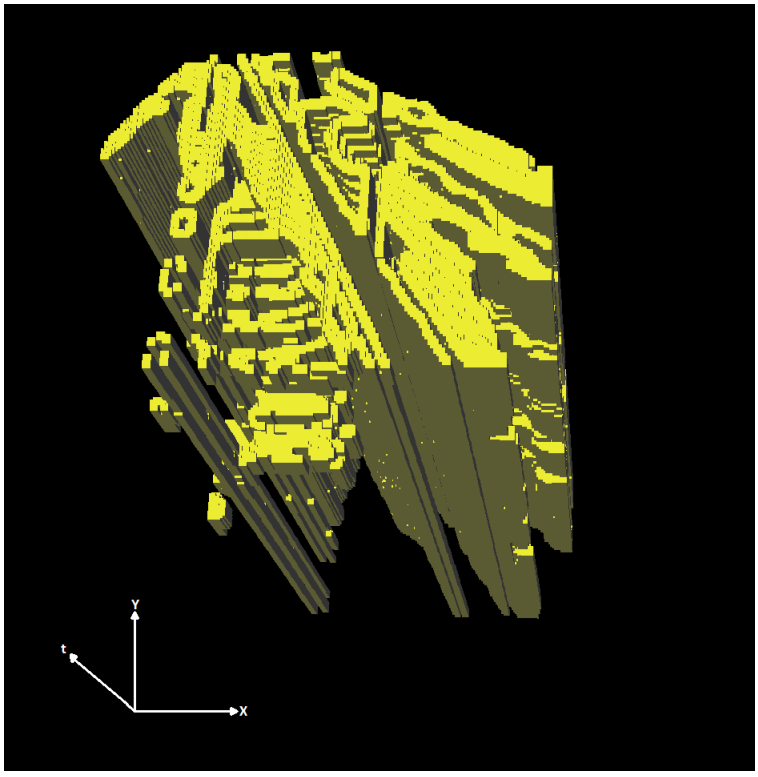


Figura 7.39: 3D-EVM del cluster 6 para el video de control de trafico y 10 neuronas.

7.2.3. Pruebas con 15 neuronas

En este aparado se presentan los resultados de agrupamiento para una configuración de 15 neuronas en el SOM. En la tabla 7.6 se muestra la cantidad de vértices extremos por cada 3D-EVM de cada cluster formado, en donde se tiene un total de 206140 vértices extremos, lo cual en comparación con el tamaño del 6D-EVM de la animación original, se obtiene una tasa de compresión de aproximadamente 82.75.

Tabla 7.6: Tamaño de los 3D-EVMs generados por cada cluster para el video de control de tráfico y 15 neuronas.

Cluster	Tamaño del 3D-EVM
0	176
1	38826
2	8560
3	15116
4	3070
5	23346
6	1408
7	8352
8	33272
9	45096
10	496
11	0
12	0
13	4796
14	23626
Total	206140

En la figura 7.40 se muestra la cantidad de sub-animaciones agrupadas por cada cluster, en donde se observa que los clusters que contiene mayor información son los clusters 1, 5 y 9. Por ende, éstos clusters son analizados para observar su contenido.

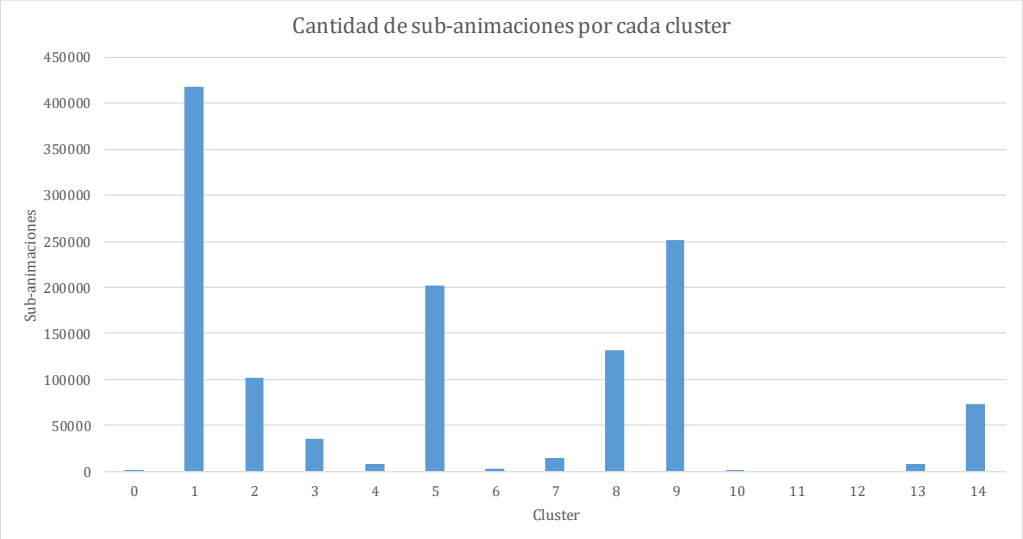


Figura 7.40: Gráfica con la cantidad de sub-animaciones por cada cluster para el video de control de tráfico y 15 neuronas.

Con base en lo anterior, a continuación se proporciona una descripción de los clusters mencionados:

- Para el cluster 1, en la figura 7.41 se muestra la secuencia de frames para dicho cluster. En esta secuencia de frames se observa que se agrupan regiones donde el pavimento es más oscuro, no obstante también se registran las regiones por donde hay movimiento de autos. Con lo anterior el 3D-EVM que se forma con las regiones de las sub-animaciones se muestra en la figura 7.42.
- Para el cluster 5, en la figura 7.43 se muestra la secuencia de frames obtenida de las sub-animaciones agrupadas. Se observa que en este caso también se realiza una separación adecuada entre los autos en movimiento y el pavimento. No obstante, se agrupan regiones que pertenecen a las zonas verdes de la escena. Este resultado es importante ya que ayuda a identificar patrones de movimiento.

  En la figura 7.44 se muestra el 3D-EVM formado por las regiones agrupadas, en donde se observa claramente la evolución en el tiempo de las regiones correspondientes a los autos en movimiento.
- Por último en la figura 7.45 se presentan los resultados de agrupamiento para el cluster 9, en donde se observa que se agrupan las regiones del pavimento que son relativamente claras, no obstante las regiones agrupadas también se ven afectadas por los autos en movimiento. En la figura 7.46 se muestra el 3D-EVM para el cluster 9.

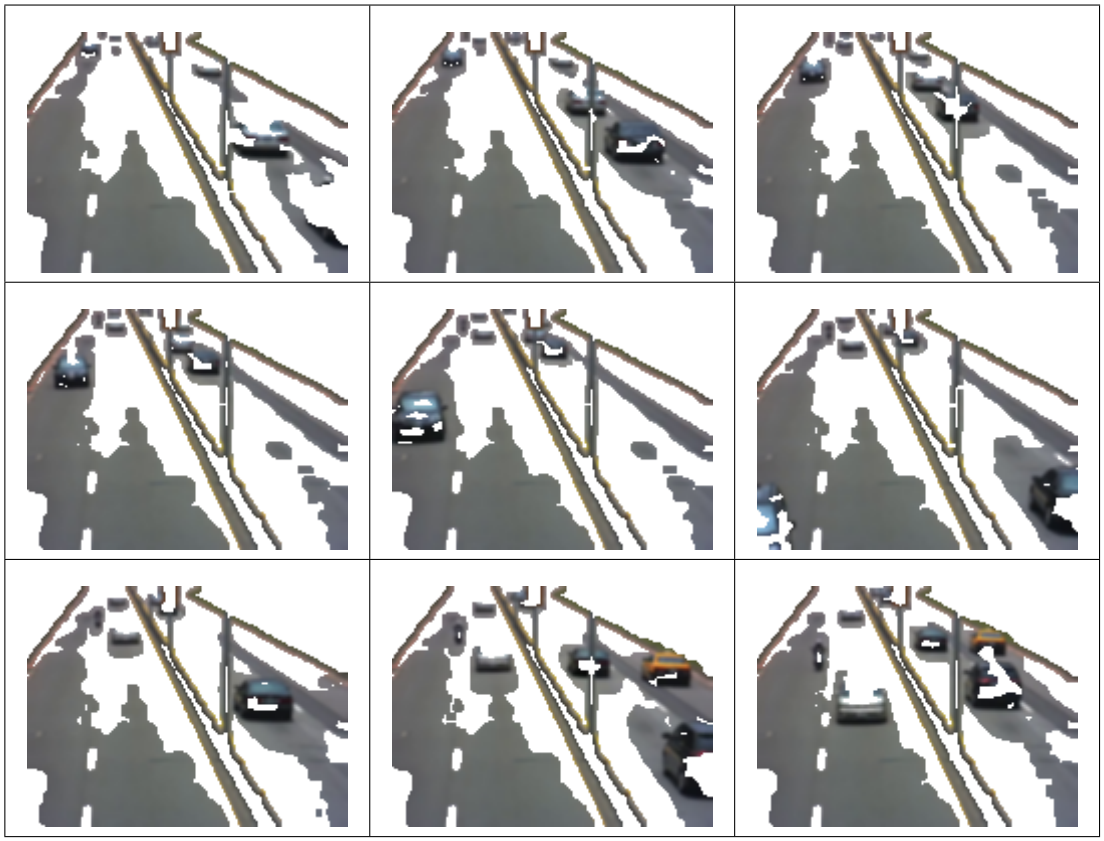


Figura 7.41: Video de control de trafico y 15 neuronas, resultados de agrupamiento del cluster 1.

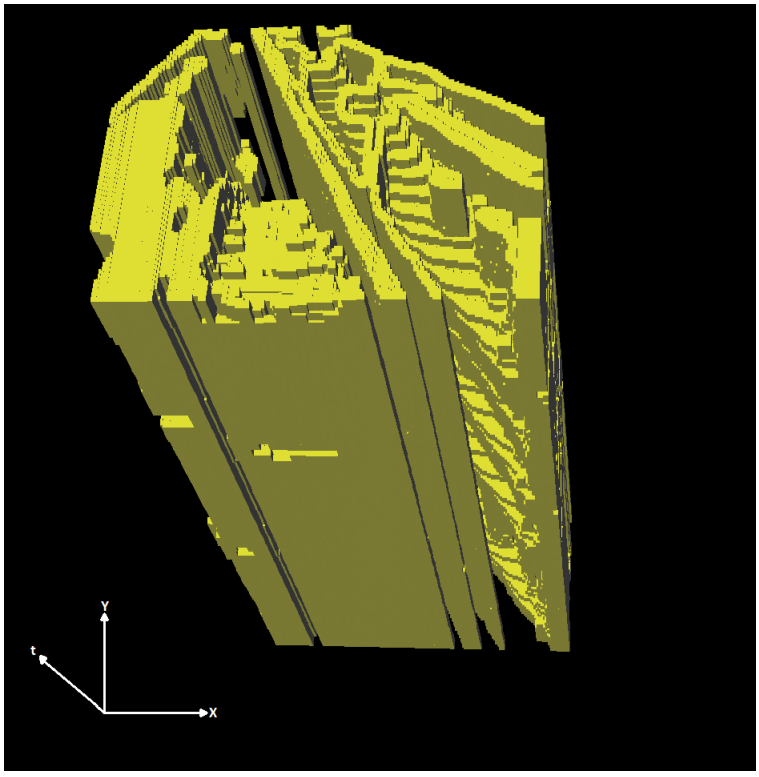


Figura 7.42: 3D-EVM del cluster 1 para el video de control de trafico y 15 neuronas.

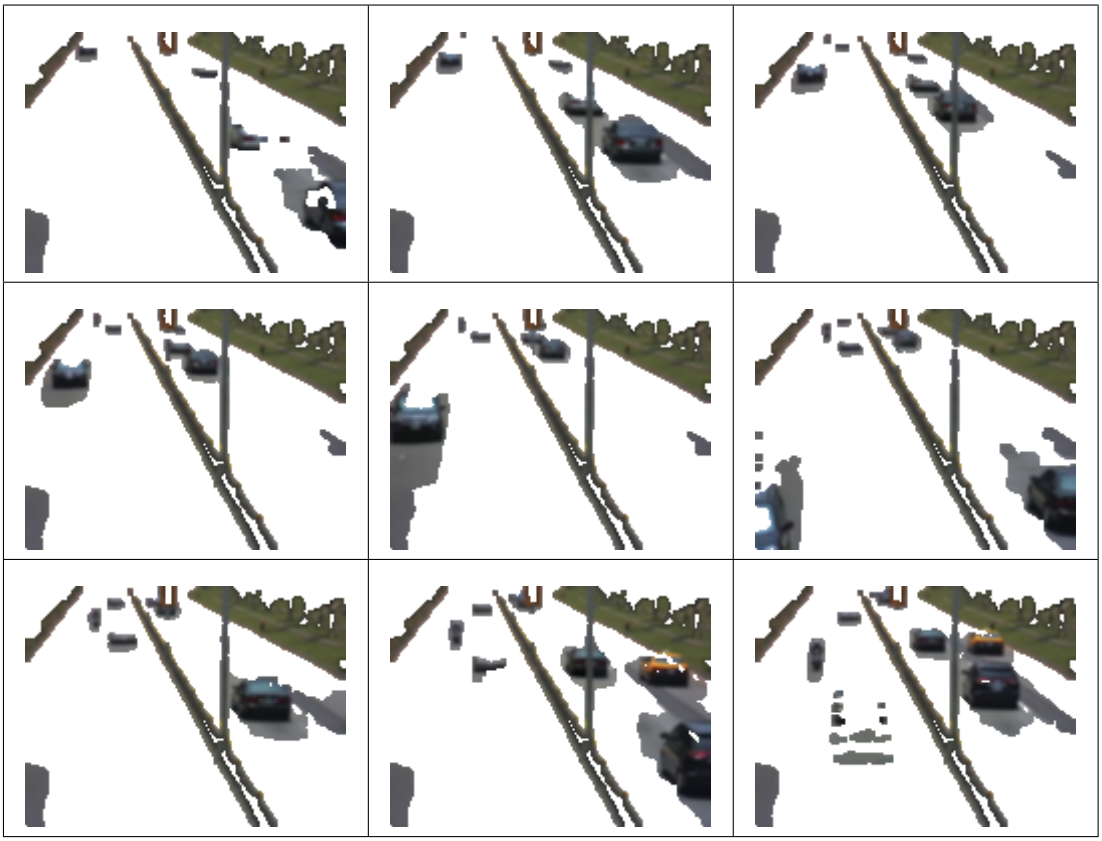


Figura 7.43: Video de control de trafico y 15 neuronas, resultados de agrupamiento del cluster 5.

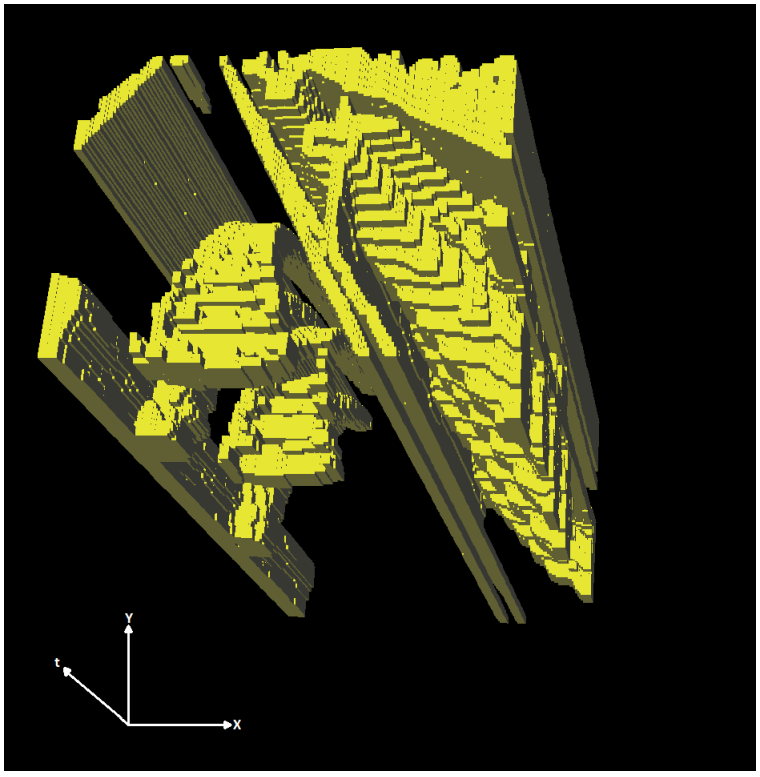


Figura 7.44: 3D-EVM del cluster 5 para el video de control de trafico y 15 neuronas.

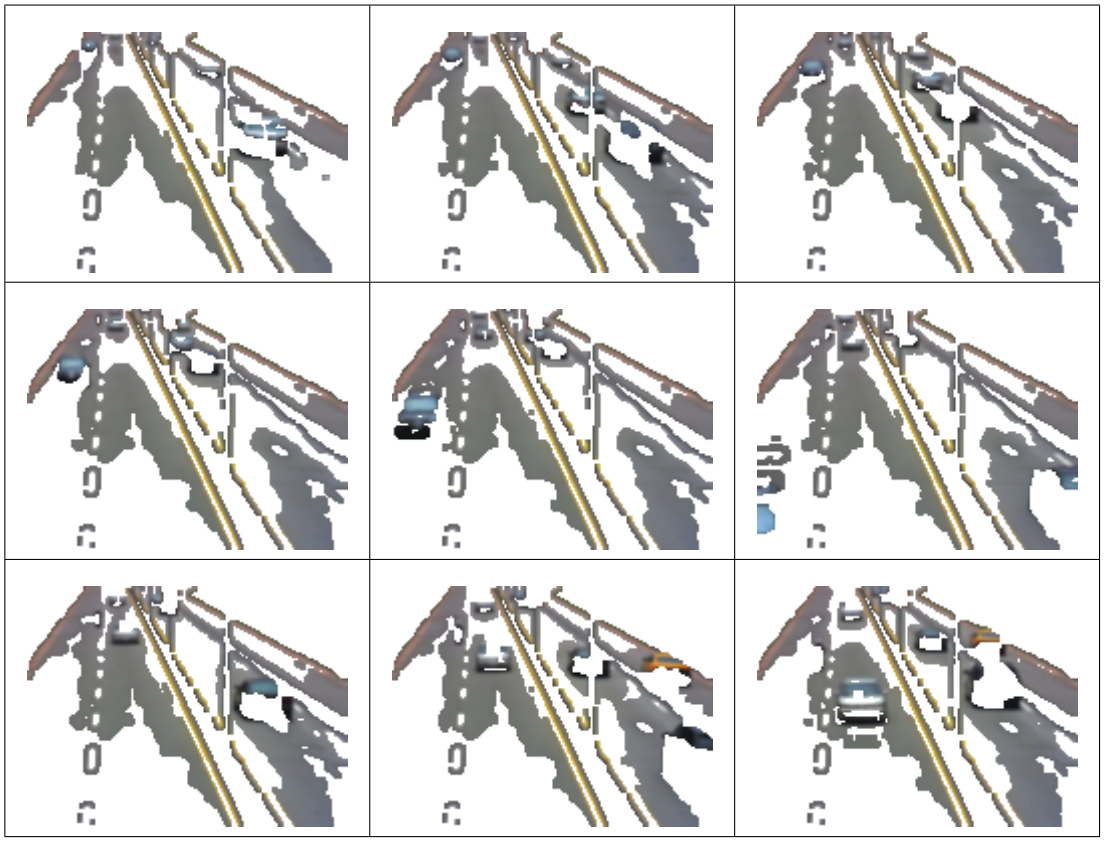


Figura 7.45: Video de control de trafico y 15 neuronas, resultados de agrupamiento del cluster 9.

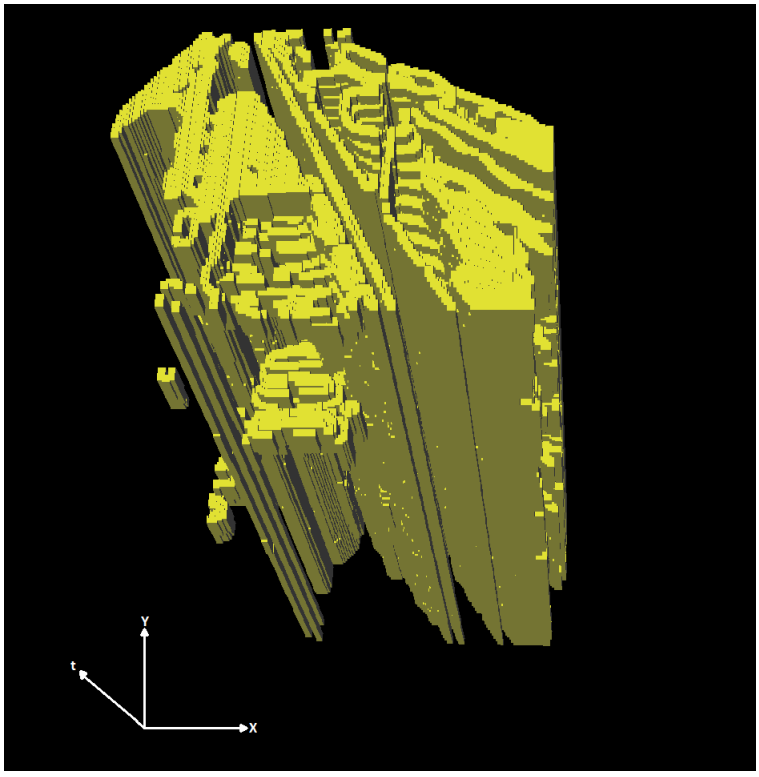


Figura 7.46: 3D-EVM del cluster 9 para el video de control de trafico y 15 neuronas.

7.3 Pruebas con un Video Animado

Este apartado presenta los resultados obtenidos de aplicar el proceso de segmentación a un video animado, el cual es una pequeña parte de la película animada *Los Increibles - Pixar* [53]. Como se observa, solo se analiza una escena de este video, en donde se encuentra un bebé sentado en el suelo y con juguetes alrededor de él. El bebé es el único que se mueve en la escena el resto de la escena es estático. Con esto los cambios en los frames radican en el movimiento del bebé, y aunque es un video generado por computadora (digital), se presentan ciertas diferencias entre frames consecutivos. En la figura 7.47 se muestran algunos frames de dicho video.

Figura 7.47: Algunos frames del video animado.

Los frames de este video tienen un tamaño de 160x120 y la cantidad de frames que dura la escena son 37. Con base en lo anterior, se obtuvo la representación de la animación en el 6D-EVM, el cual tiene un tamaño de 4454534 vértices extremos considerando que el video tiene un esquema de color RGB. Posteriormente se realizaron las pruebas de segmentación mediante el SOM y para 5, 10 y 15 neuronas.

7.3.1. Pruebas con 5 neuronas

En este apartado se analizan los resultados de segmentación obtenidos mediante un SOM y 5 neuronas. En la tabla 7.7 se muestran los tamaños de los 3D-EVMs correspondientes a cada cluster, y como se observa se tiene un total de 15600 vértices extremos, con lo cual en comparación con la cantidad de vértices extremos de la animación original, se obtiene una tasa de compresión de 285.54 aproximadamente.

Tabla 7.7: Tamaño de los 3D-EVMs generados por cada cluster para el video animado y 5 neuronas.

Cluster	Tamaño del 3D-EVM
0	3080
1	4784
2	5538
3	2190
4	8
Total	15600

En la figura 7.48, se muestra de manera gráfica la cantidad de sub-animaciones que contiene cada cluster, en donde se observa que los clusters con una mayor cantidad de información son los clusters 0, 1 y 2.

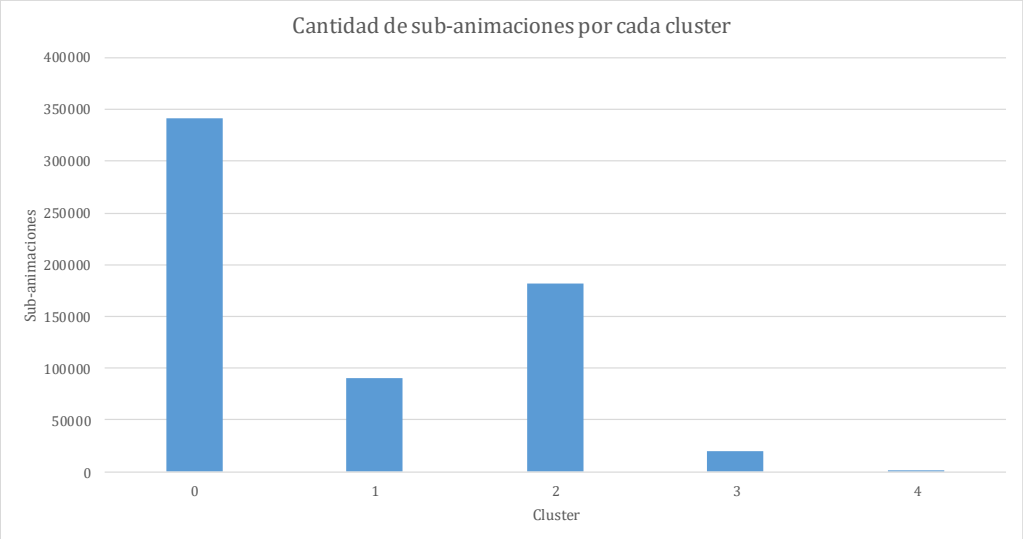


Figura 7.48: Gráfica con la cantidad de sub-animaciones por cada cluster para el video animado y 5 neuronas.

En base lo anterior, ahora se realiza un análisis con respecto al contenido de los clusters mencionados anteriormente:

- En el cluster 0 se obtiene una secuencia de frames que se muestra en la figura 7.49, en donde se observa que las regiones que se están agrupando son aquellas en donde hay sombras o regiones oscuras. Este resultado es importante en el sentido que con base en las sombras en una imagen se puede ayudar a determinar la cantidad de objetos en ella, véase por ejemplo [51, 55].

  En la figura 7.50 se muestra el correspondiente 3D-EVM del cluster 0. En donde se pueden observar prismas en las zonas de las sombras o regiones oscuras, los cuales se forman así debido a que el fondo de la escena no cambia y por ende en todos los frames están presentes dichas zonas.

- Para el cluster 1, en la figura 7.51 se muestra la secuencia de frames obtenida, en donde se observa que se agrupan las regiones más claras de la escena, las cuales están asociadas al cuerpo del bebé y la parte de la alfombra que está mas iluminada.

  En la figura 7.52 se muestra el correspondiente 3D-EVM del cluster 1, en el cual se observan las regiones mencionadas anteriormente, pero considerando su desplazamiento en el tiempo se forman prismas.

- El cluster 2 del cual se muestra su secuencia de frames en la figura 7.53, en donde se observa que se agrupan las regiones en donde se encuentran los bordes de los objetos y el bebé. Esto tiene sentido con base en que en las zonas donde hay bordes se presenta un cambio en el color de manera busca, y con ello se podría decir que estas regiones son similares en cuanto a su forma.

  En la figura 7.54 se muestra el 3D-EVM correspondiente al cluster, en donde se observa, los bordes antes mencionados y la formación de los prismas a lo largo del eje coordenado del tiempo.

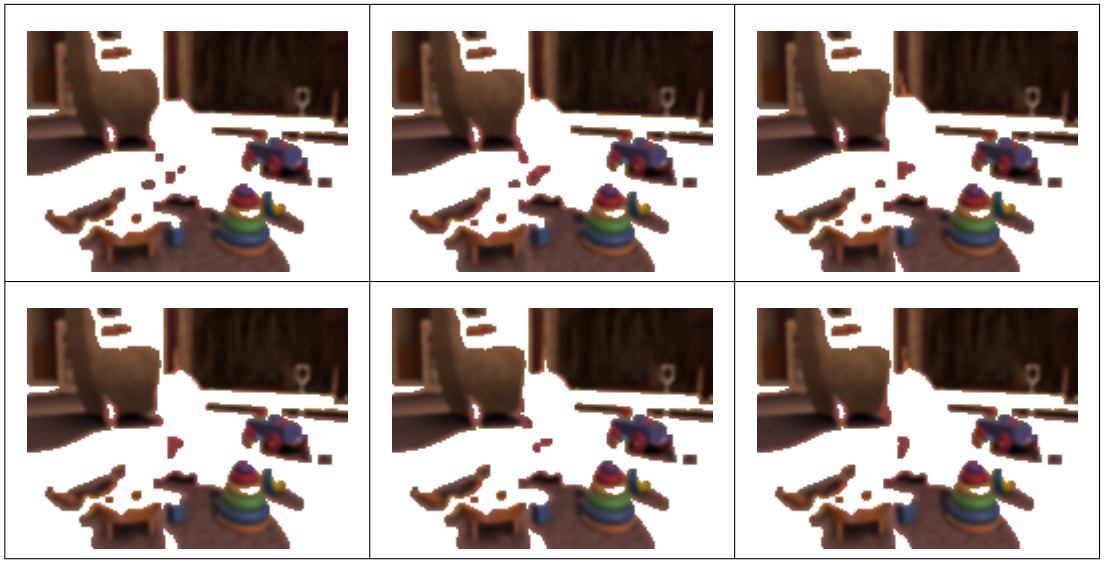


Figura 7.49: Video de animado y 5 neuronas, resultados de agrupamiento del cluster 0.

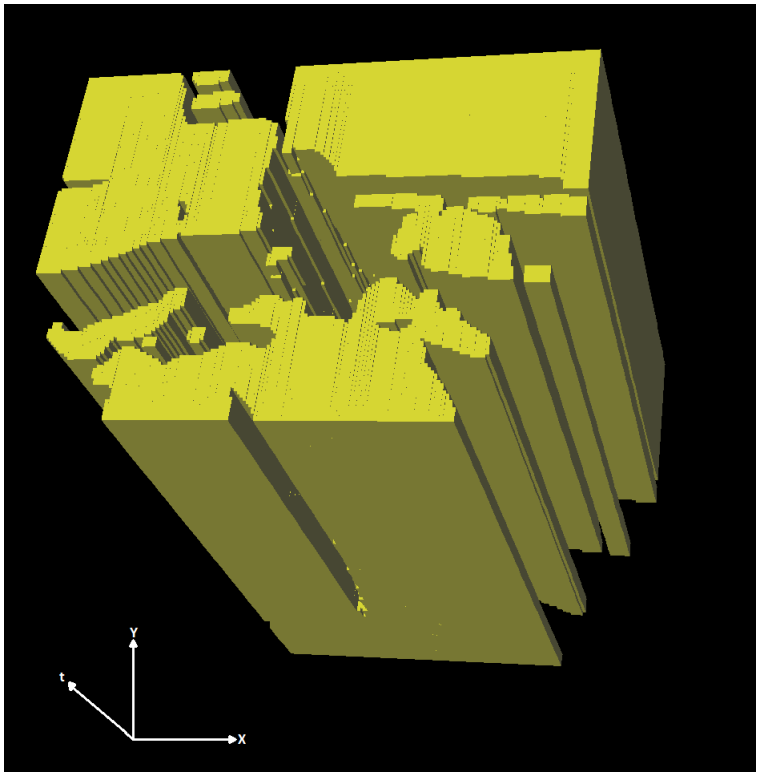


Figura 7.50: 3D-EVM del cluster 0 para el video animado y 5 neuronas.

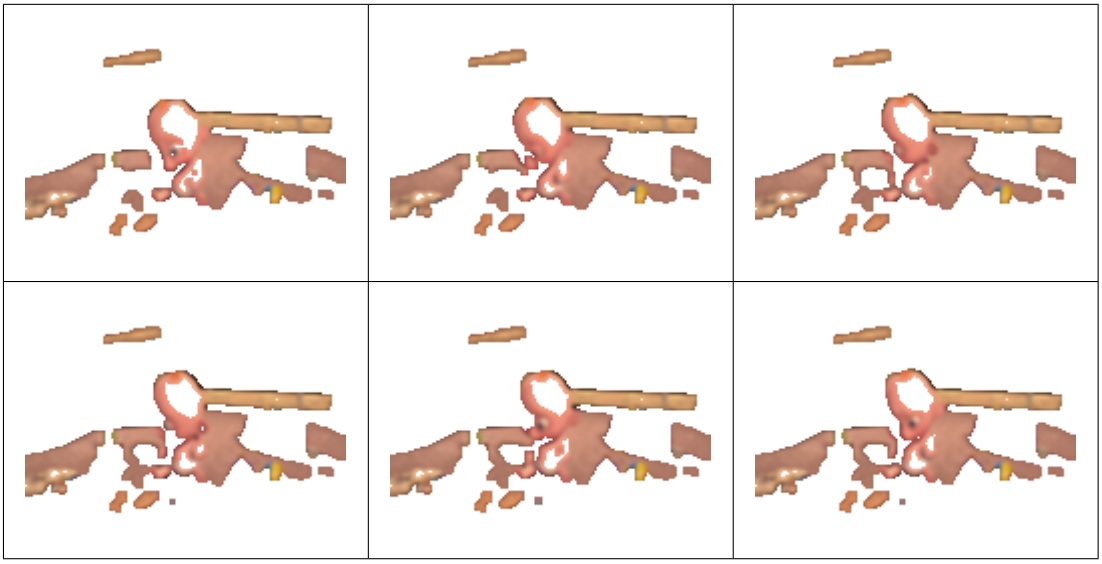


Figura 7.51: Video de animado y 5 neuronas, resultados de agrupamiento del cluster 1.

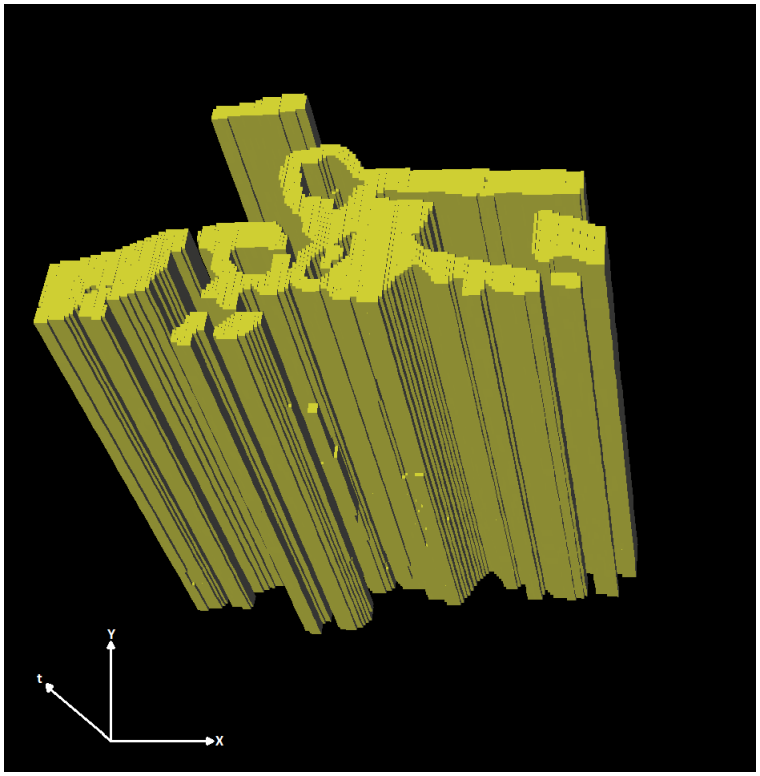


Figura 7.52: 3D-EVM del cluster 1 para el video animado y 5 neuronas.

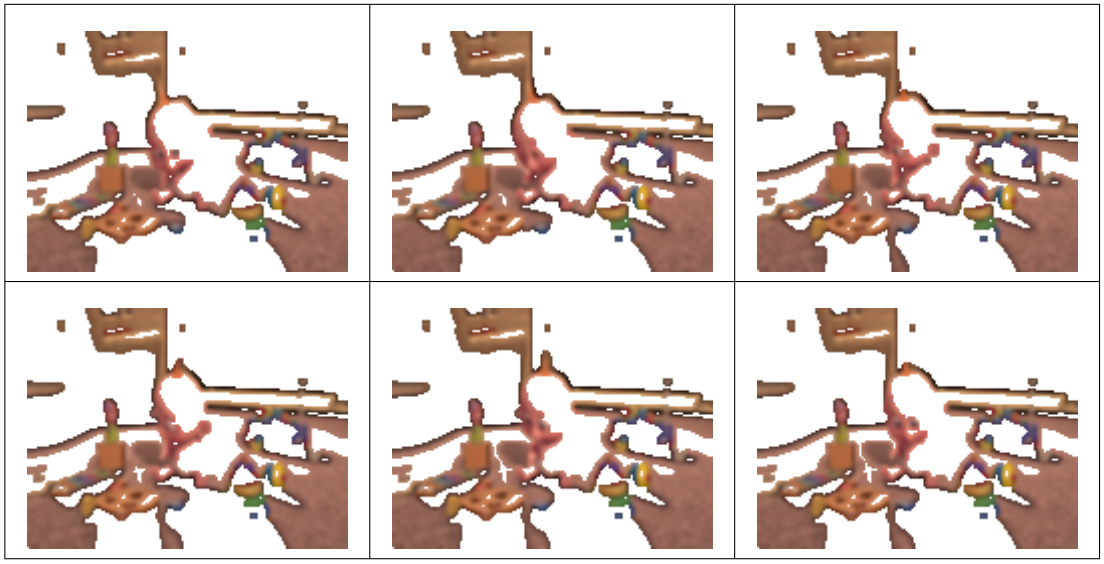


Figura 7.53: Video de animado y 5 neuronas, resultados de agrupamiento del cluster 2.

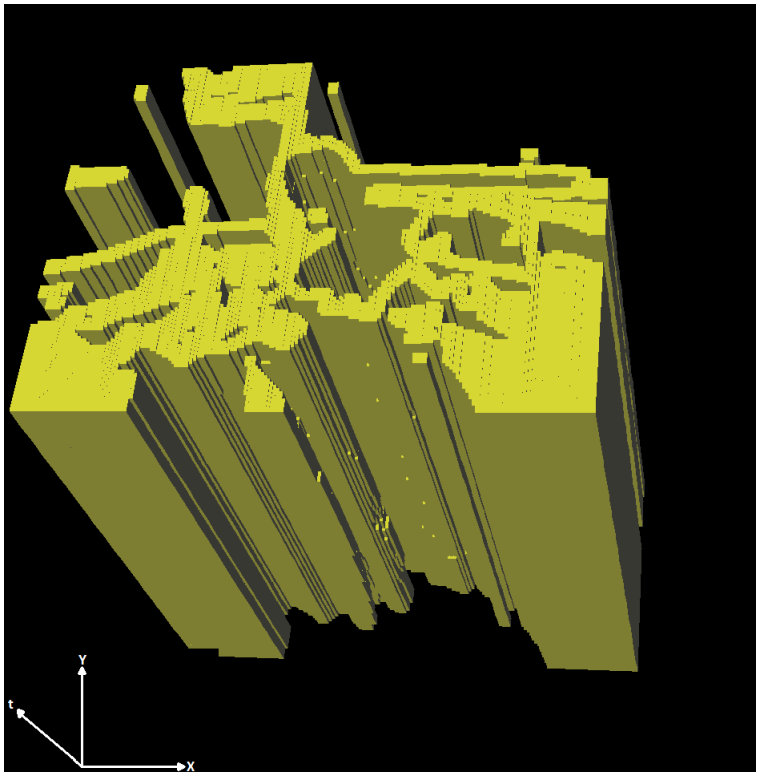


Figura 7.54: 3D-EVM del cluster 2 para el video animado y 5 neuronas.

7.3.2. Pruebas con 10 neuronas

En este apartado se presentan los resultados de las pruebas realizadas para el video animado y una configuración de 10 neuronas en el SOM. En la tabla 7.8 se muestra el tamaño de los 3D-EVMs formados para cada cluster mediante dicha configuración del SOM. La cantidad total de vértices extremos es de 35068, la cual en comparación con el tamaño del 6D-EVM de la animación original, se obtiene una tasa de compresión de 127 aproximadamente.

Tabla 7.8: Tamaño de los 3D-EVMs generados por cada cluster para el video animado y 10 neuronas.

Cluster	Tamaño del 3D-EVM
0	6610
1	2312
2	0
3	3012
4	4316
5	5260
6	178
7	6366
8	5388
9	1626
Total	35068

En la figura 7.55 se muestra de manera gráfica la cantidad de sub-animaciones que agrupa cada cluster, con lo cual se determina que los custers que contienen información más relevante para la escena son los clusters 0, 1, 5 y 8.

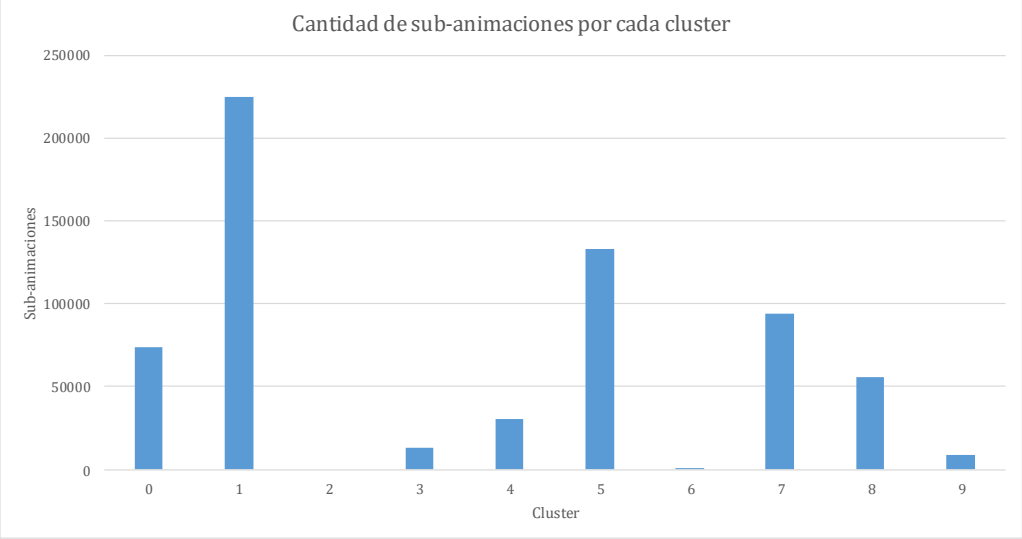


Figura 7.55: Gráfica con la cantidad de sub-animaciones agrupadas en cada cluster para el video animado y 10 neuronas.

Con base en lo anterior, a continuación se da una descripción de los clusters seleccionados:

- En el cluster 0 del cual se tiene su secuencia de frames en la figura 7.56, se observa que se tienen las regiones asociadas a los bordes de los objetos y del bebé. Nuevamente se presenta este comportamiento debido a que las zonas de los bordes tienen una forma similar. En la figura 7.57 se muestra el 3D-EVM correspondiente a este grupo, en donde claramente se observan los bordes de los objetos, y su desplazamiento en el tiempo.

- Para el cluster 1, se tiene su secuencia de frames en la figura 7.58, en donde se puede apreciar que se tienen regiones oscuras de los frames, en particular las regiones donde hay sombras y específicamente las sombras que se encuentran justo debajo de los objetos. Es importante notar que la sombras agrupadas en este cluster pueden en algún momento dado ayudar a determinar la cantidad de objetos presentes en la escena. En la figura 7.59, se observa el 3D-EVM correspondiente a este cluster, en donde se observan las regiones de las sombras y su respectivo desplazamiento en el tiempo.
- Para el cluster 5 se tiene la secuencia de frames que se observa en la figura 7.60, en donde se observa que se tienen regiones donde se encuentran bordes de los objetos. Este resultado es muy parecido al resultado del cluster 0, no obstante se puede notar que en este caso las regiones agrupadas pertenecen a zonas donde los bordes son más notorios y el cambio de color es más pronunciado. En la figura 7.61 se muestra el 3D-EVM correspondiente al cluster 5, en donde se puede observar que claramente se tienen los bordes del frame y su desplazamiento en el tiempo.
- Por último, en la figura 7.62 se muestra la secuencia de frames para el cluster 8, en donde se observa que se tienen regiones asociadas a los elementos más sobresalientes de la escena, que en este caso es el bebé y las partes más claras de la alfombra. Para este cluster, su correspondiente 3D-EVM se muestra en la figura 7.63, en donde se observa claramente que se tiene la silueta del bebé y las partes claras de la alfombra.

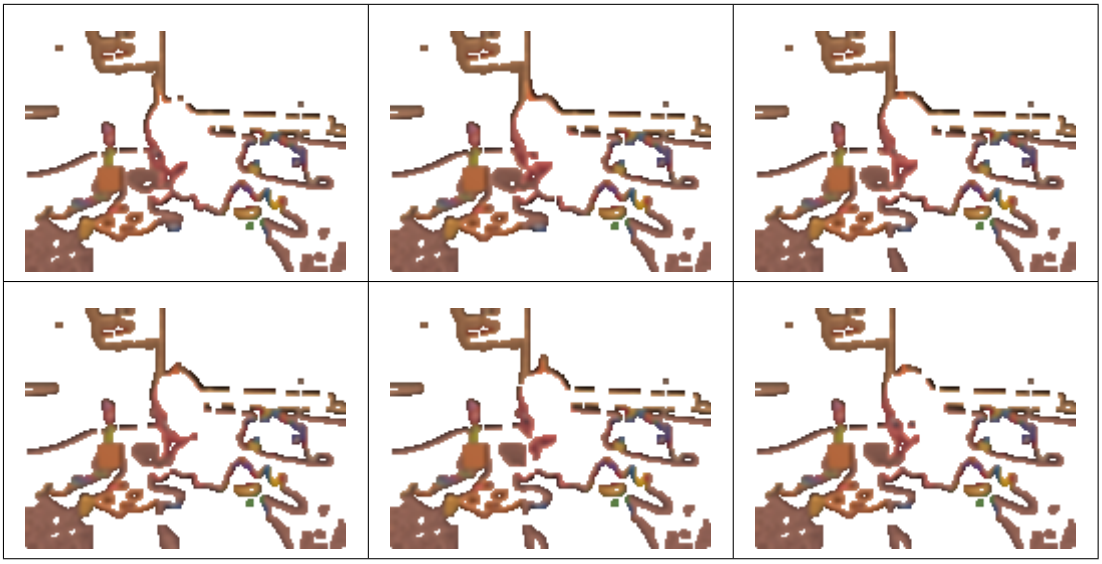


Figura 7.56: Video de animado y 10 neuronas, resultados de agrupamiento del cluster 0.

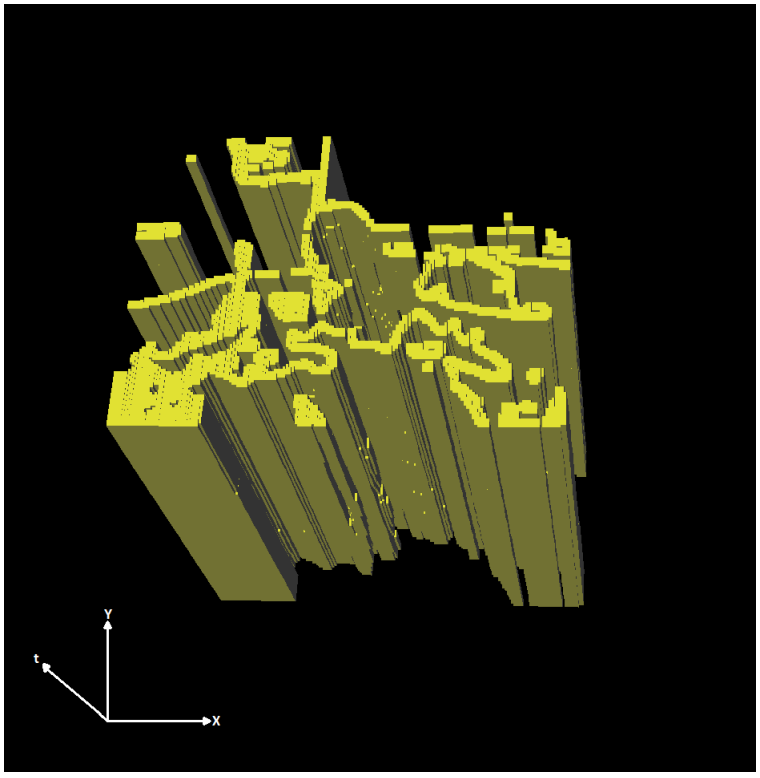


Figura 7.57: 3D-EVM del cluster 0 para el video animado y 10 neuronas.

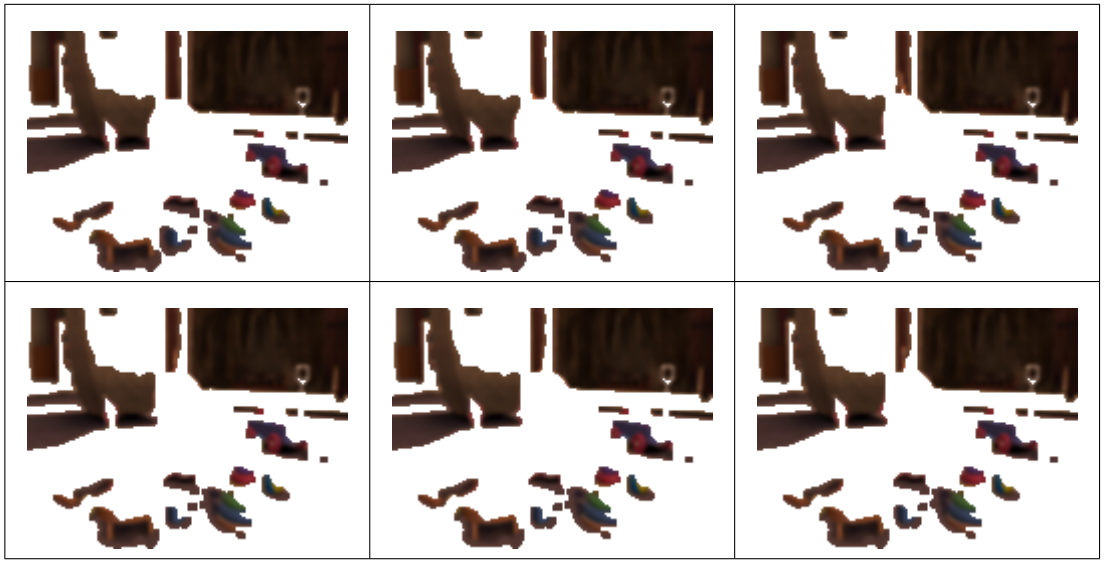


Figura 7.58: Video de animado y 10 neuronas, resultados de agrupamiento del cluster 1.

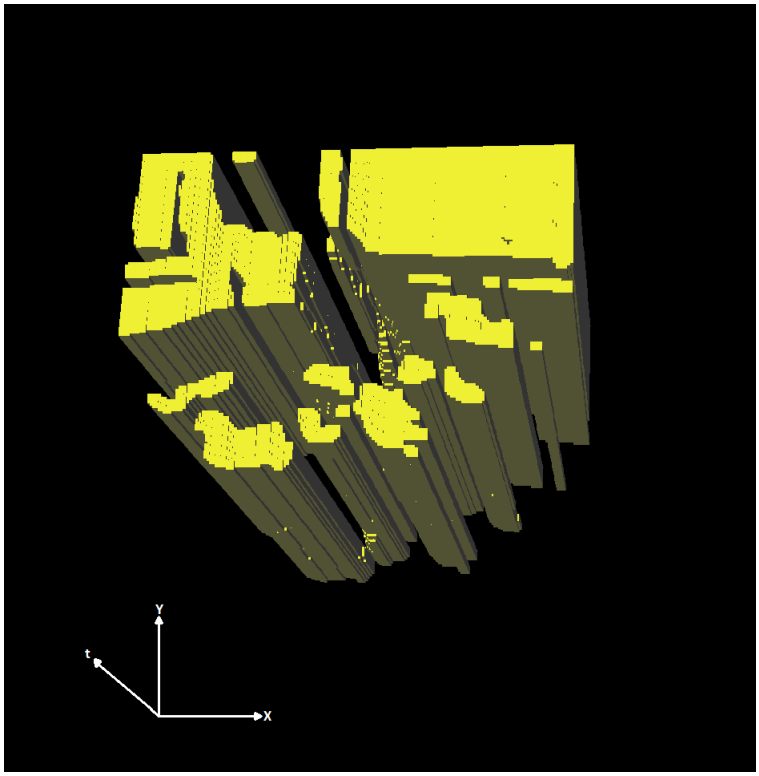


Figura 7.59: 3D-EVM del cluster 1 para el video animado y 10 neuronas.

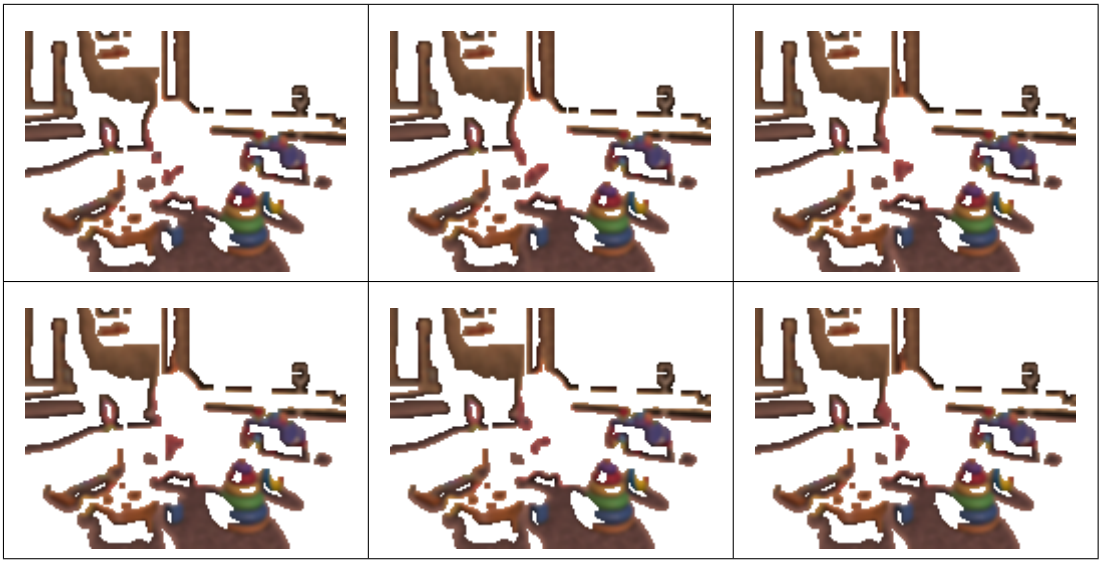


Figura 7.60: Video de animado y 10 neuronas, resultados de agrupamiento del cluster 5.

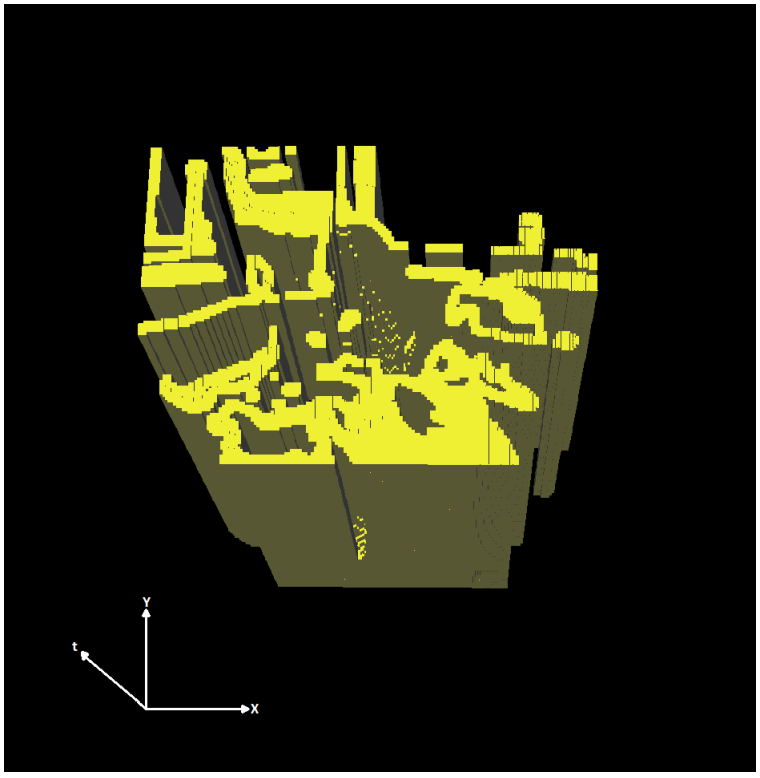


Figura 7.61: 3D-EVM del cluster 5 para el video animado y 10 neuronas.

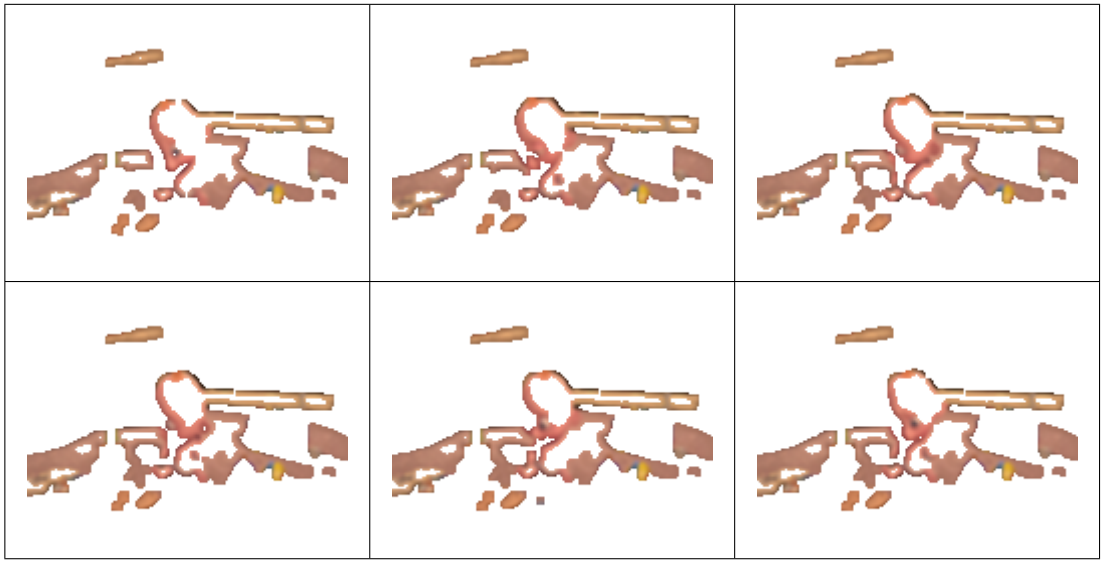


Figura 7.62: Video de animado y 10 neuronas, resultados de agrupamiento del cluster 8.

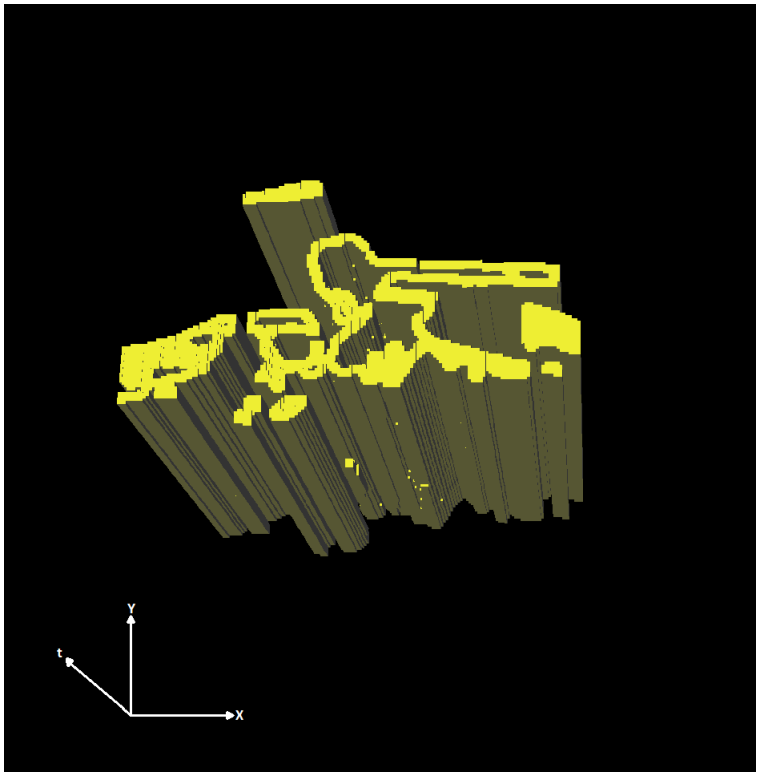


Figura 7.63: 3D-EVM del cluster 8 para el video animado y 10 neuronas.

7.3.3. Pruebas con 15 neuronas

En este apartado se muestran los resultados de aplicar el proceso de segmentación mediante un SOM con 15 neuronas. En la tabla 7.9 se tiene la cantidad de vértices extremos para cada cluster, y como se observa se tiene un total de 49312 vértices extremos, con lo cual en comparación con el tamaño del 6D-EVM correspondiente a la animación original se obtiene una tasa de compresión de 90.33 aproximadamente.

Tabla 7.9: Tamaño de los 3D-EVMs generados por cada cluster para el video animado y 15 neuronas.

Cluster	Tamaño del 3D-EVM
0	4750
1	0
2	5802
3	4490
4	6830
5	5250
6	0
7	168
8	6418
9	1090
10	16
11	3404
12	6460
13	2010
14	2624
Total	49312

En la figura 7.64 se muestra de manera gráfica la cantidad de sub-animaciones que cuenta cada uno de los clusters. Con ello se determina que los clusters que tienen información más relevante de la escena son los clusters 0, 2, 8 y 13. Se eligieron estos clusters con base en su contenido, ya que hay otros clusters con una cantidad considerable de información, pero para fines de proporcionar al lector una idea general del resultado de agrupamiento se eligieron éstos.

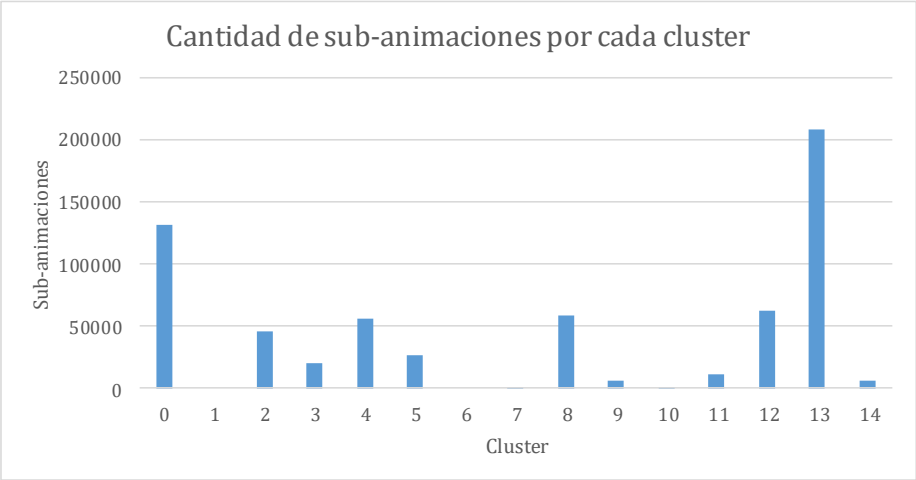


Figura 7.64: Gráfica con la cantidad de sub-animaciones agrupadas en cada cluster para el video animado y 15 neuronas.

Con base en lo anterior, ahora se describen los clusters seleccionados:

- El cluster 0 del cual se muestra su secuencia de frames en la figura 7.65, se observa que se

agrupan regiones donde se encuentran las sombras de la escena o regiones oscuras. Con lo anterior se obtiene el 3D-EVM que se muestra en la figura 7.66 en donde se observan las regiones de las sombras y su correspondiente desplazamiento en el tiempo.

- Para el cluster 2 se muestra su secuencia de frames en la figura 7.67, en donde se observa que se tiene agrupadas las regiones asociadas a los elementos más sobresalientes de la escena, en este caso se tienen las regiones que están próximas a los bordes de éstos elementos. Con esto en la figura 7.68 se muestra el correspondiente 3D-EVM, en donde claramente se observan las regiones descritas anteriormente y su desplazamiento en el tiempo.
- El cluster 8 cuenta con la secuencia de frames que se muestra en la figura 7.69, en donde se observa que se tienen agrupadas las regiones específicas donde se encuentran los bordes de los objetos y el bebé dentro de la escena. Este resultado ya se ha observado en configuraciones previas, ya que las regiones de los bordes presentan una forma similar debido al cambio de color en ellas.

  Para este cluster, en la figura 7.70 se muestra su correspondiente 3D-EVM, en donde claramente se observan los bordes descritos anteriormente y también se observa su desplazamiento en el tiempo.
- Por último, en la figura 7.71 se presenta la secuencia de frames correspondiente al cluster 13, en donde se observa que se tiene agrupadas las regiones de las sombras u oscuras. Como se ha mencionado anteriormente, este resultado es importante en el sentido que permite ubicar las zonas donde las sombras son más oscuras, que de manera general es justo debajo de los objetos.

  En la figura 7.72 se muestra el correspondiente 3D-EVM para el cluster 13, y como se observa se tienen las regiones mencionadas anteriormente y también su desplazamiento en el tiempo.

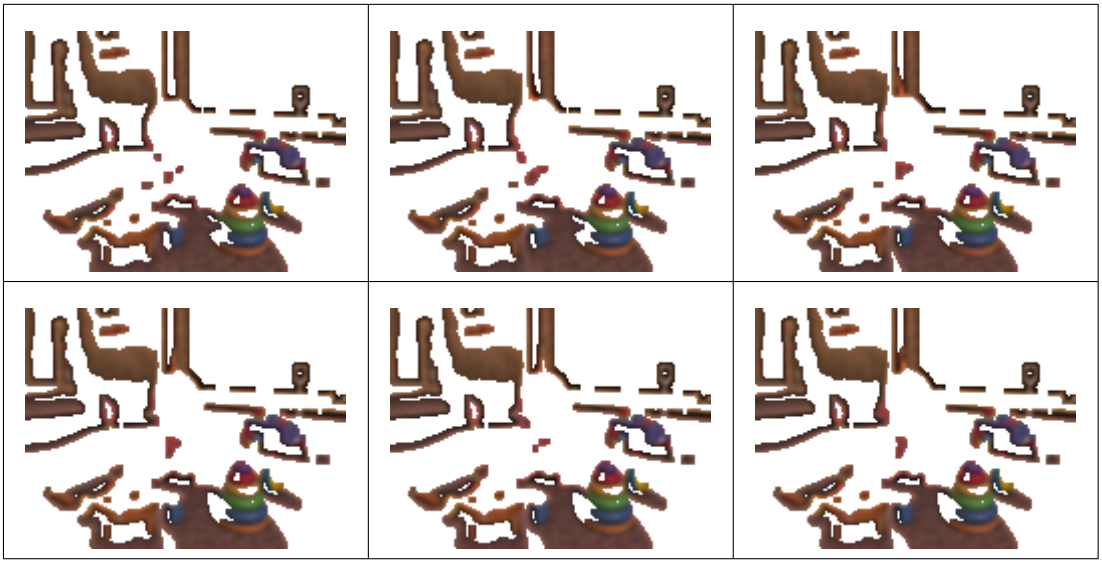


Figura 7.65: Video de animado y 15 neuronas, resultados de agrupamiento del cluster 0.

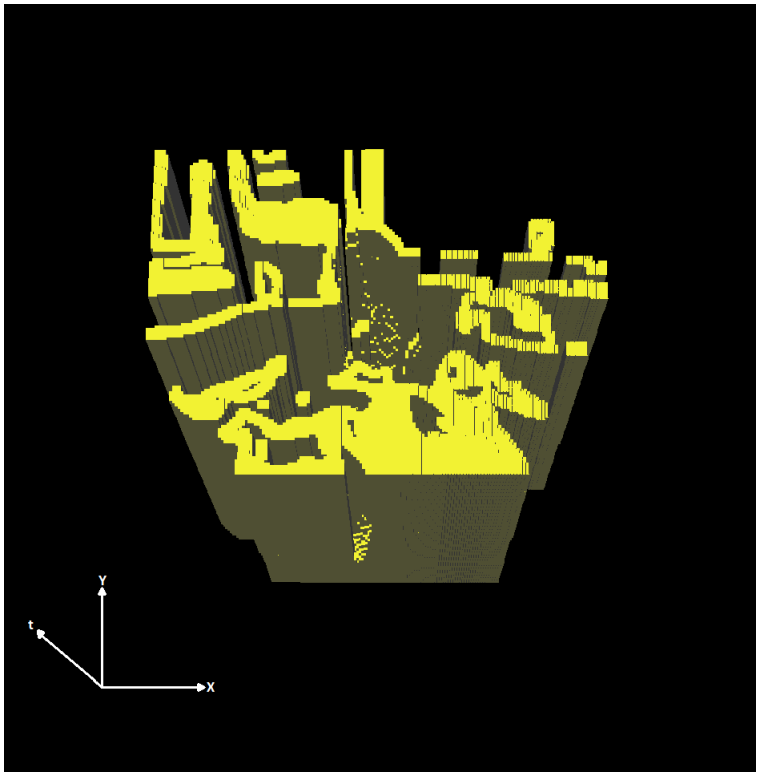


Figura 7.66: 3D-EVM del cluster 0 para el video animado y 15 neuronas.

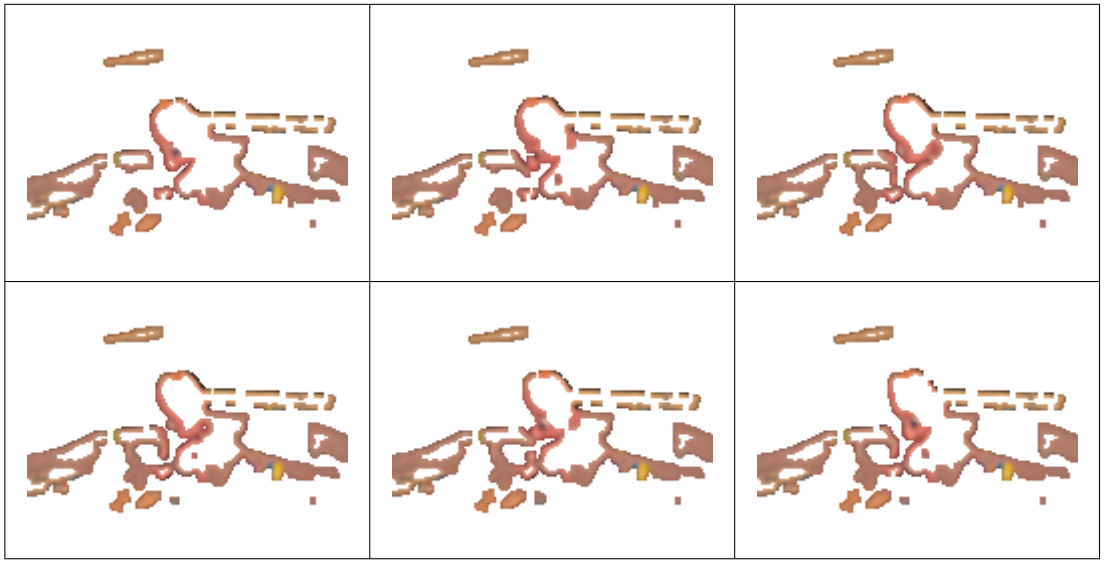


Figura 7.67: Video de animado y 15 neuronas, resultados de agrupamiento del cluster 2.

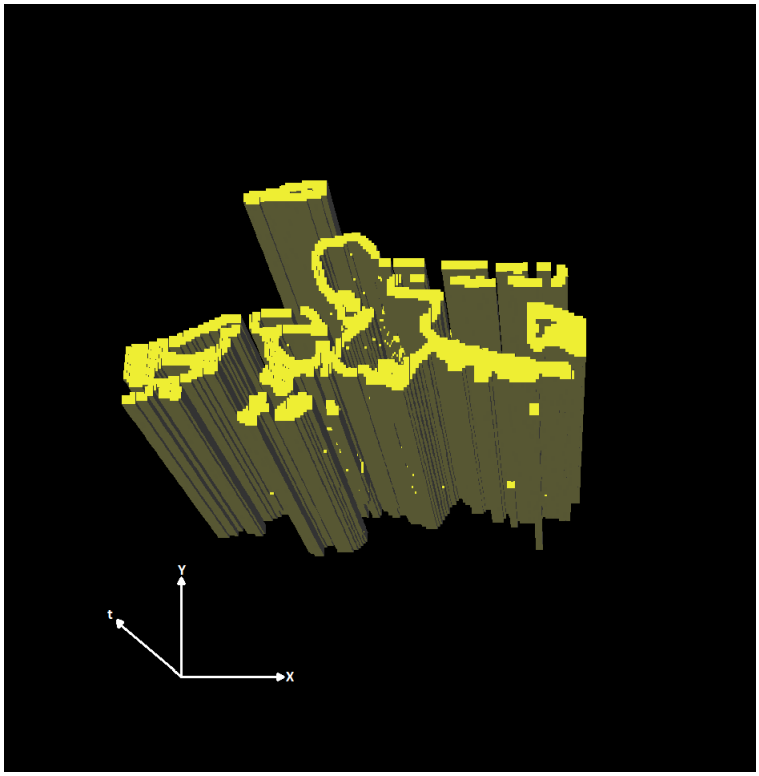


Figura 7.68: 3D-EVM del cluster 2 para el video animado y 15 neuronas.

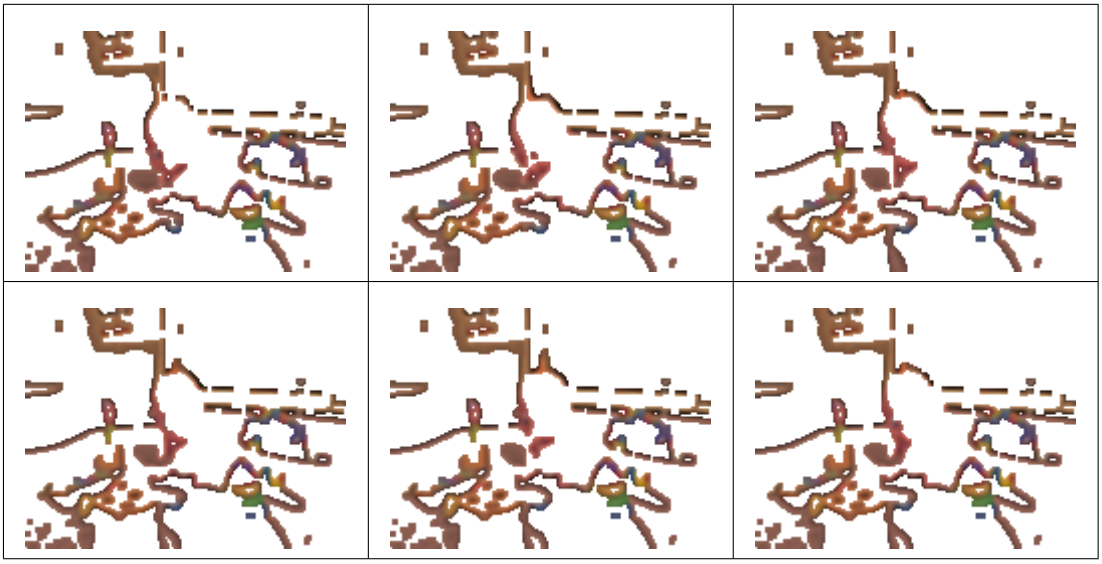


Figura 7.69: Video de animado y 15 neuronas, resultados de agrupamiento del cluster 8.

Figura 7.70: 3D-EVM del cluster 8 para el video animado y 15 neuronas.

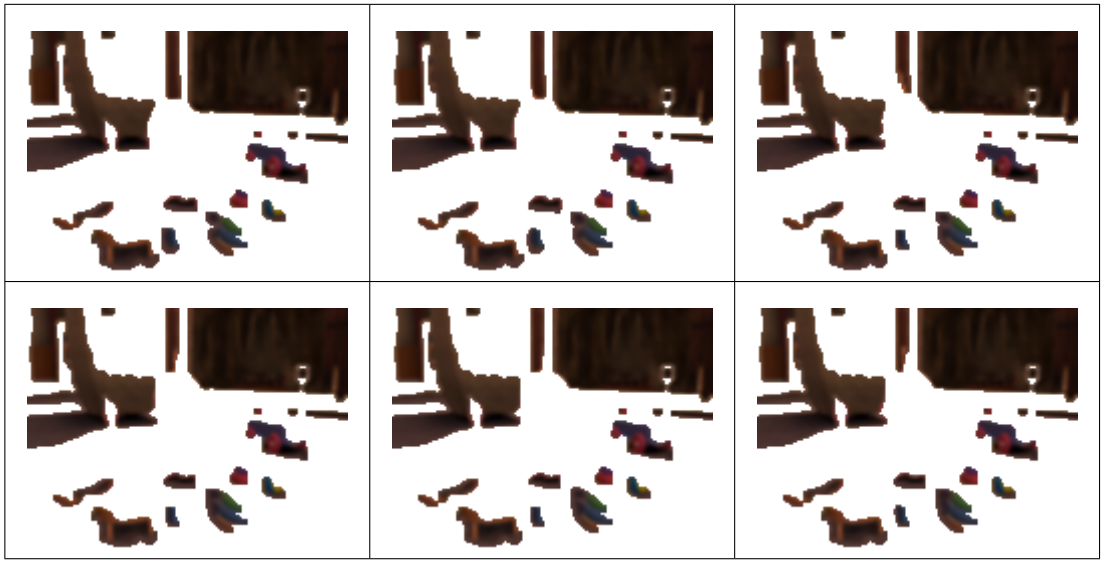


Figura 7.71: Video de animado y 15 neuronas, resultados de agrupamiento del cluster 13.

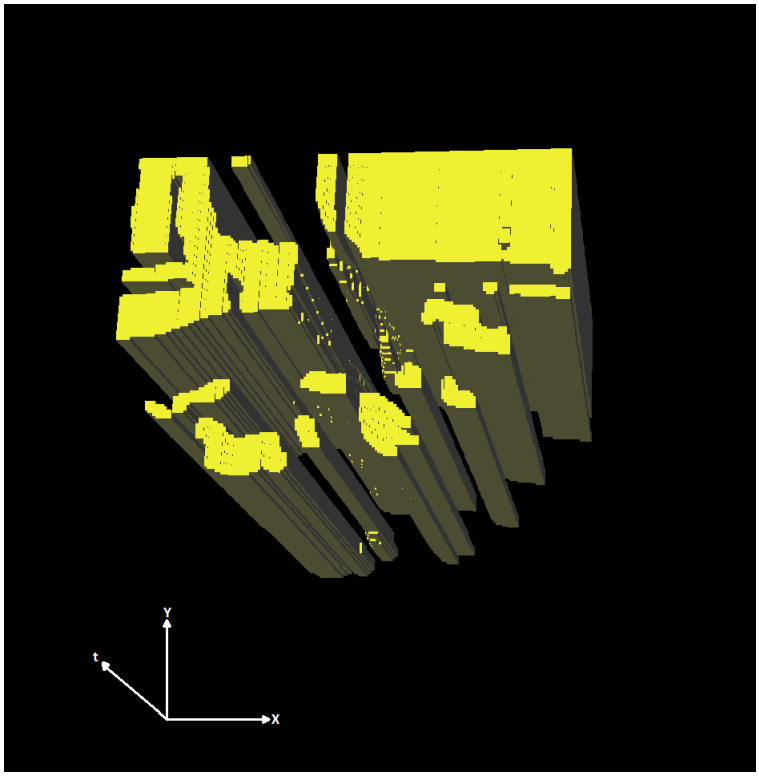


Figura 7.72: 3D-EVM del cluster 13 para el video animado y 15 neuronas.

7.4 Pruebas con un Video de Billar

En este apartado se describen las pruebas realizadas con un video de un jugador profesional de billar [54], este video consiste en una captura de una jugada, en el cual se observa su secuencia de frames en la figura 7.73. Este video consiste en una mesa de billar, tres bolas de billar y un jugador del cual solo se observa su taco (de billar) para pegarle a las bolas.

Figura 7.73: Algunos frames del video de billar.

Los frames de este video son de tamaño 240x160 y se tiene una cantidad de frames de 48, con lo cual se obtuvo que la representación en el 6D-EVM tiene un tamaño de 22076702 vertices extremos, ello considerando que el video tiene un esquema de color RGB.

7.4.1. Pruebas con 5 neuronas

En este apartado se presentan las pruebas de segmentación realizadas mediante un SOM y 5 neuronas. En la tabla 7.10 se muestran los tamaños de cada uno de los 3D-EVMs de los clusters. Como se observa en total se tienen 37588 vértices extremos, lo cual en comparación con el tamaño del 6D-EVM que representa la animación original, se tiene una tasa de compresión de 587.33 aproximadamente. Con ello se obtiene una compactación significativa de la animación original.

Tabla 7.10: Tamaño de los 3D-EVMs generados por cada cluster para el video de billar y 5 neuronas.

Cluster	Tamaño del 3D-EVM
0	6358
1	13550
2	1062
3	13178
4	3440
Total	37588

En la figura 7.74 se presenta de manera gráfica la cantidad de sub-animaciones que contiene cada cluster. Es importante notar que debido a que se tiene una escena donde el fondo es prácticamente estático, y que el evento principal de la escena consiste en el movimiento de las bolas de billar, algunos clusters contienen información sobre zonas con color uniforme, tal como la superficie de la mesa. Desde nuestro criterio, estos clusters no son relevantes para el evento de la escena por lo que no se presenta su contenido, y en cambio se analizará el contenido de los clusters 0, 1 y 4.

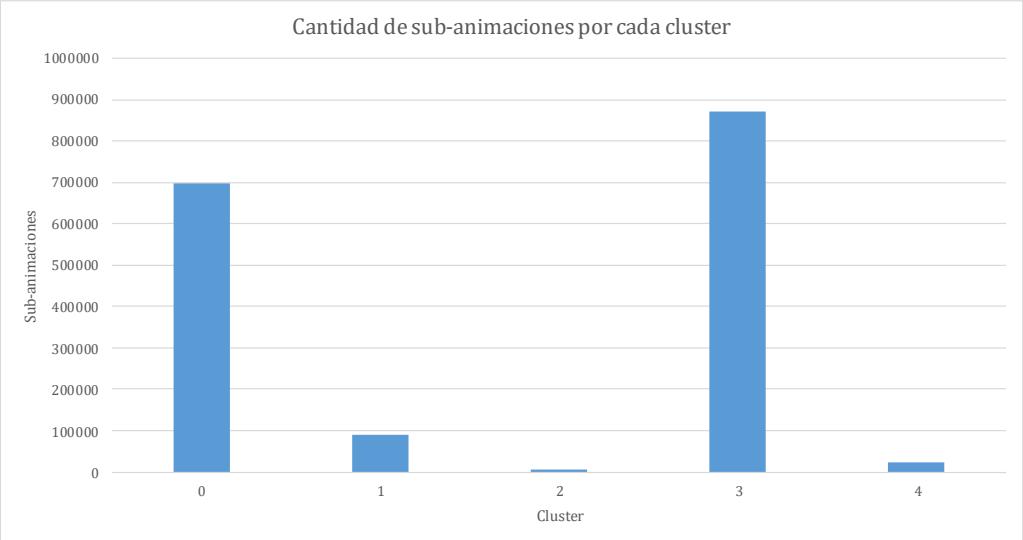


Figura 7.74: Gráfica con la cantidad de sub-animaciones agrupadas por cada cluster para el video de billar y 5 neuronas.

Con base en lo anterior, a continuación se describen los clusters seleccionados:

- El cluster 0 que contiene la secuencia de frames que se muestra en la figura 7.75, contiene regiones donde hay sombras o regiones oscuras. En estos frames se observa que también se encuentran las sombras de las bolas de billar. No obstante, este cluster no tiene mucha relevancia en relación al evento que se presenta en la escena. El 3D-EVM correspondiente a este cluster se presenta en la figura 7.76.
- Para el cluster 1 del cual se muestra su secuencia de frames en la figura 7.77, se observa que

se agrupan regiones relacionadas con el movimiento de las bolas de billar. Este resultado es importante en el sentido que el SOM es capaz de identificar patrones de movimiento y aislarlos de su entorno. Para este cluster, en la figura 7.78 se muestra su correspondiente 3D-EVM, en donde se puede observar que en la parte central se presenta el movimiento de las bolas de billar.

- Por último, para el cluster 4 se tiene la secuencia de frames de la figura 7.79, en donde se observa que se agrupan regiones que específicamente pertenecen al movimiento de las bolas de billar, lo cual es relevante en el sentido que se aísla de mejor manera dicho movimiento, incluso se presenta el movimiento de manera menos ruidosa que como en el caso del cluster 1. En la figura 7.80 se observa el correspondiente 3D-EVM en donde se observa claramente el desplazamiento de las bolas de billar.

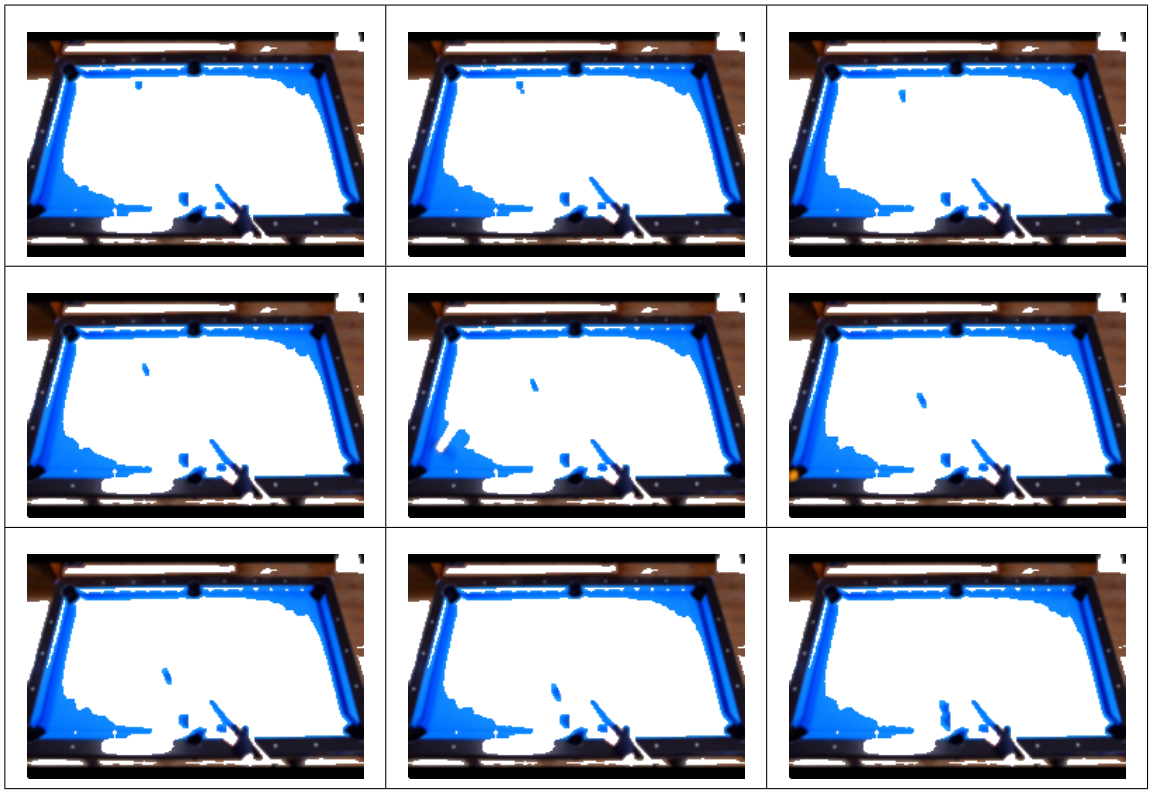


Figura 7.75: Video de billar y 5 neuronas, resultados de agrupamiento del cluster 0.

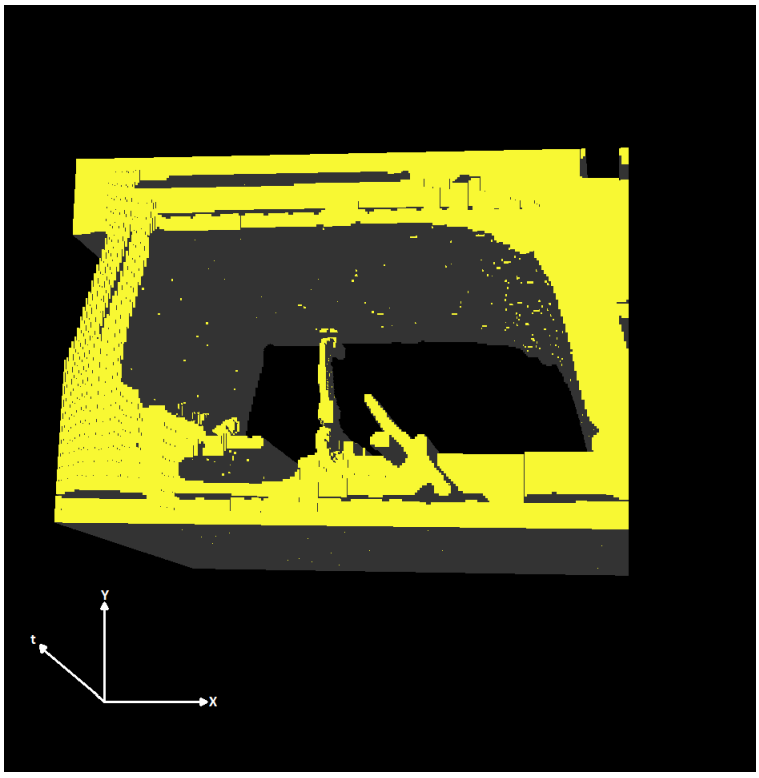


Figura 7.76: 3D-EVM del cluster 0 para el video de billar y 5 neuronas.

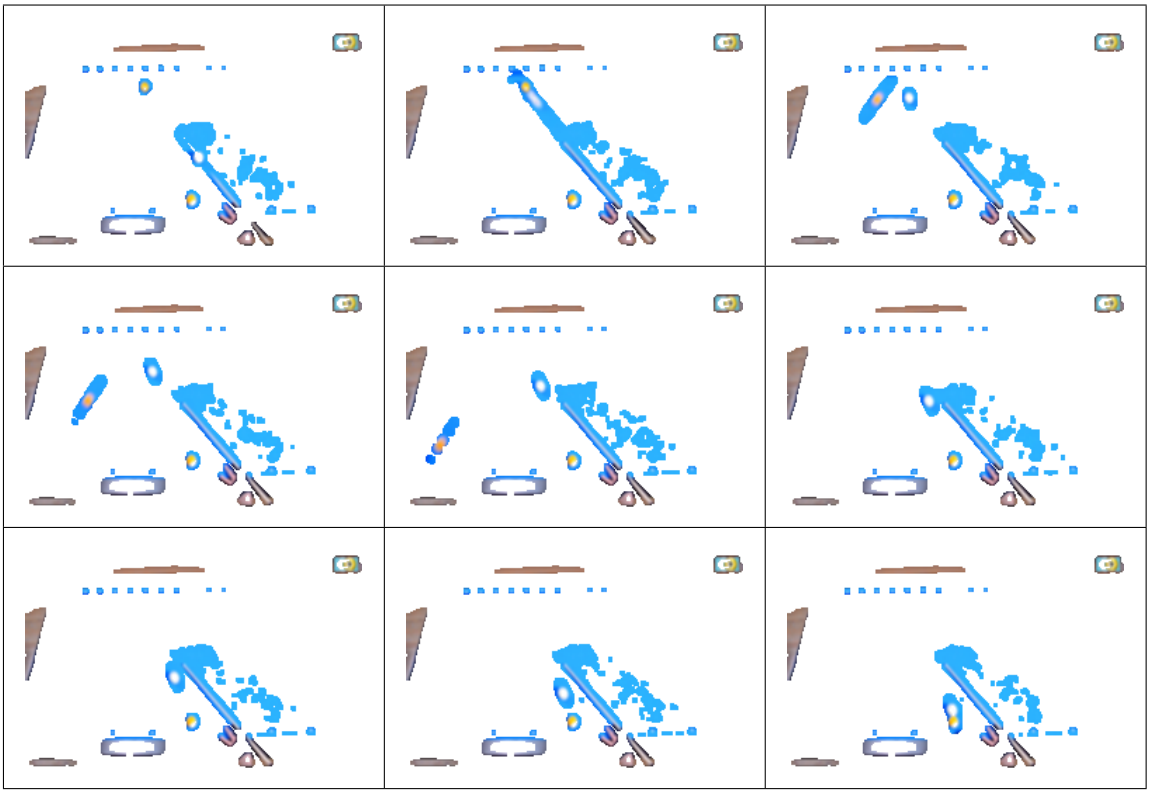


Figura 7.77: Video de billar y 5 neuronas, resultados de agrupamiento del cluster 1.

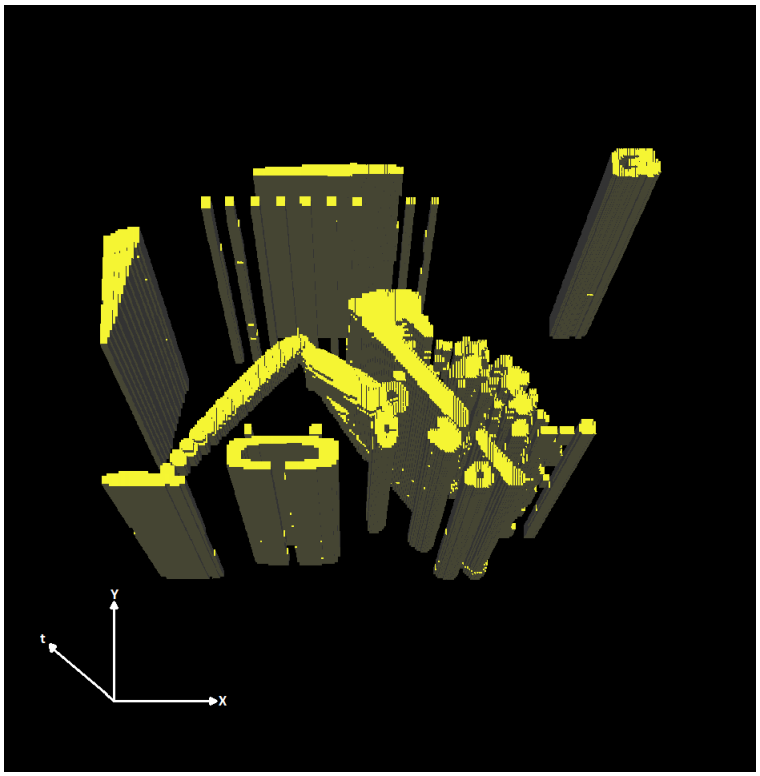


Figura 7.78: 3D-EVM del cluster 1 para el video de billar y 5 neuronas.

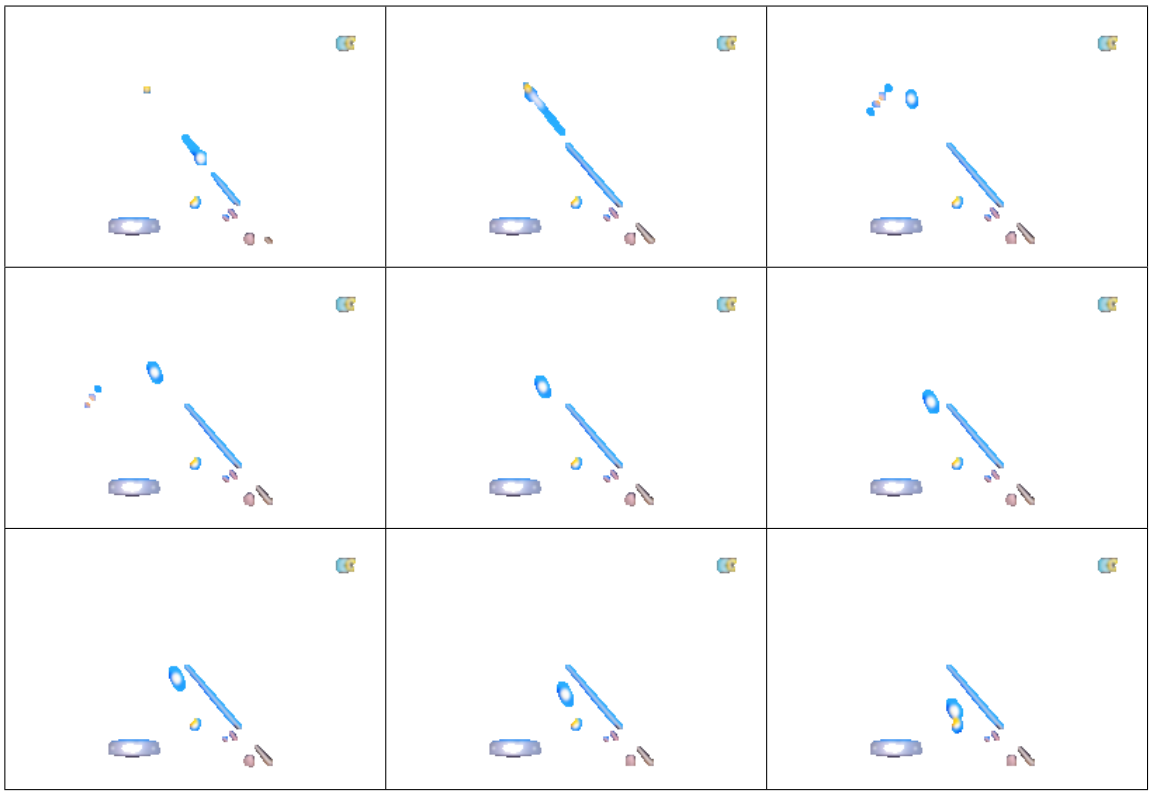


Figura 7.79: Video de billar y 5 neuronas, resultados de agrupamiento del cluster 4.

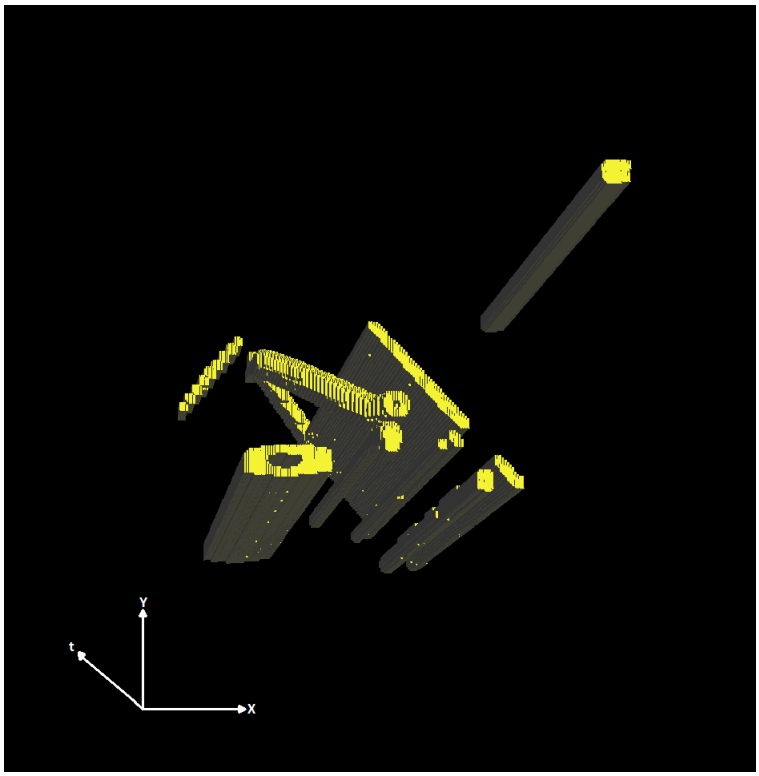


Figura 7.80: 3D-EVM del cluster 4 para el video de billar y 5 neuronas.

7.4.2. Pruebas con 10 neuronas

En este apartado se analizan los resultados de segmentación mediante un SOM y 10 neuronas. En la tabla 7.11 se muestra el tamaño de cada uno de los 3D-EVMs generados para cada cluster, en donde se observa que se tiene un total de 76754 vértices extremos, lo cual en comparación con el tamaño del 6D-EVM de la animación original presenta una tasa de compresión de 287.62 aproximadamente.

Tabla 7.11: Tamaño de los 3D-EVMs generados por cada cluster para el video de billar y 10 neuronas.

Cluster	Tamaño del 3D-EVM
0	2352
1	10364
2	13604
3	866
4	4434
5	4552
6	6614
7	18520
8	14194
9	1254
Total	76754

En la figura 7.81 se muestra de manera gráfica la cantidad de sub-animaciones que contiene cada uno de los clusters. En esta gráfica se observa que algunos clusters contienen una cantidad considerable de información, sin embargo eso no implica que sean clusters relevantes para el evento del video, por ello se han seleccionado los clusters a analizar con base en su relación con la escena o aspectos relevantes. Los clusters seleccionados para ser presentados y analizados son los clusters 1, 4 y 6.

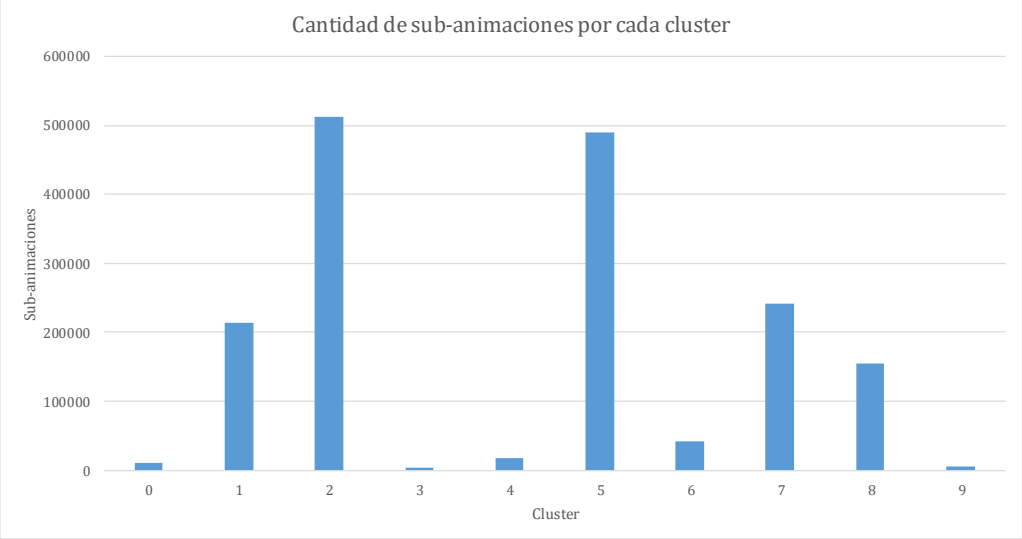


Figura 7.81: Gráfica con la cantidad de sub-animaciones agrupadas por cada cluster para el video de billar y 10 neuronas.

Con base en lo anterior, a continuación se describen los clusters seleccionados:

- El cluster 1 que tiene la secuencia de frames que se muestra en la figura 7.82, contiene las regiones donde se presentan los bordes de los elementos de la escena. Nuevamente, el agrupamiento se realiza con base en que estas regiones presentan una forma similar, ya que

hay un cambio en el color. En la figura 7.83 se muestra el 3D-EVM generado en el cluster 1, en donde claramente se observan los bordes mencionados anteriormente.

- El cluster 4 cuenta con la secuencia de frames que se muestra en la figura 7.84, en donde se observa que se tienen agrupadas las regiones asociadas al movimiento de las bolas de billar. En este caso, nuevamente el SOM logra identificar los patrones de movimiento que se presentan en la escena, lo cual es muy relevante para realizar un post-procesamiento. El 3D-EVM correspondiente a este cluster se muestra en la figura 7.85, en el cual claramente se puede notar que en la parte central se encuentran las regiones del desplazamiento de las bolas de billar.
- El cluster 6 del cual se muestra su secuencia de frames en la figura 7.86, como se observa contiene regiones asociadas al movimiento de las bolas de billar. A diferencia que en el caso del cluster 4, en este cluster también se agrupan regiones con color uniforme o sobresaliente, pero en sí también captura la dinámica de la escena. Con esto, en la figura 7.87 se muestra el 3D-EVM correspondiente, en donde se observa que en la parte central se encuentran las regiones de las bolas de billar y su movimiento con respecto al espacio 2D y al tiempo.

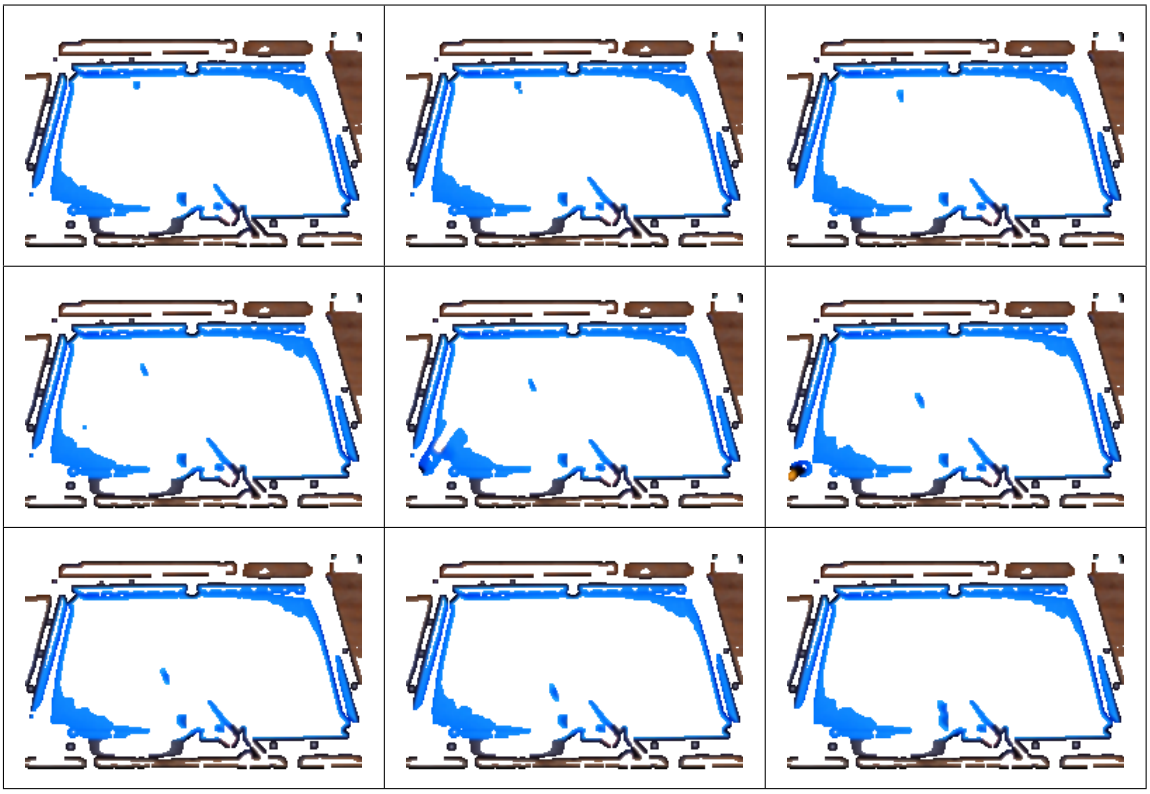


Figura 7.82: Video de billar y 10 neuronas, resultados de agrupamiento del cluster 1.

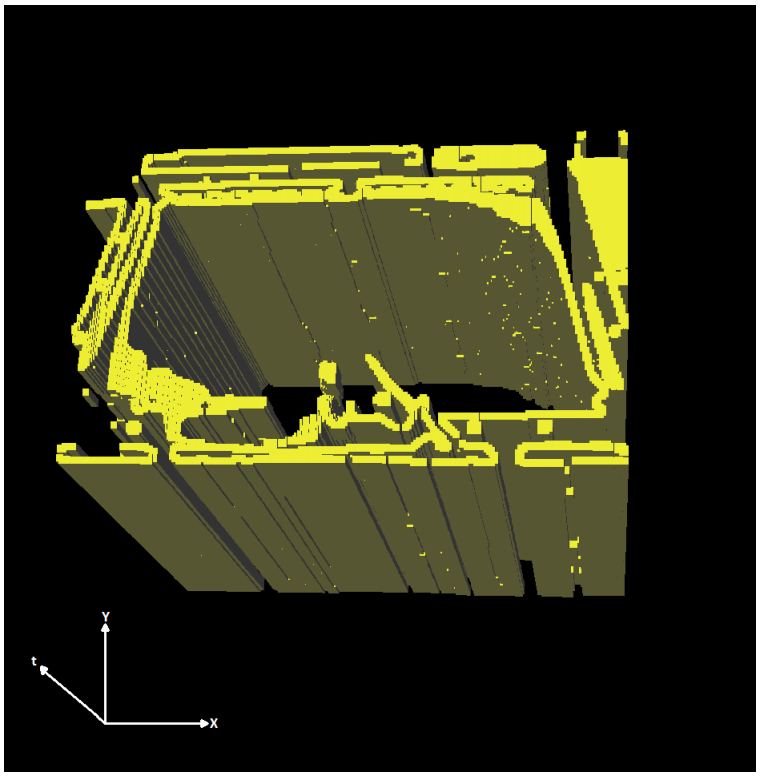


Figura 7.83: 3D-EVM del cluster 1 para el video de billar y 10 neuronas.

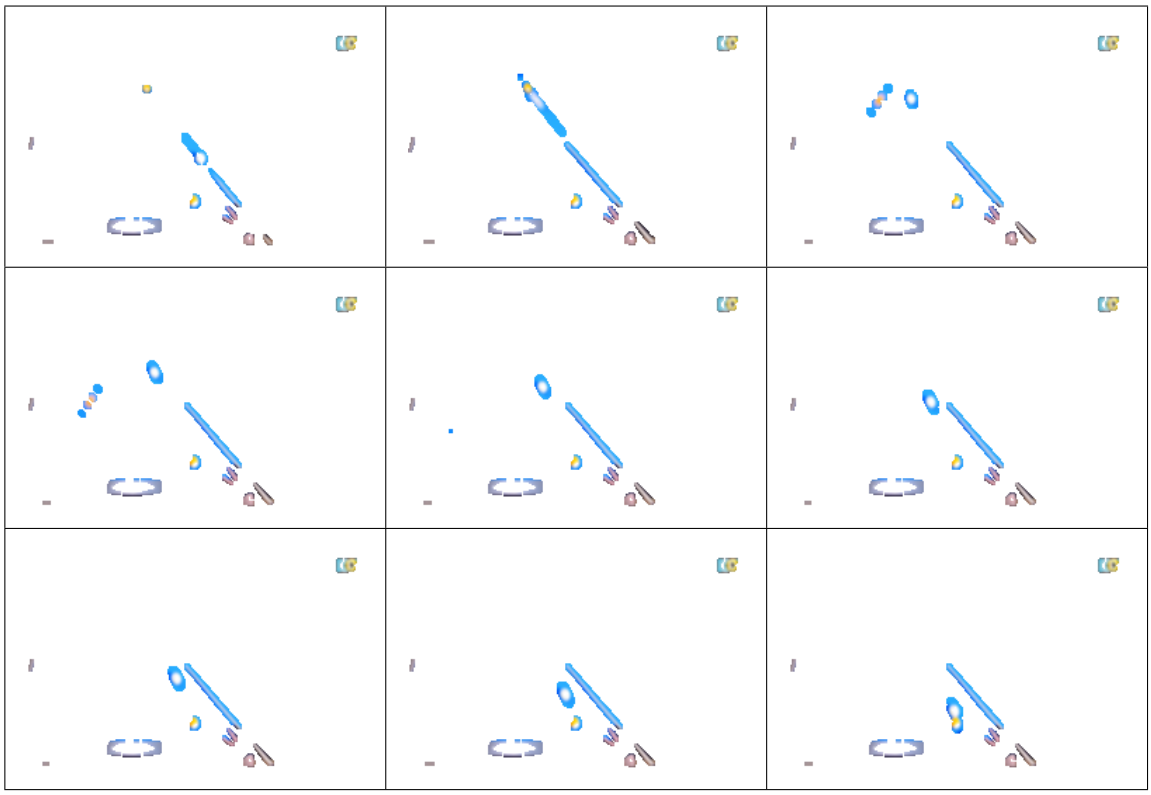


Figura 7.84: Video de billar y 10 neuronas, resultados de agrupamiento del cluster 4.

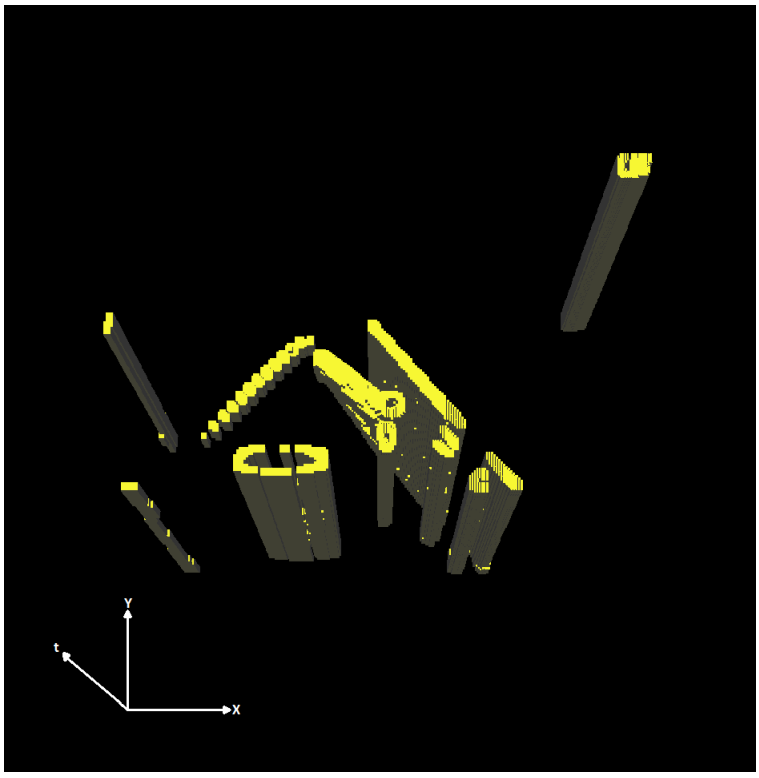


Figura 7.85: 3D-EVM del cluster 4 para el video de billar y 10 neuronas.

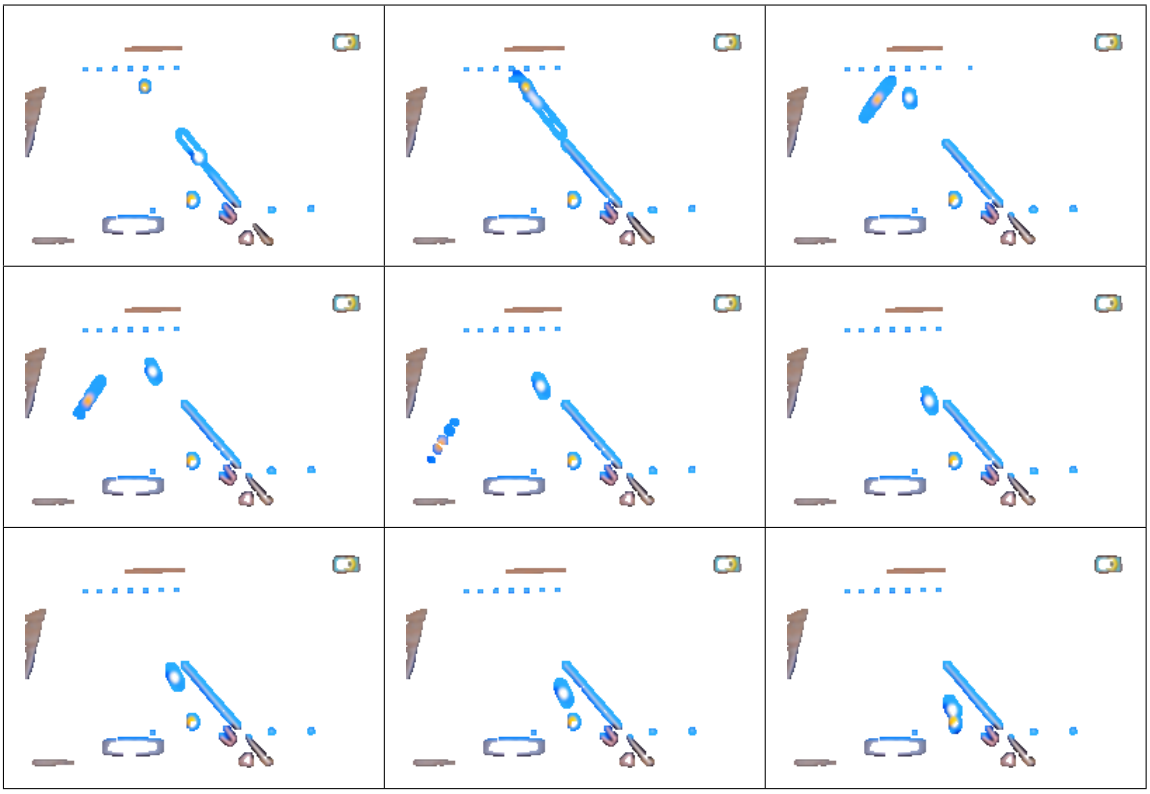


Figura 7.86: Video de billar y 10 neuronas, resultados de agrupamiento del cluster 6.

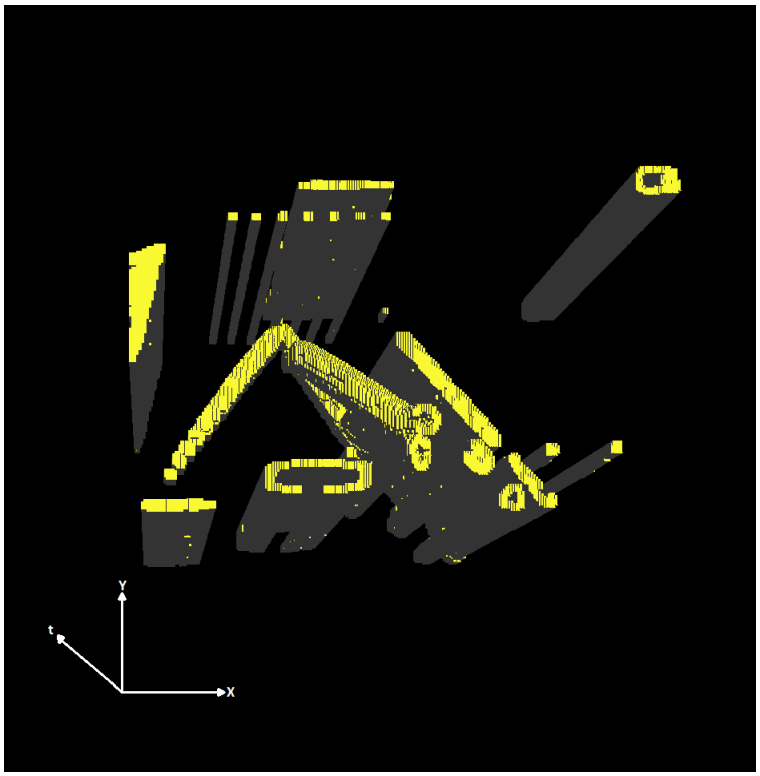


Figura 7.87: 3D-EVM del cluster 6 para el video de billar y 10 neuronas.

7.4.3. Pruebas con 15 neuronas

En este apartado se muestran los resultados obtenidos mediante aplicar el proceso de segmentación con un SOM y 15 neuronas. En la tabla 7.12 se muestran los tamaños de los 3D-EVMs de los clusters obtenidos, en donde se tiene un total de 121204 vértices extremos, lo cual comparado con el tamaño del 6D-EVM que representa la animación original, se tiene una tasa de compresión de 182.14 aproximadamente.

Tabla 7.12: Tamaño de los 3D-EVMs generados por cada cluster para el video de billar y 15 neuronas.

Cluster	Tamaño del 3D-EVM
0	17960
1	10666
2	6072
3	4700
4	1706
5	7782
6	634
7	13838
8	1338
9	4064
10	15570
11	2798
12	966
13	12054
14	21056
Total	121204

Ahora, en la figura 7.88 se muestra de manera gráfica la cantidad de sub-animaciones que contiene cada uno de los clusters, en donde nuevamente se observa que algunos clusters contienen una cantidad considerable de información, lo cual no implica que sean relevantes al evento de la escena. Por ello se han seleccionado los clusters a analizar con base en su relevancia, y los clusters seleccionados son el cluster 1, 2 y 9.

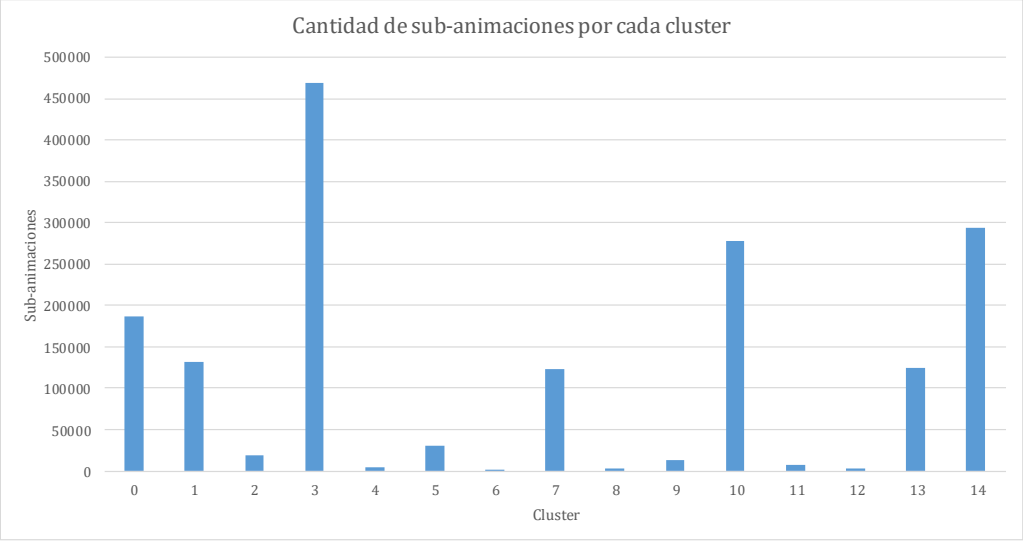


Figura 7.88: Gráfica con la cantidad de sub-animaciones agrupadas por cada cluster para el video de billar y 15 neuronas.

Con base en lo anterior, ahora se describen los clusters seleccionados:

- El cluster 1 cuenta con la secuencia de frames que se muestra en la figura 7.89, en donde se observa que se tienen agrupadas las regiones donde hay bordes. Este caso ya se ha presentado en configuraciones anteriores, y como se ha descrito estas regiones tienen una forma similar. El 3D-EVM correspondiente se muestra en la figura 7.90.
- Para los clusters 2 y 9, se presenta nuevamente el caso en donde se agrupa las regiones relacionadas con el movimiento de las bolas de billar. Es importante considerar que ambos clusters contiene información similar pero el cluster 2 contiene información sobre el entorno, y específicamente las regiones más sobresalientes de la escena. Las secuencias de frames correspondientes a los clusters se muestran en las figuras 7.91 y 7.93, y sus respectivos 3D-EVMs se muestran en las figuras 7.92 y 7.94.

  En los 3D-EVMs de los clusters se observa claramente que en la parte central se encuentran las regiones asociadas al movimiento de las bolas de billar. No obstante, el 3D-EVM del cluster 9 (figura 7.94) tiene una mejor separación de este evento del resto de la escena.

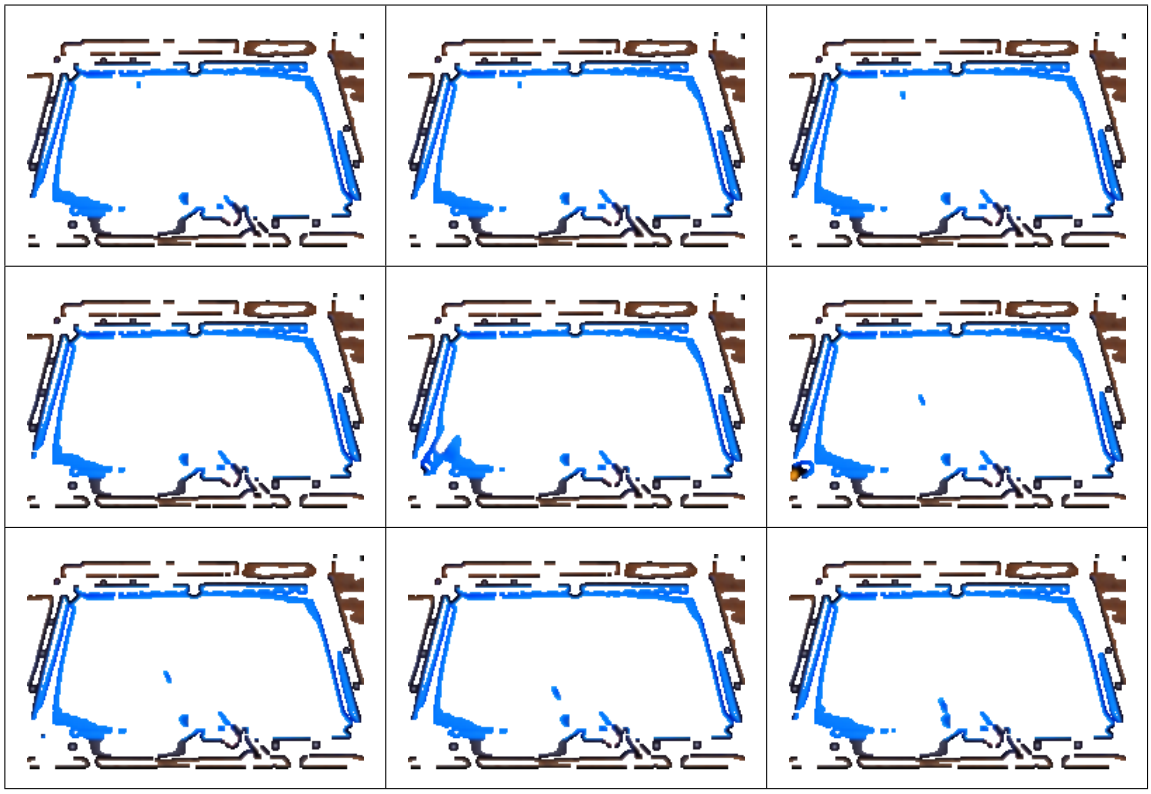


Figura 7.89: Video de billar y 15 neuronas, resultados de agrupamiento del cluster 1.

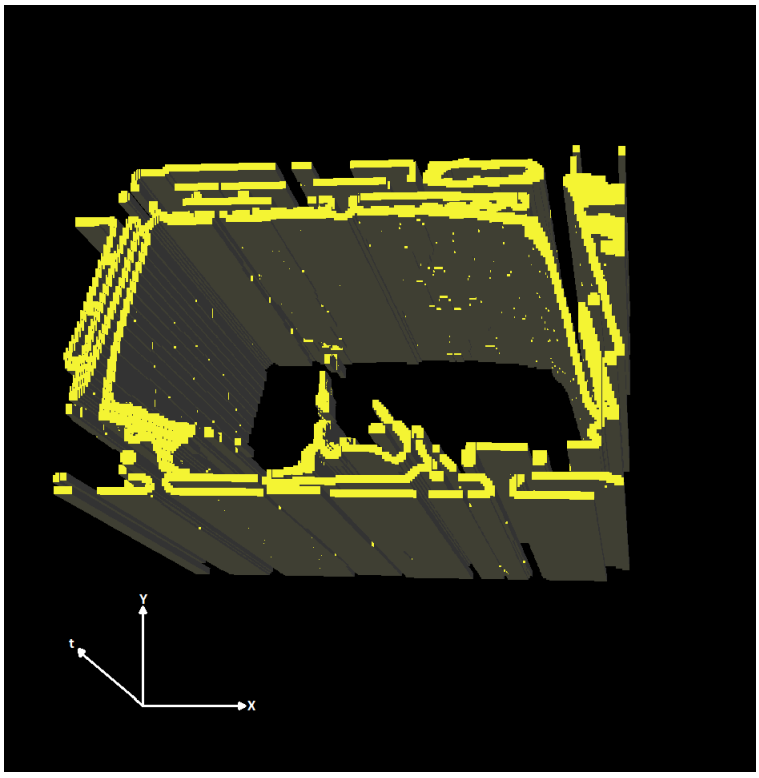


Figura 7.90: 3D-EVM del cluster 1 para el video de billar y 15 neuronas.

Figura 7.91: Video de billar y 15 neuronas, resultados de agrupamiento del cluster 2.

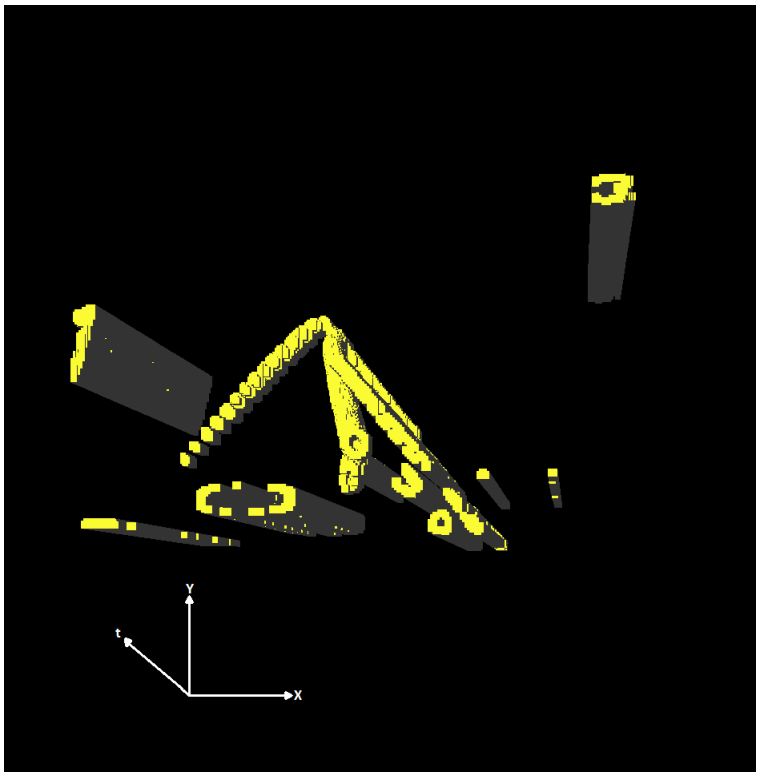


Figura 7.92: 3D-EVM del cluster 2 para el video de billar y 15 neuronas.

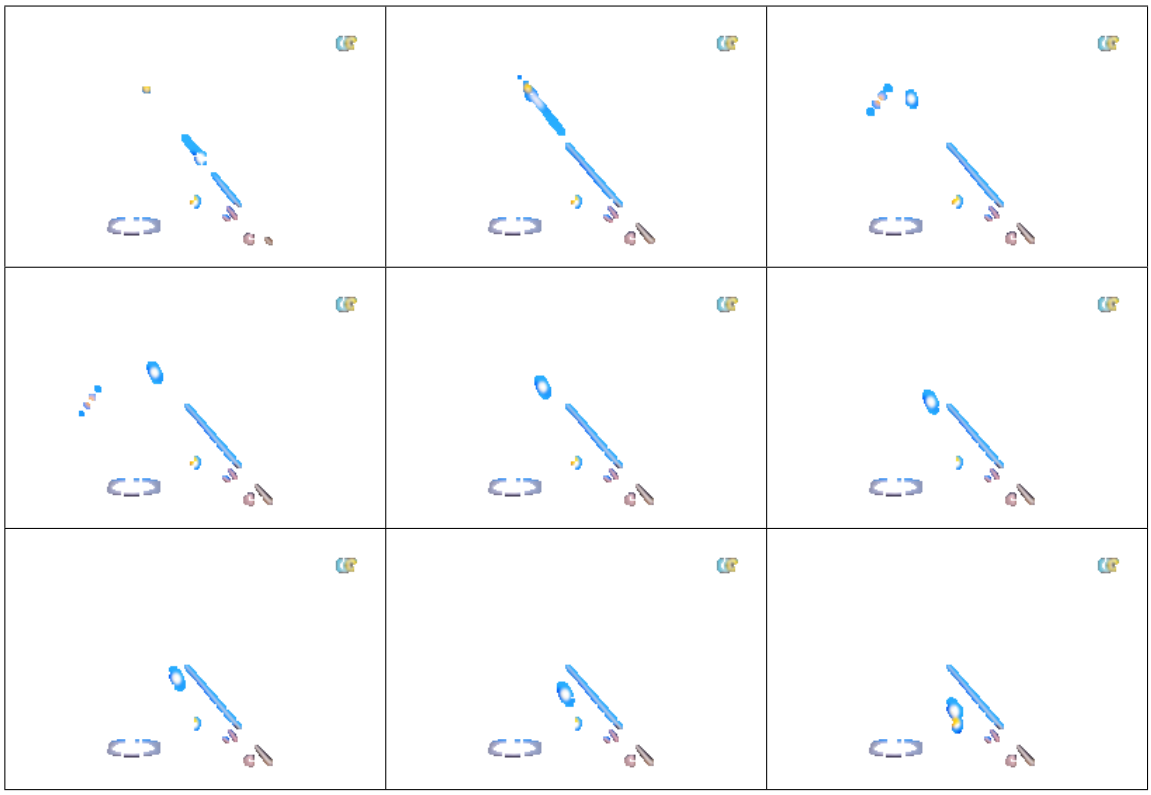


Figura 7.93: Video de billar y 15 neuronas, resultados de agrupamiento del cluster 9.

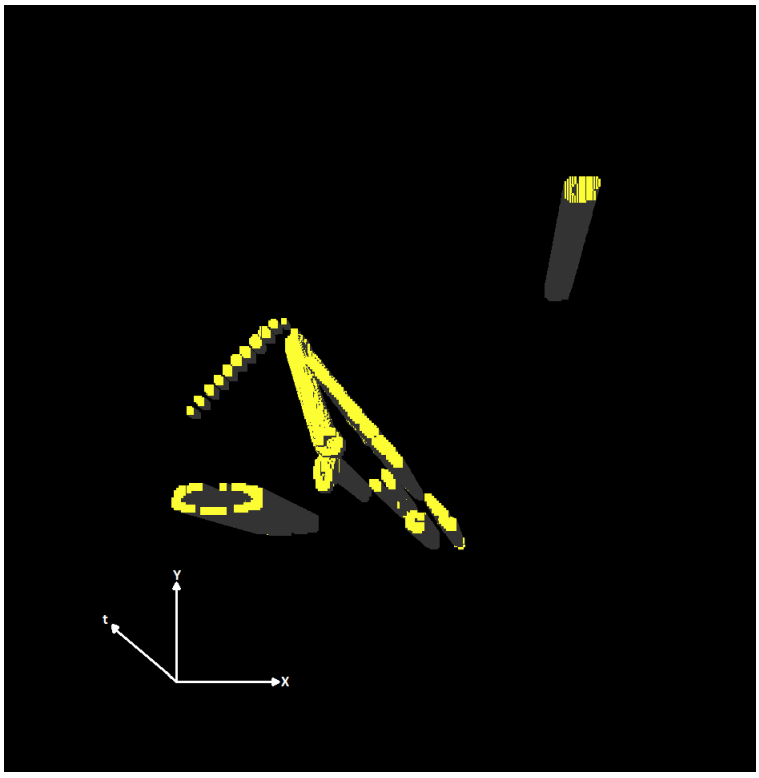


Figura 7.94: 3D-EVM del cluster 9 para el video de billar y 15 neuronas.

7.5 Pruebas con un Video de Vigilancia

En esta sección se describen los resultados de segmentación obtenidos de aplicar el framework desarrollado en este trabajo a un video de vigilancia, del cual se muestran algunos frames en la figura 7.95. Este video consiste en una cámara de vigilancia estática, la cual se utiliza para monitorear el paso de personas y autos sobre una calle, cabe mencionar que este video toma lugar en las instalaciones de la Universidad Tecnológica de la Mixteca (UTM), en la ciudad de Huajuapan de León, Oax. La escena utilizada en estas pruebas consiste en dos personas que caminan sobre el pavimento.

Figura 7.95: Algunos frames del video de vigilancia.

Como se observa en la figura 7.95, los frames del video tienen un esquema de color de escala de grises, con estas pruebas se valida que efectivamente el framework desarrollado, así como que la implementación del modelo nD-EVM en el lenguaje de programación C++ puede utilizarse para representar politopos de diversas dimensionalidades. Los frames son de tamaño 240x160 y se tiene una cantidad de frames de 54, con ello se obtuvo que la representación en el 4D-EVM tiene un tamaño de 2170802 vértices extremos. Es importante considerar que se se obtiene un 4D-EVM debido a que se necesitan 3 dimensiones para representar un frame, dos dimensiones para el plano 2D del frame y una dimensión para la profundidad de gris (x, y, g), y adicionalmente se requiere una dimensión para la variable temporal.

7.5.1. Pruebas con 5 neuronas

En este apartado se presentan los resultados obtenidos del proceso de segmentación mediante un SOM y 5 neuronas. En la tabla 7.13 se muestra la cantidad de vértices extremos por cada 3D-EVM generado en cada cluster, como se observa se tiene un total de 77062 vértices extremos, lo cual en comparación con el tamaño del 4D-EVM de la animación original se tiene una tasa de compresión de 28.16 aproximadamente. En este caso se obtiene una tasa de compresión menor en comparación con las pruebas de videos en RGB, lo cual se debe a que la animación original se representa mediante un 4D-EVM y no un 6D-EVM.

Tabla 7.13: Tamaño de los 3D-EVMs generados por cada cluster para el video de vigilancia y 5 neuronas.

Cluster	Tamaño del 3D-EVM
0	27056
1	21086
2	7640
3	18544
4	2736
Total	77062

En la figura 7.96 se muestra de manera gráfica la cantidad de sub-animaciones que agrupa cada cluster, en donde se observa que los clusters con una mayor cantidad de información son los clusters 0, 1, 2 y 3. Con base en lo anterior, estos clusters han sido seleccionados para analizar su contenido.

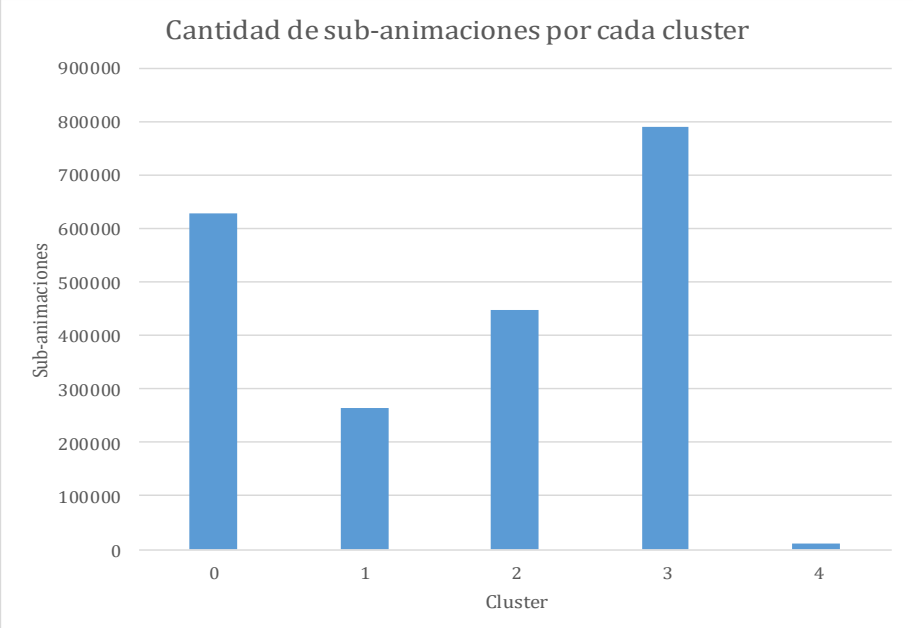


Figura 7.96: Gráfica con la cantidad de sub-animaciones agrupadas por cada cluster para el video de vigilancia y 5 neuronas.

Ahora se presenta una descripción del contenido de cada uno de los clusters seleccionados:

- En el cluster 0, del cual se muestra su secuencia de frames en la figura 7.97, se observa que se agrupan regiones claras del pavimento, además se observa el movimiento de las personas

sobre dichas regiones. En la figura 7.98 se muestra el 3D-EVM correspondiente al cluster 0, en donde se observa la evolución de las regiones mencionadas a través de la variable temporal.

- Para el cluster 1 se presenta su secuencia de frames en la figura 7.99, en donde se observa que se agrupan regiones oscuras de la escena, en particular regiones donde hay sombras y además hay tendencia hacia los bordes de los objetos. Por ejemplo se puede apreciar el borde de la carretera y la silueta de las personas. En la figura 7.100 se observa el 3D-EVM correspondiente al cluster 1, en donde se puede observar de manera clara el borde de la carretera.
- Para el cluster 2, se muestra su secuencia de frames en la figura 7.101, en donde se puede observar que se agrupan las regiones más oscuras de la escena, en particular la zona donde se encuentran los arbustos. Este resultado tiene sentido con base en que en esas zonas la forma de los politopos es similar ya que los valores de escala de gris son similares, también se pueden observar las siluetas de las personas y la zona del bosque está aislada del resto de la escena. En la figura 7.102 se muestra el correspondiente 3D-EVM del cluster 2, en donde se observa claramente la zona del bosque y su evolución con respecto al tiempo.
- Por último, para el cluster 3 se muestra su secuencia de frames en la figura 7.103, en donde se observa que se agrupan regiones del centro de la carretera. En este sentido, la zona del centro de la carretera es más clara que en las orillas, además también se agrupa el movimiento de las personas a través de estas regiones y como se puede observar que solo se agrupan las regiones más claras del cuerpo de las personas. En la figura 7.104 se muestra el 3D-EVM correspondiente al cluster 3, en donde claramente se observa la región del centro de la carretera y en la parte superior se tiene el movimiento de las personas.

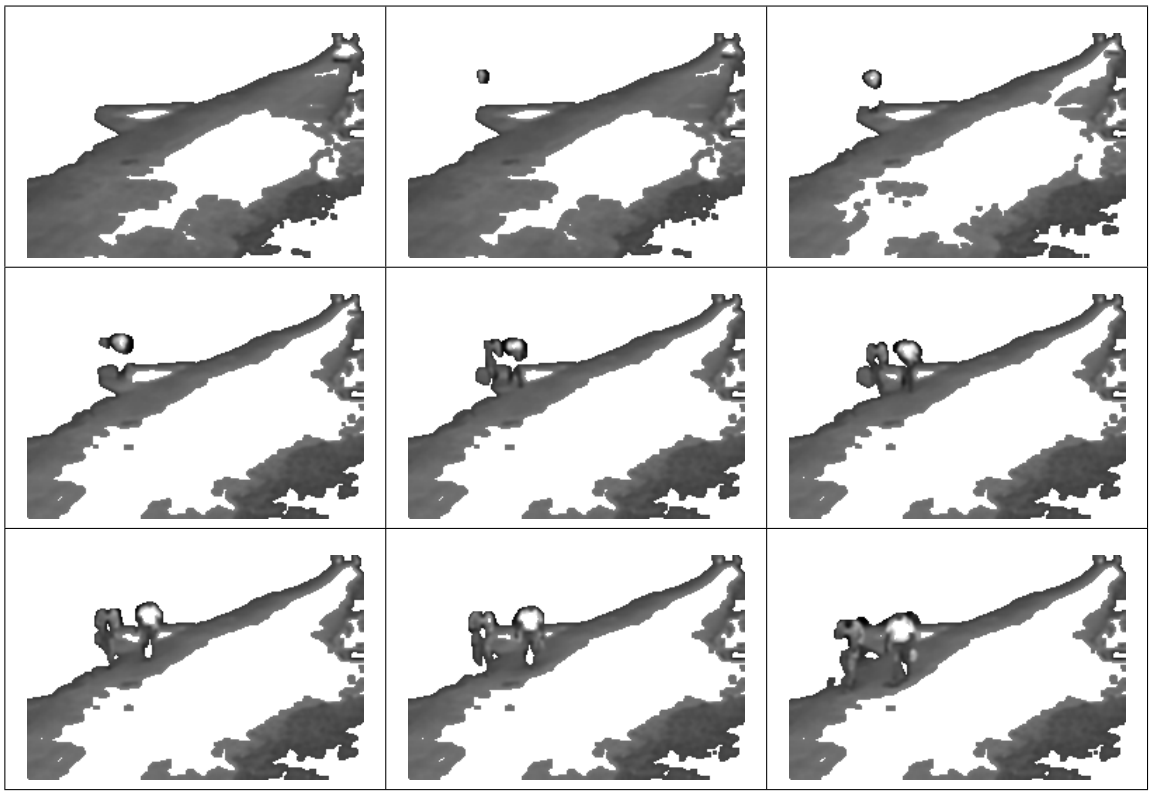


Figura 7.97: Video de vigilancia y 5 neuronas, resultados de agrupamiento del cluster 0.

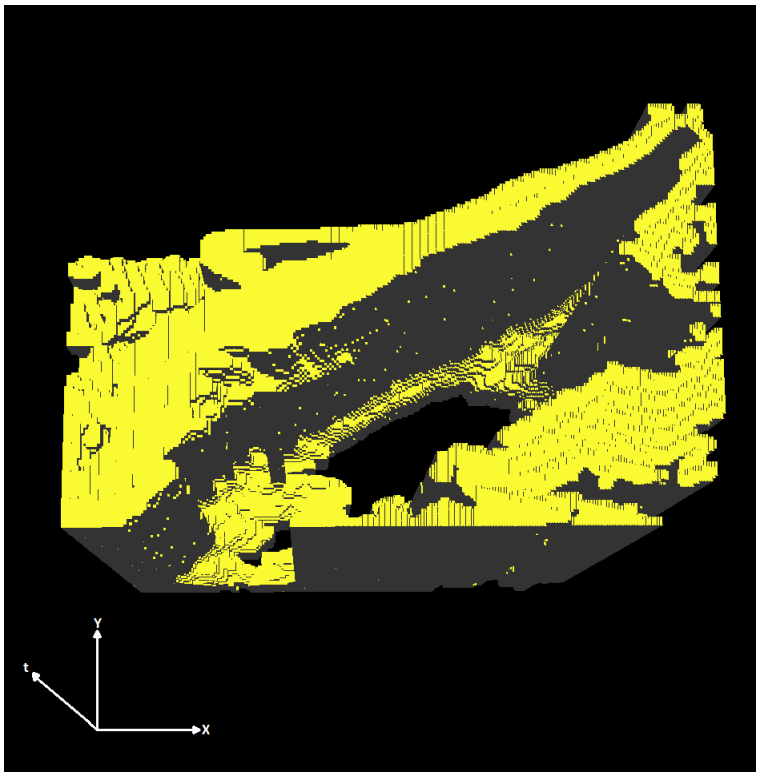


Figura 7.98: 3D-EVM del cluster 0 para el video de vigilancia y 5 neuronas.

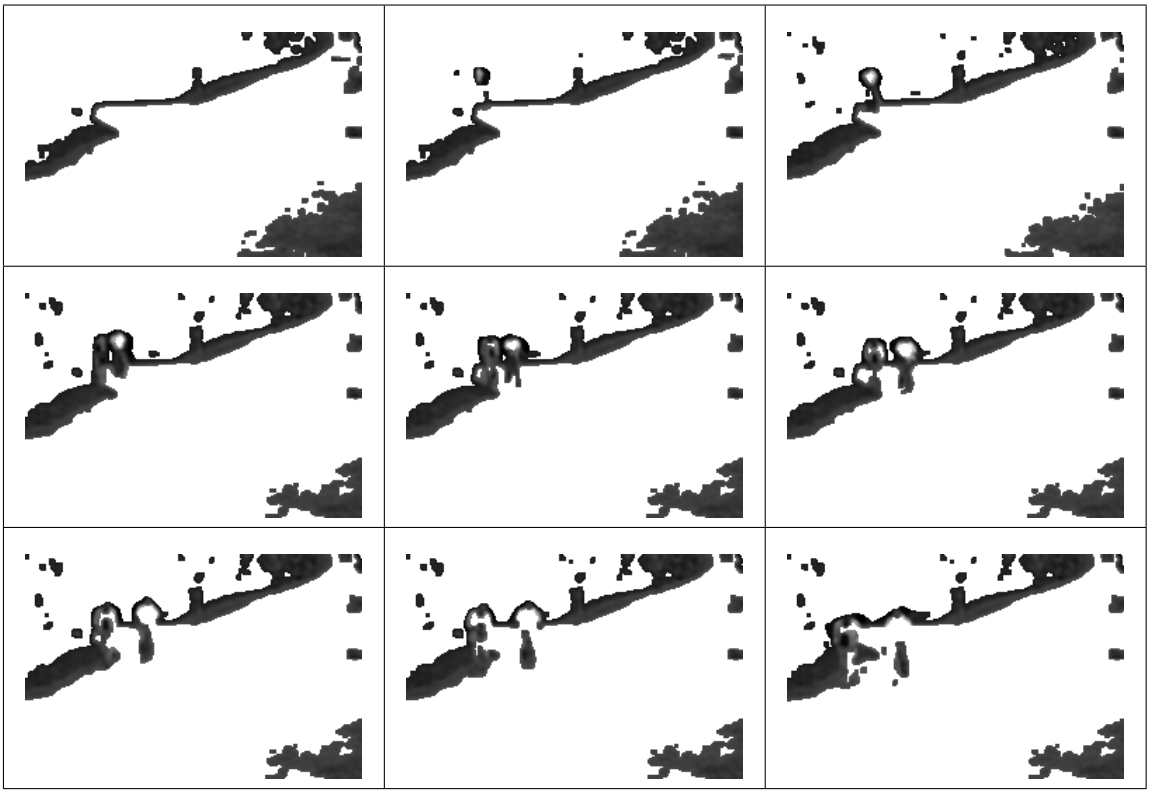


Figura 7.99: Video de vigilancia y 5 neuronas, resultados de agrupamiento del cluster 1.

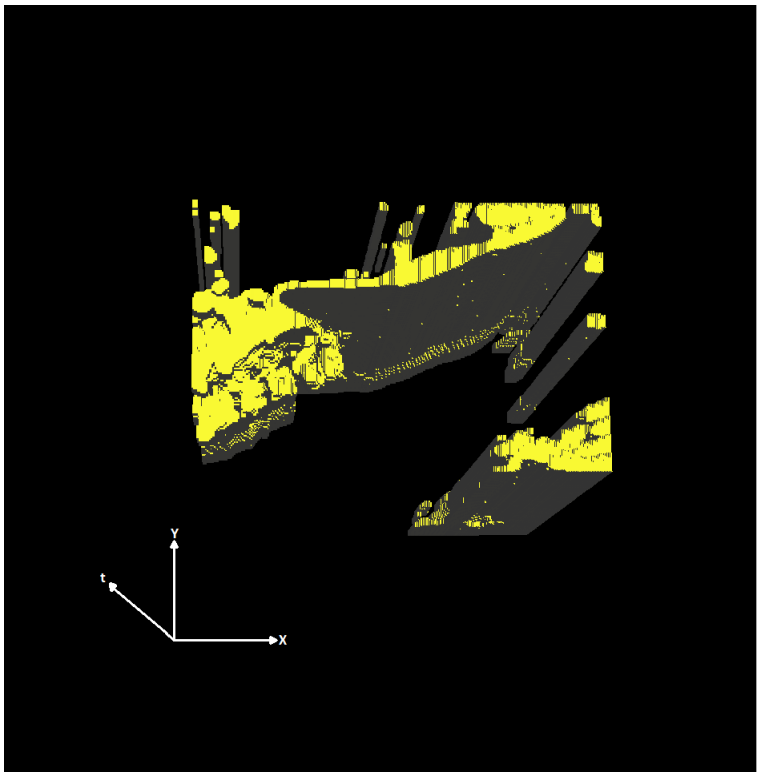


Figura 7.100: 3D-EVM del cluster 1 para el video de vigilancia y 5 neuronas.

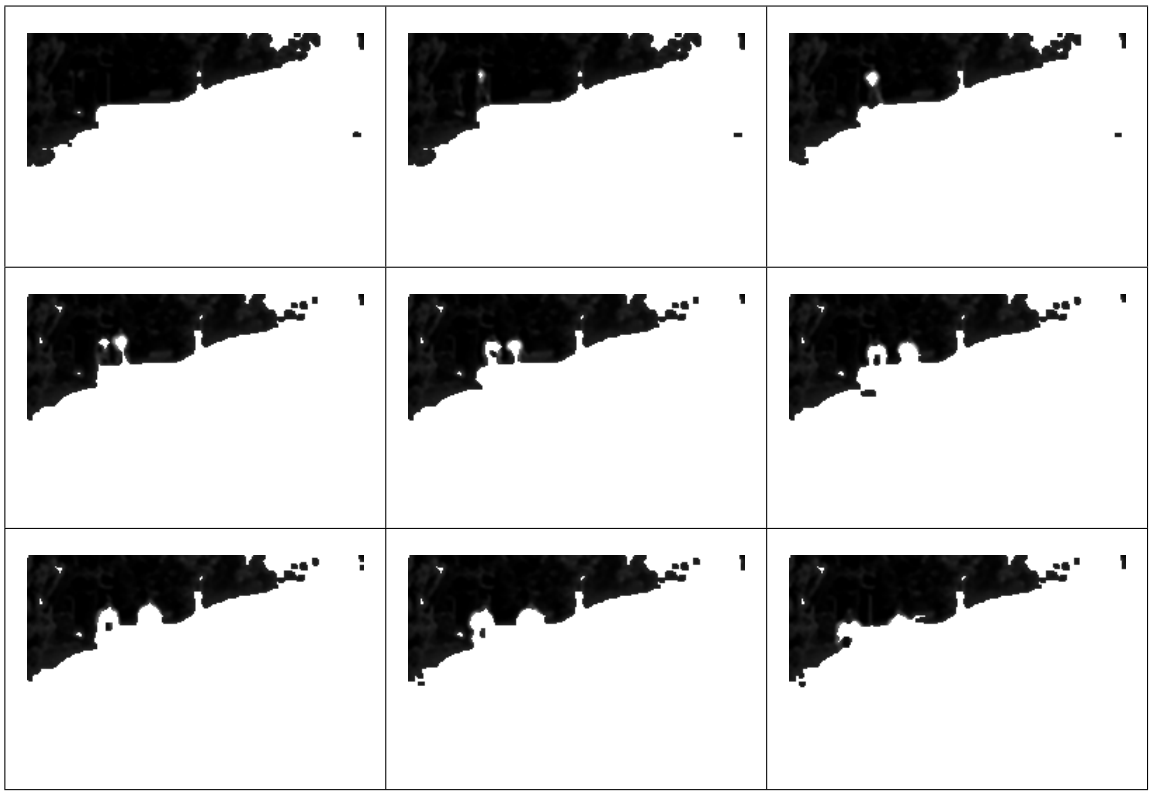


Figura 7.101: Video de vigilancia y 5 neuronas, resultados de agrupamiento del cluster 2.

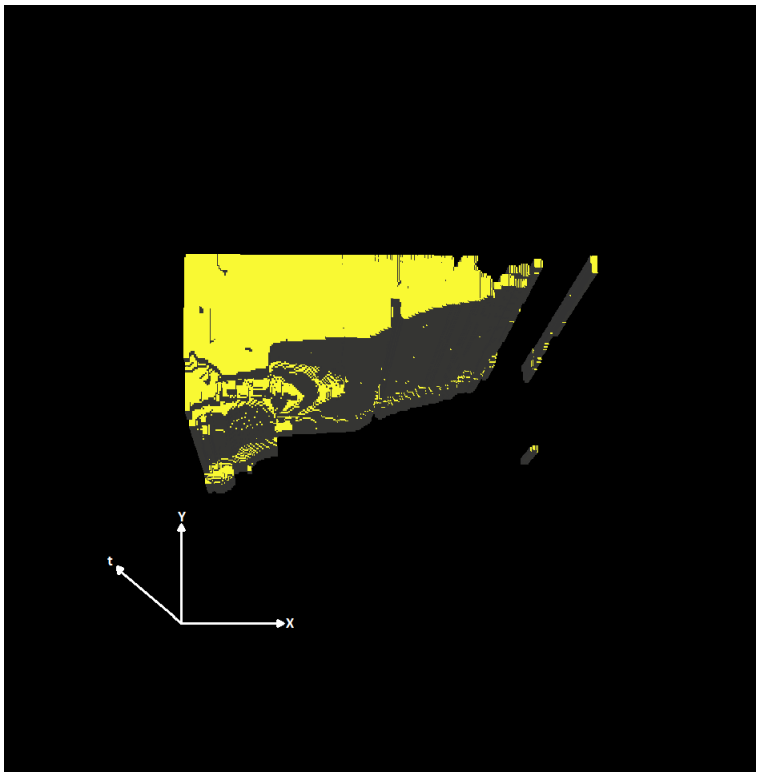


Figura 7.102: 3D-EVM del cluster 2 para el video de vigilancia y 5 neuronas.

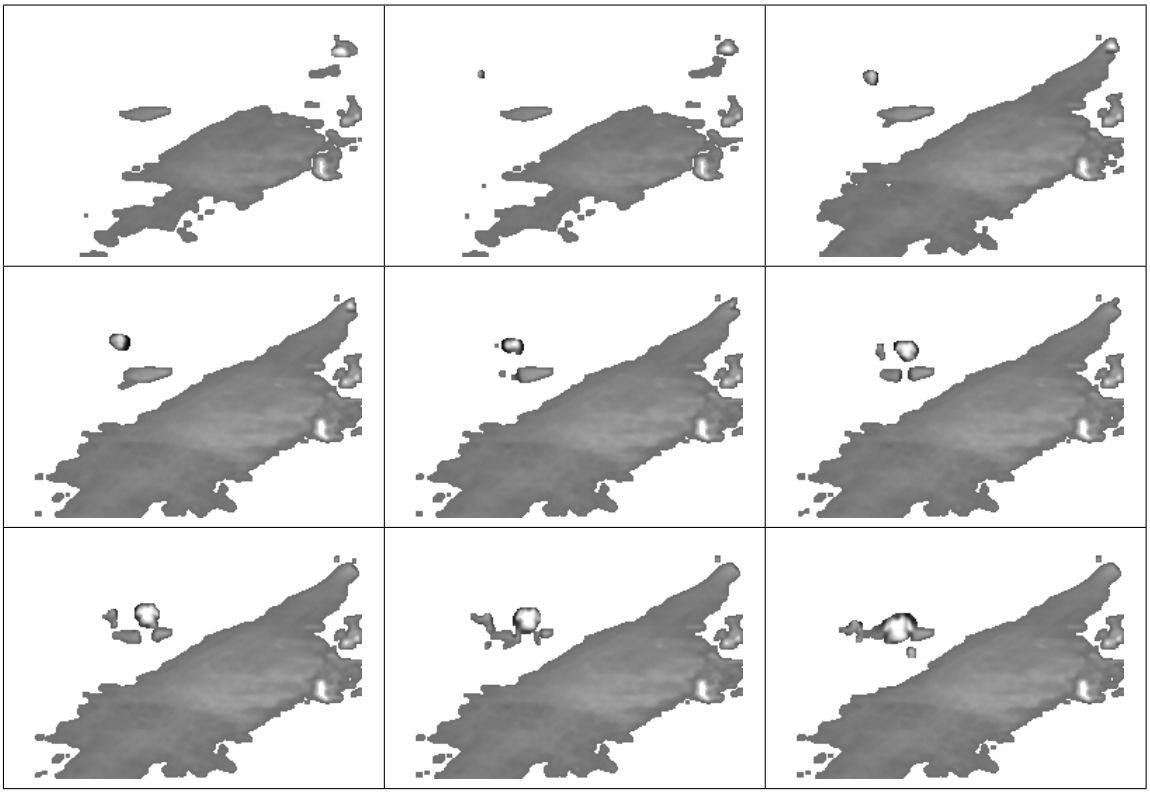


Figura 7.103: Video de vigilancia y 5 neuronas, resultados de agrupamiento del cluster 3.

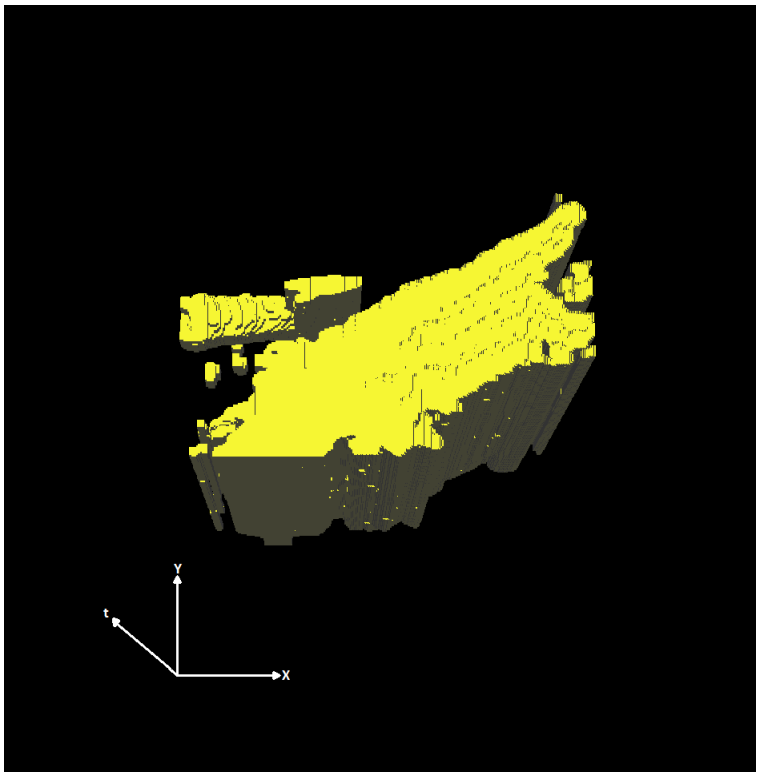


Figura 7.104: 3D-EVM del cluster 3 para el video de vigilancia y 5 neuronas.

7.5.2. Pruebas con 10 neuronas

En este apartado se describen los resultados de segmentación obtenidos para una configuración de 10 neuronas en el SOM. En la tabla 7.14 se muestra la cantidad de vértices extremos por cada 3D-EVM formado en cada cluster, en donde se observa que se tiene un total de 161934 vértices extremos. En comparación con el tamaño del 4D-EVM que representa la animación original, se tiene una tasa de compresión de 13.4 aproximadamente.

Tabla 7.14: Tamaño de los 3D-EVMs generados por cada cluster para el video de vigilancia y 10 neuronas.

Cluster	Tamaño del 3D-EVM
0	16540
1	3550
2	7888
3	1854
4	29458
5	24908
6	20250
7	8926
8	28490
9	20070
Total	161934

En la figura 7.105 se muestra de manera gráfica la distribución de sub-animaciones en los clusters, en donde se observa que los clusters que contiene una mayor cantidad de información son los clusters 2, 6 y 8. Sin embargo, se seleccionaron los clusters 0, 2, 5 y 8 debido a que contiene información más relevante de la escena.

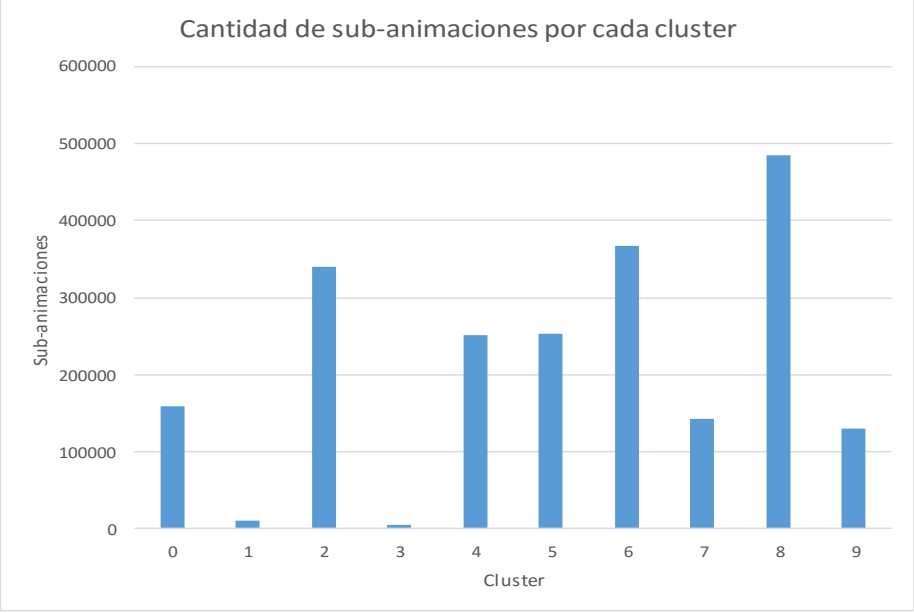


Figura 7.105: Gráfica con la cantidad de sub-animaciones agrupadas por cada cluster para el video de vigilancia y 10 neuronas.

Con base en lo anterior, a continuación se describen los clusters seleccionados:

- Para el cluster 0, se muestra su secuencia de frames en la figura 7.106, en donde se observa que se agrupan regiones oscuras de la escena, en particular se agrupan las regiones donde

se presentan bordes, ya que en estas zonas se encuentran los bordes de los elementos de la escena tal como las personas, el pavimento y los arbustos. Con lo anterior, en la figura 7.107 se presenta el 3D-EVM que se forma con dichas regiones, y claramente se observa el borde de la carretera así como su desplazamiento en el tiempo.

- En la figura 7.108 se muestra la secuencia de frames para el cluster 2, en donde se observa que se agrupan las regiones más oscuras de la escena, en este caso se tiene agrupadas las regiones de las sombras de los arbustos. Nuevamente se presenta el caso en el que se aísla el movimiento de las personas, por lo que solo se observa su silueta. El 3D-EVM correspondiente al cluster 2 se muestra en la figura 7.109, en donde claramente se observan las regiones de los arbustos como se mencionó anteriormente.
- Para el cluster 5, en la figura 7.110 se muestra su secuencia de frames, en donde se observa que se tienen agrupadas las regiones que corresponden a las zonas donde hay bordes. En este caso se puede observar que se tiene agrupada la parte de la orilla de la carretera, y de esta manera también se agrupa el movimiento de las personas, ya que el movimiento de éstas se efectúa sobre dichas regiones. Ahora, en la figura 7.111 se muestra el 3D-EVM correspondiente al cluster 5, en donde claramente se puede observar las zonas de la orilla de la carretera, también se muestra el desplazamiento a lo largo del tiempo.
- El cluster 8, del cual se tiene la secuencia de frames que se muestran en la figura 7.112, agrupa las regiones del centro de la carretera. No obstante se observa que en la parte del centro de la carretera hay regiones que no son agrupadas en este cluster, lo cual se debe a que éstas son regiones más brillantes y por ello se encuentran agrupadas en otro cluster. En comparación con las pruebas para 5 neuronas de la sección 7.5.1, se puede decir que debido a que en este caso se tienen más neuronas en el SOM, el contenido del cluster 3 de dichas pruebas es separado en más clusters.

  En la figura 7.113 se muestra el 3D-EVM del cluster 8, en donde se pueden observar las regiones agrupadas de la carretera, y en la parte de arriba se tienen las regiones asociadas al movimiento de las personas.

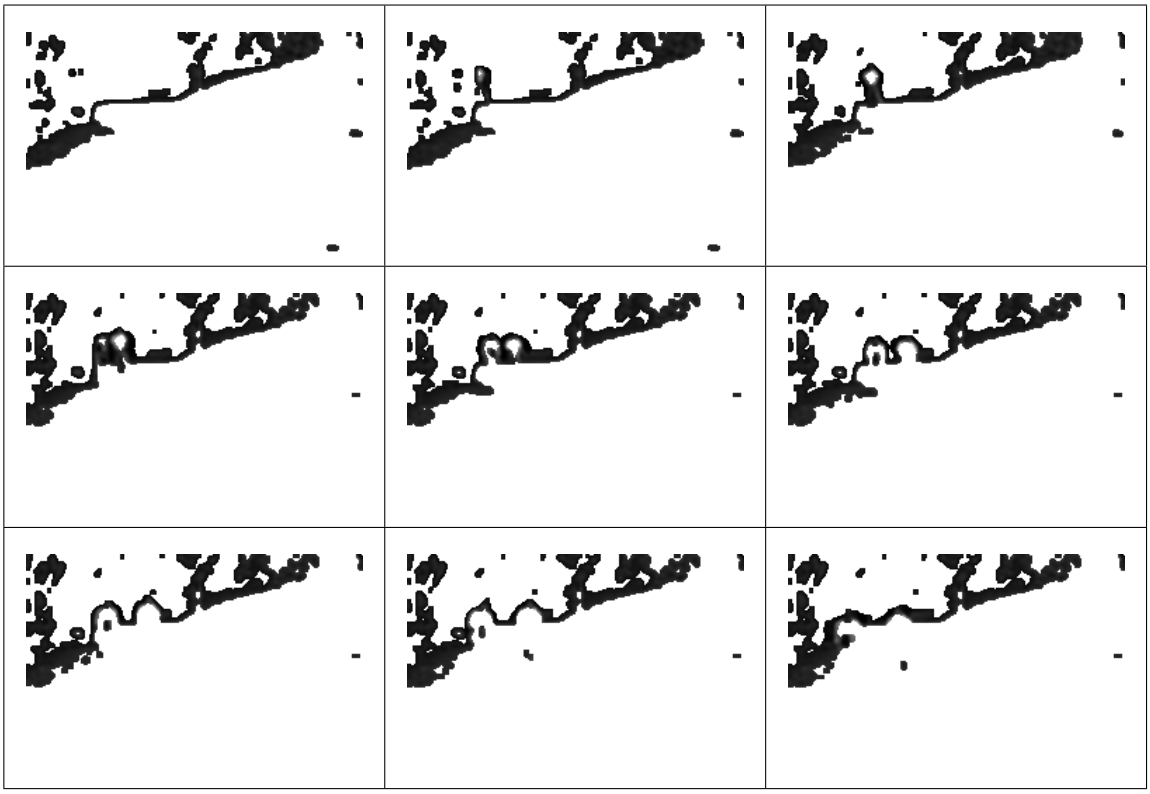


Figura 7.106: Video de vigilancia y 10 neuronas, resultados de agrupamiento del cluster 0.

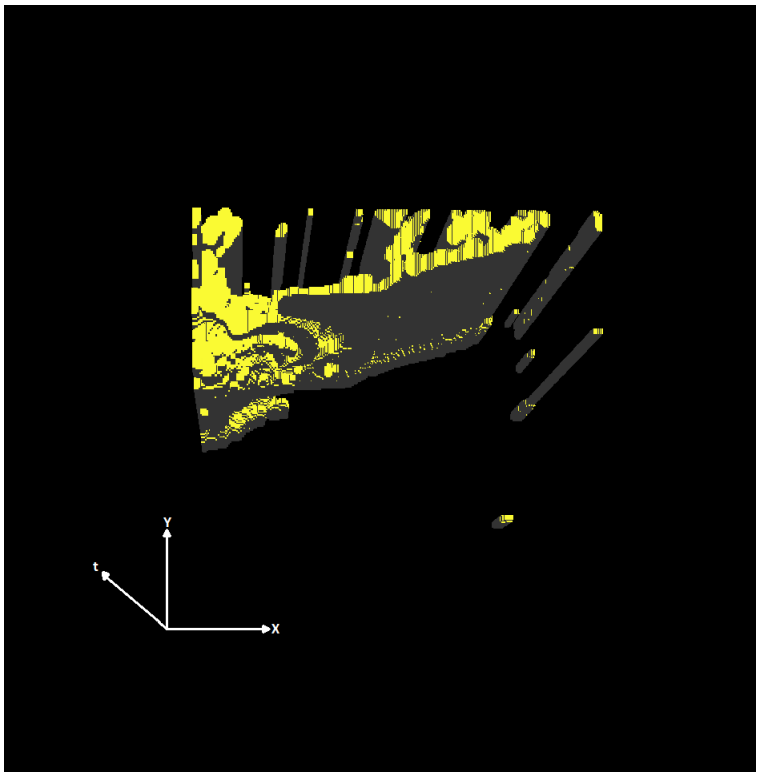


Figura 7.107: 3D-EVM del cluster 0 para el video de vigilancia y 10 neuronas.

Figura 7.108: Video de vigilancia y 10 neuronas, resultados de agrupamiento del cluster 2.

Figura 7.109: 3D-EVM del cluster 2 para el video de vigilancia y 10 neuronas.

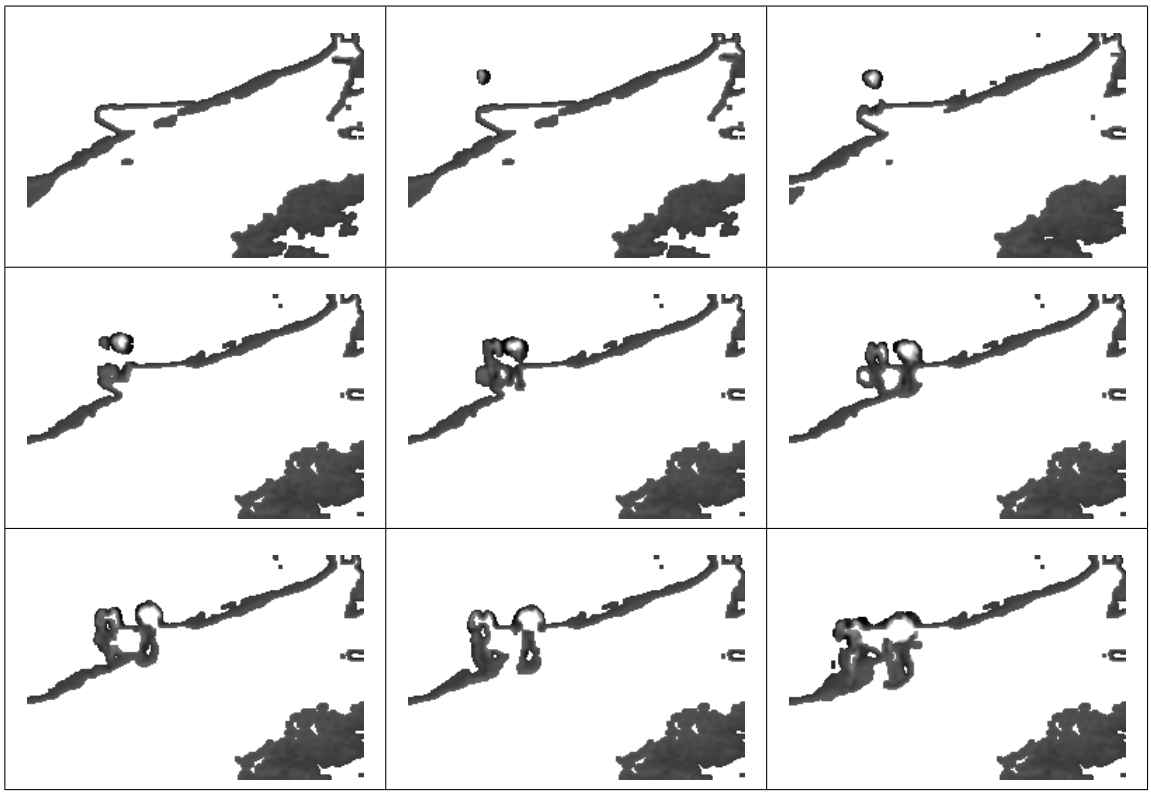


Figura 7.110: Video de vigilancia y 10 neuronas, resultados de agrupamiento del cluster 5.

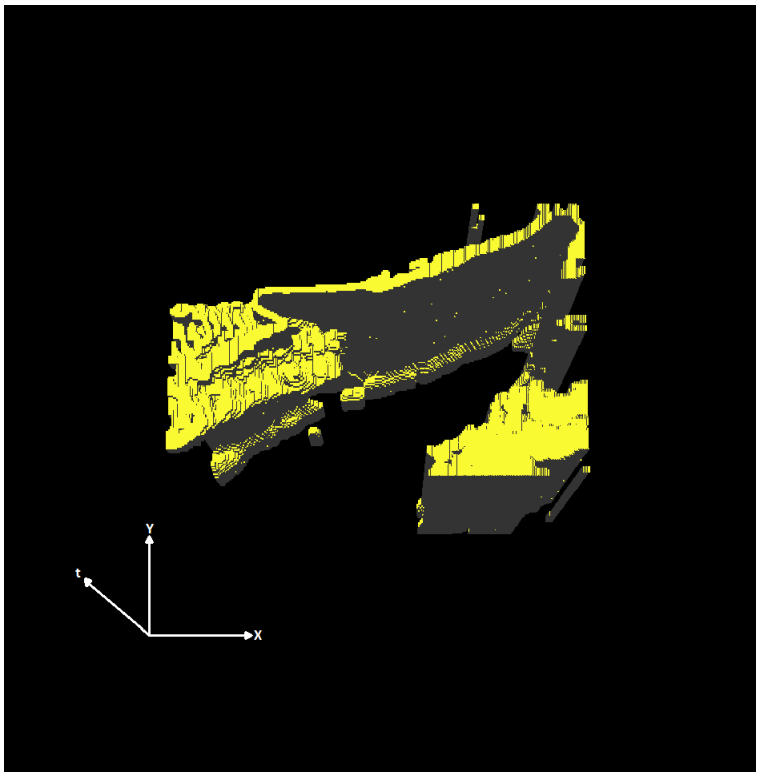


Figura 7.111: 3D-EVM del cluster 5 para el video de vigilancia y 10 neuronas.

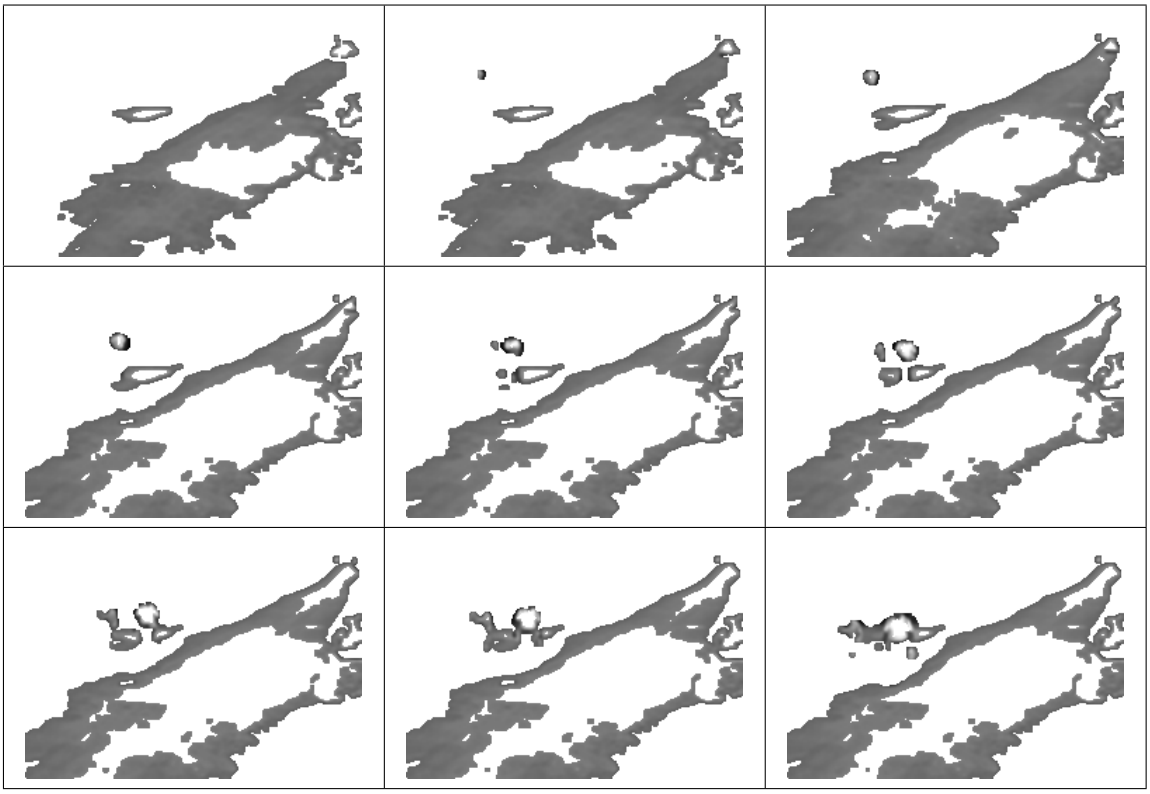


Figura 7.112: Video de vigilancia y 10 neuronas, resultados de agrupamiento del cluster 8.

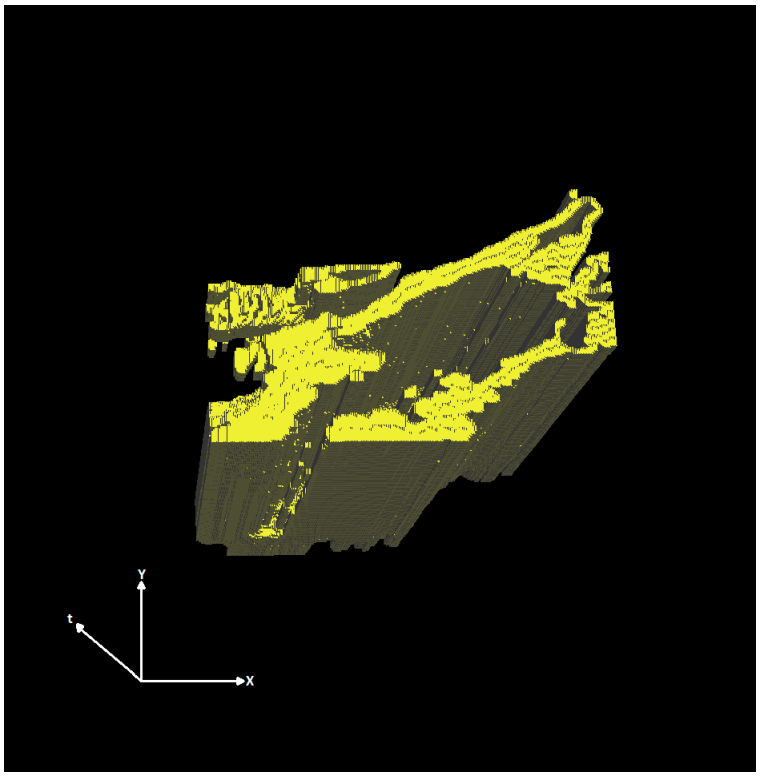


Figura 7.113: 3D-EVM del cluster 8 para el video de vigilancia y 10 neuronas.

7.5.3. Pruebas con 15 neuronas

En este apartado se presentan los resultados de las pruebas de segmentación del video de vigilancia con una configuración de 15 neuronas en el SOM. En la tabla 7.15 se muestran los tamaños de los 3D-EVMs de cada cluster, en donde se observa que se tiene un total de 241636 vértices extremos, lo cual en comparación con el tamaño del 4D-EVM que representa la animación original, se obtiene una tasa de compresión de 8.98 aproximadamente.

Tabla 7.15: Tamaño de los 3D-EVMs generados por cada cluster para el video de vigilancia y 15 neuronas.

Cluster	Tamaño del 3D-EVM
0	7200
1	1378
2	2654
3	18646
4	30836
5	16556
6	23080
7	18950
8	15314
9	7912
10	3720
11	11826
12	27280
13	29536
14	26748
Total	241636

Ahora, en la figura 7.114 se muestra de manera gráfica la cantidad de sub-animaciones que agrupa cada uno de los clusters, en donde se observa que los clusters que contienen más información son los clusters 4, 7, 9 y 12. Sin embargo, se seleccionaron los clusters 5, 6, 9 y 12 para ser analizados, ya que éstos contienen información más relevante con respecto a la escena.

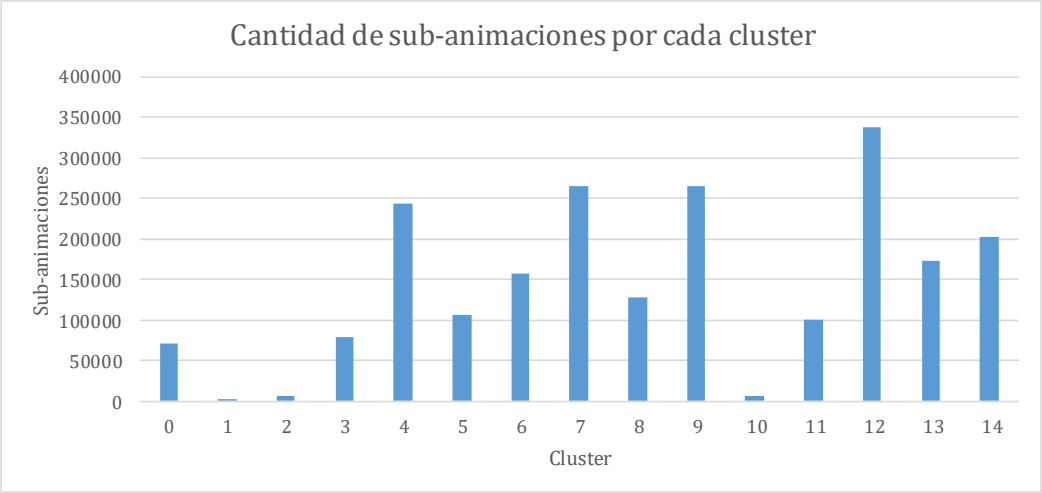


Figura 7.114: Gráfica con la cantidad de sub-animaciones agrupadas por cada cluster para el video de vigilancia y 15 neuronas.

Con base en lo anterior, a continuación se describen los clusters seleccionados:

- Para el cluster 5, se muestra su secuencia de frames en la figura 7.115, en donde se observa que se agrupan regiones que corresponden a zonas oscuras y donde se presentan los bordes. En

este caso se puede observar la orilla de la carretera y la silueta de las dos personas. En la figura 7.116 se muestra el 3D-EVM correspondiente al cluster 5, en el cual se observan los bordes y el movimiento de las personas.

- En el cluster 6, del cual se muestra su secuencia de frames en la figura 7.117, se agrupan regiones que corresponden a los bordes, este caso difiere al anterior en el sentido que las regiones agrupadas no son tan oscuras. No obstante en dichas regiones se tiene una tendencia hacia los bordes. Con esto, en la figura 7.118 se muestra el 3D-EVM, el cual contiene los bordes mencionados y claramente se observa el movimiento de las personas a lo largo del tiempo.
- Por otro lado, en la figura 7.119 se muestra la secuencia de frames para el cluster 9, en donde se observa que se agrupan las regiones más oscuras de la escena, en particular las zonas donde se encuentran los arbustos. Con esto, en la figura 7.120 se muestra el 3D-EVM correspondiente al cluster 9, en donde claramente se observan las regiones mencionadas y su evolución en el tiempo.
- Por último, en la figura 7.121 se muestra la secuencia de frames correspondiente al cluster 12, en donde se observa que se tienen agrupadas las regiones de las orillas de la carretera, así como el movimiento de personas que pasan sobre dichas regiones. Con esto en la figura 7.122 se presenta el 3D-EVM correspondiente al cluster 12 y como se observa en la parte superior se tiene el movimiento de las personas .

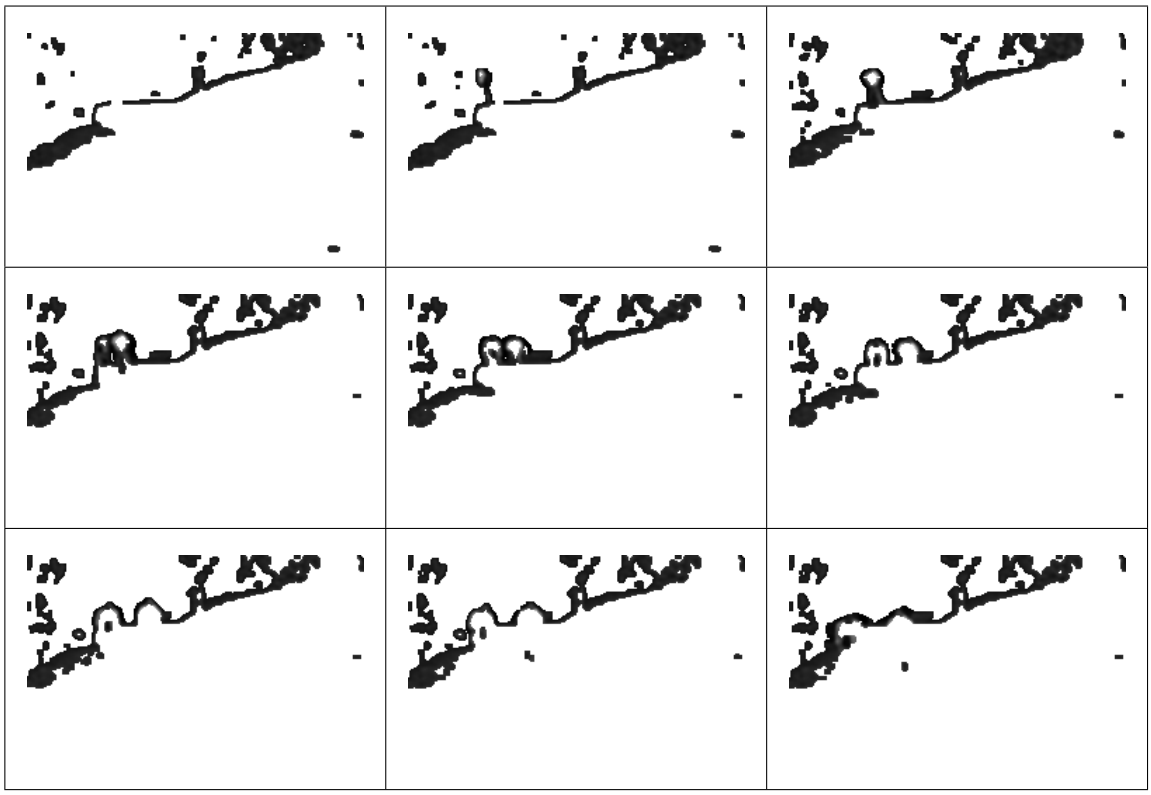


Figura 7.115: Video de vigilancia y 15 neuronas, resultados de agrupamiento del cluster 5.

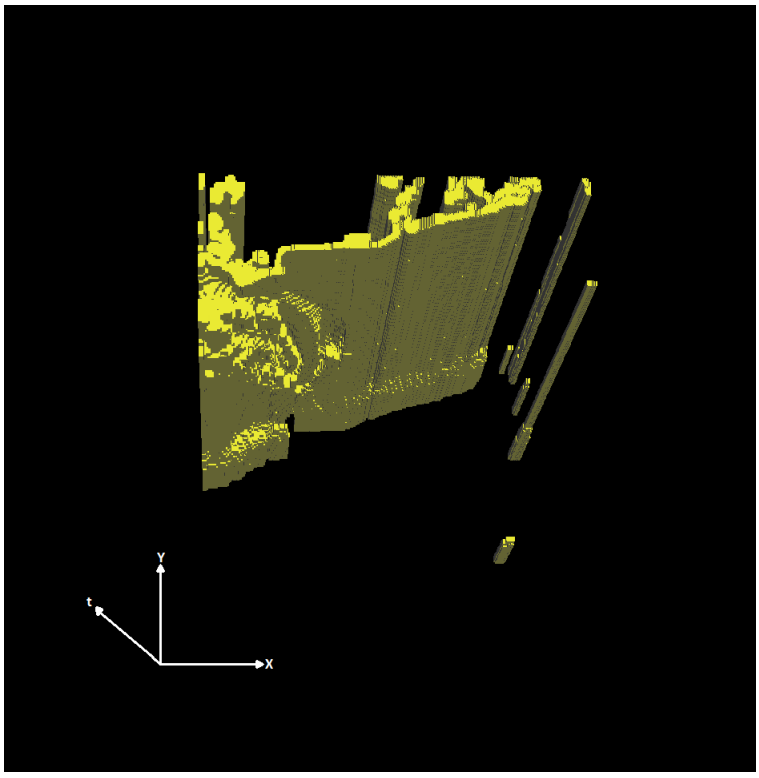


Figura 7.116: 3D-EVM del cluster 5 para el video de vigilancia y 15 neuronas.

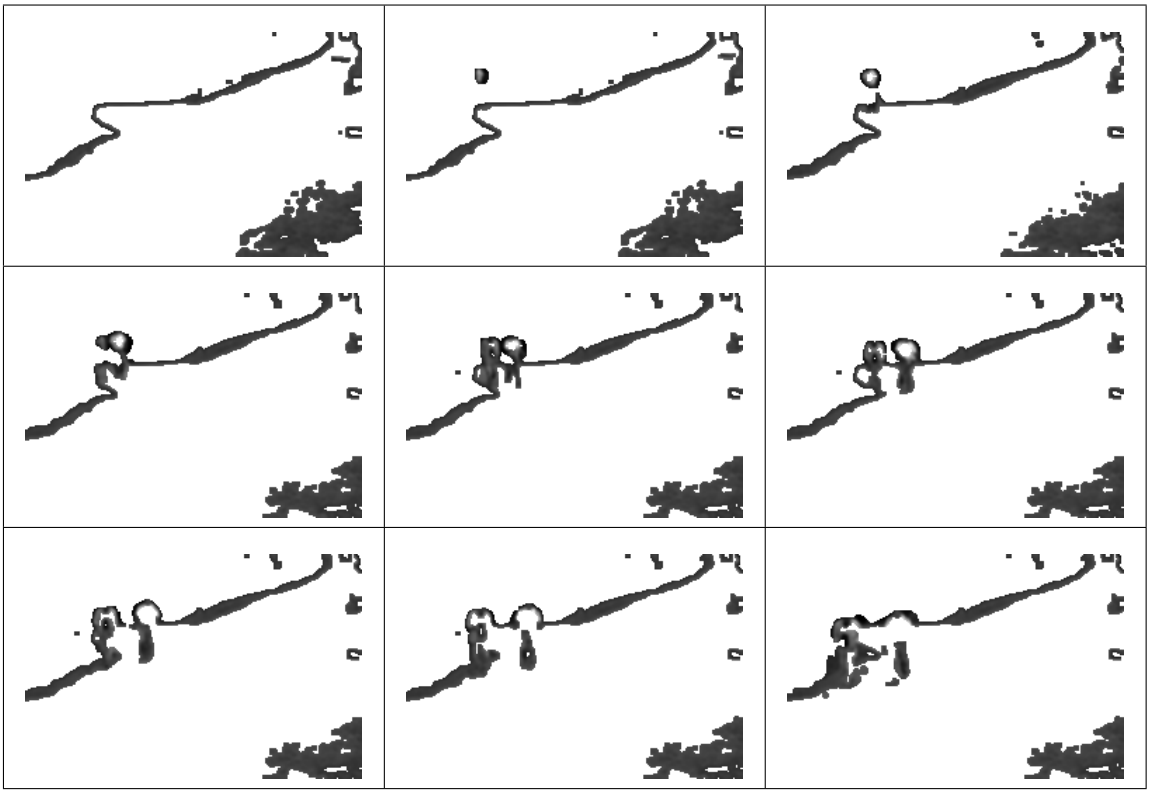


Figura 7.117: Video de vigilancia y 15 neuronas, resultados de agrupamiento del cluster 6.

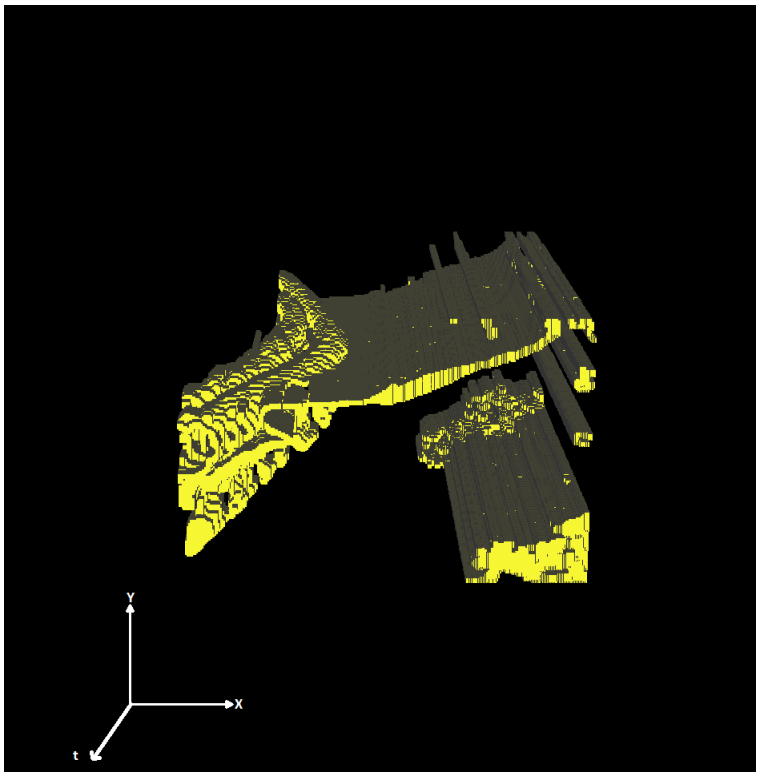


Figura 7.118: 3D-EVM del cluster 6 para el video de vigilancia y 15 neuronas.

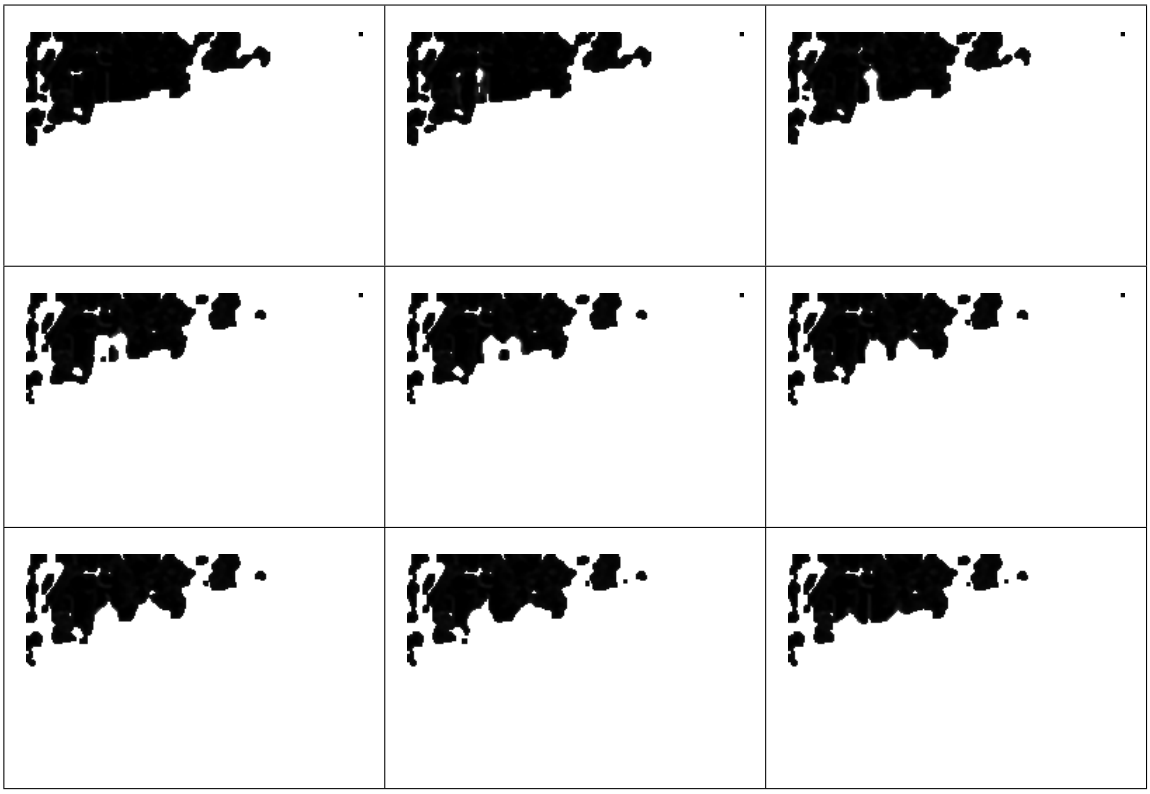


Figura 7.119: Video de vigilancia y 15 neuronas, resultados de agrupamiento del cluster 9.

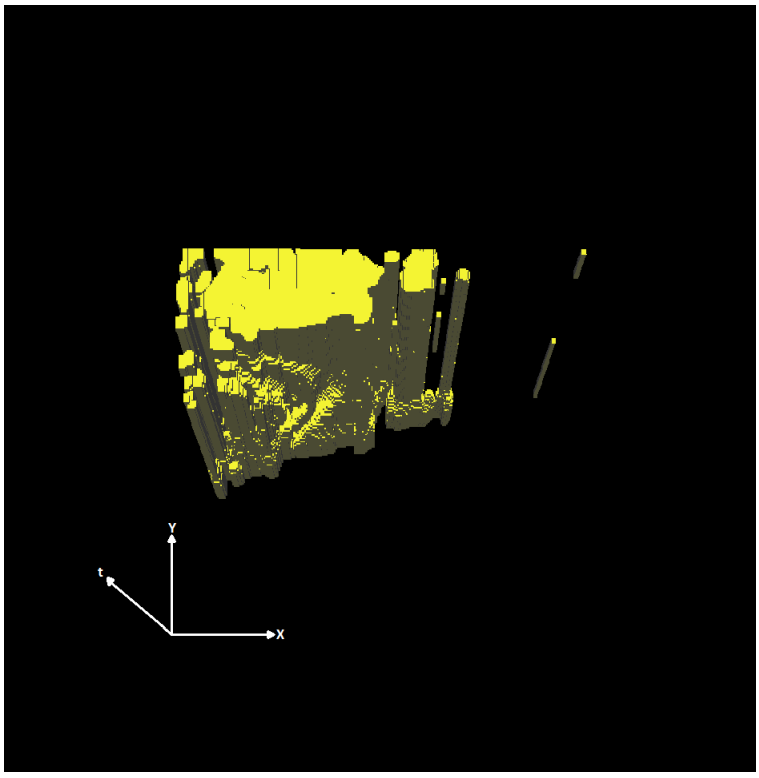


Figura 7.120: 3D-EVM del cluster 9 para el video de vigilancia y 15 neuronas.

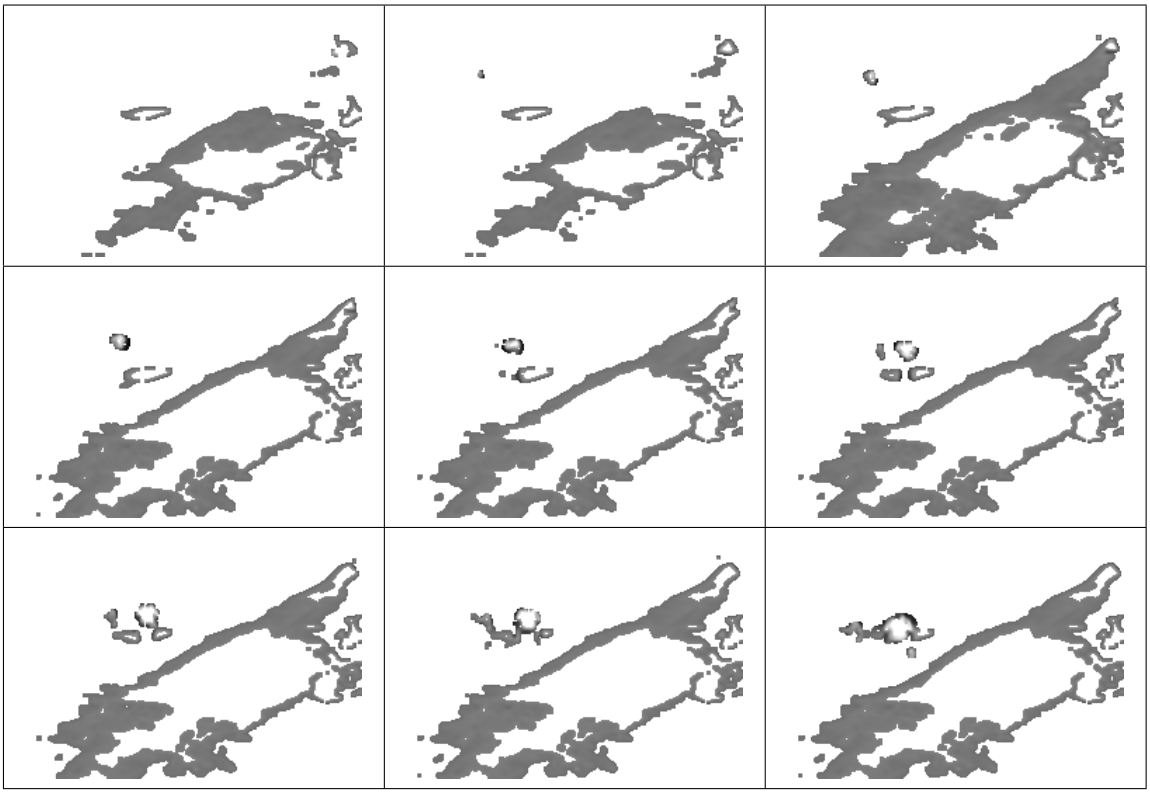


Figura 7.121: Video de vigilancia y 15 neuronas, resultados de agrupamiento del cluster 12.

Figura 7.122: 3D-EVM del cluster 12 para el video de vigilancia y 15 neuronas.

Capítulo 8

Conclusiones y Trabajo Futuro

En base al planteamiento del proyecto presentado en el capítulo 1 y al desarrollo e implementación presentados en los capítulos 5 y 6, se formulan las siguientes conclusiones:

- Se obtuvo la implementación de la librería del modelo nD-EVM y sus métodos en el lenguaje de programación *C++*, tal como se describe en la sección 6.1.
- Es importante notar que de manera jerárquica la librería del nD-EVM contiene a su vez la implementación de árboles Trie, la cual tiene como objetivo abordar el problema de almacenar los vértices extremos de un nD-EVM en una estructura de datos. Esta implementación se presenta en el apéndice A.
- Se implementó la etapa de representación de una secuencia de video en el modelo nD-EVM, la cual es la primer etapa para el framework propuesto. En la sección 5.1 se presentan los fundamentos y consideraciones para el desarrollo de esta etapa y su implementación se muestra en la sección 6.2.1, en donde también se describen los algoritmos básicos de extrusión de los componentes de color para la información de los píxeles en los frames.

  Es importante notar que aunque en este trabajo solo se consideraron casos de prueba con videos con un esquema de color RGB y escala de grises, la implementación del nD-EVM permite representar frames con cualquier esquema de color, lo cual es útil para el desarrollo de trabajos futuros.
- Se obtuvieron las implementaciones en *C++* de los algoritmos de cálculo de contenido y cálculo de DC, secciones 6.1.4 y 6.1.5 respectivamente.
- Para el proceso de segmentación se hace uso de máscaras para extraer sub-animaciones del nD-EVM que representa a la animación original y extraerle su descriptor de DC, tal como se describe en la sección 5.2.1. Para obtener el descriptor de DC se consideran los valores mínimos y máximos de la ecuación 3.6 en base a la configuración de la máscara, lo cual se establece en la sección 5.2.2.

  Con esto, se deben obtener todas las posibles sub-animaciones de la animación original, lo cual se logra mediante desplazamientos unitarios sobre los ejes coordenados (x, y, t). Este proceso es similar al proceso de convolución utilizado en el área de procesamiento de imágenes y se describe su desarrollo en la sección 5.2.3. Con ello se obtiene el conjunto de datos $DCValues$, el cual contiene los valores de DC para todas las posibles sub-animaciones.
- Un aspecto relevante acerca del proceso de extracción de sub-animaciones, consiste en que al momento de aplicar la operación intersección entre una máscara y la animación original, se considera solo la cantidad de Secciones o frames (en el nD-EVM) de la animación original que son suficientes para la intersección. De esta manera se realiza esta operación de manera eficiente ya que cuando se realicen desplazamientos en el tiempo de la máscara,

no será necesario calcular todas las Secciones previas según el algoritmo de Operaciones Booleanas. Esto se describe en la sección 6.2.3.

- Con el conjunto $DCValues$ formado por el conjunto de valores de DC de las sub-animaciones, se realiza el proceso de segmentación mediante el SOM. En donde se considera que el SOM es un módulo que toma este conjunto y obtiene un conjunto de índices de valores de DC por cada neurona en su configuración, los cuales contienen los índices de DC que agrupa cada cluster. Estos índices son utilizados para determinar la posición de las sub-animaciones en el espacio 2D y en el tiempo, y de esta manera se obtienen las respectivas regiones de la animación original que corresponden a cada cluster.

  Con las posiciones espaciales en 2D y el desplazamiento en el tiempo se realizan desplazamientos en una máscara sin componentes de color para formar los respectivos 3D-EVMs para cada cluster, ya que un cluster solo contiene las regiones correspondientes y su clasificación es el número de la neurona. Los 3D-EVMs se forman mediante la unión de todas las máscaras agrupadas por el cluster y considerando sus respectivos desplazamientos en los ejes (x, y, t). No obstante en el capítulo 7 para cada prueba se muestra la secuencia de frames por cada cluster analizado, lo cual se obtiene mediante la consulta de la información de color de la animación original para las regiones que agrupa cada cluster.

- En general, el framework de segmentación desarrollado permite separar las regiones de una animación, de manera que se agrupan patrones similares. De los resultados presentados en el capítulo 7 se observaron los siguientes casos:

  - ⋆ Hay clusters que agrupan una cantidad considerable de información, ya que agrupan regiones del fondo de la escena o superficies con color uniforme.
  - ⋆ Cuando se tiene una cantidad pequeña de clusters, algunos clusters tienden a agrupar la mayor parte de la información, mientras que otros tienden a estar vacíos.
  - ⋆ Hay casos en los que se presenta un agrupamiento de regiones donde hay bordes, lo cual tiene sentido debido a que en esas regiones su forma es similar debido al cambio brusco en el color. De manera general, en las regiones de los bordes hay un cambio de un color claro a un color oscuro.
  - ⋆ También se presenta el caso en el que se agrupan regiones donde hay sombras, hay casos específicos en donde se agrupan las sombras en las regiones donde son más pronunciadas, que son para las zonas que se encuentran justo debajo de los objetos de la escena.
  - ⋆ Por último, un caso interesante es para clusters que agrupan regiones donde se hallan elementos de la escena en movimiento. Esto es interesante en el sentido que el SOM es capaz de identificar patrones de movimiento y aislarlos de su entorno, tal como se puede apreciar en los casos de estudio de las secciones 7.2 y 7.4.

De manera satisfactoria, se logró el desarrollo e implementación del framework de segmentación de video, considerando todas las etapas propuestas en el capítulo 1. Con ello, se hace válida la hipótesis planteada para este proyecto.

8.1 Propuestas de Trabajos Futuros

En base a los resultados obtenidos y las implementaciones realizadas, se proponen los siguientes trabajos futuros:

- Como se describe en la sección 6.2.1, para este trabajo se consideró que los frames tienen el formato *BMP* debido a la simplicidad para obtener la información de los píxeles. Por esto se propone desarrollar una librería en *C++* que permita obtener la representación en el nD-EVM de frames en diversos formatos de imagen.

- Realizar un estudio acerca de los resultados de segmentación para frames que tengan esquemas de color diferentes al RGB y escala de grises. Este estudio permitirá evaluar si es recomendable usar diferentes esquemas de color a los ya usados, así como determinar si es mejor usar más de tres componentes de color, por ejemplo el esquema RGBA que considera un componente de transparencia.
- Como se ha observado en el capítulo 7, los resultados de clasificación sustentada en el uso de DC son realmente alentadores. Por ello consideramos es importante proponer algoritmos para el cálculo de más descriptores de forma para los nD-EVMs, ya que de esta manera se pueden formar vectores de características de un nD-EVM. Esto ayudaría a mejorar la diferenciación de los nD-EVMs al considerar más elementos para su comparación, con lo cual por ejemplo en el caso de los objetivos de este trabajo, mediante el SOM se podría obtener un mejor agrupamiento de las sub-animaciones. No obstante, el reto aquí radica en proponer algoritmos para el calculo de descriptores que apliquen a EVMs de cualquier dimensionalidad o definir casos específicos.
- Como se mencionó anteriormente, el módulo de agrupamiento utilizando un SOM es independiente de los procesos que le preceden y los subsecuentes. Con ello, es posible utilizar otra técnica según se desee, solo se debe considerar que se debe tomar como conjunto de datos el conjunto $DCValues$ que contiene el conjunto de valores de DC de las sub-animaciones de manera ordenada, y como resultado del agrupamiento se debe retornar por cada cluster el conjunto de los índices de DC que éstos agrupan. En este sentido se puede usar otros métodos de agrupamiento (*clustering*), por ejemplo *k-means*, *fuzzy c-means* y *cuantización vectorial*, por mencionar algunos.
- Considerando que en los resultados se obtuvieron casos en donde se agrupan regiones donde hay bordes de los elementos de la escena, se propone usar el enfoque de aprendizaje supervisado para entrenar un detector de bordes. Es decir, que se debe formar un conjunto de datos de entrenamiento, el cual contiene una cantidad considerable de ejemplos de regiones con bordes, con ello se realiza el entrenamiento de un clasificador (una red neuronal por ejemplo) el cual *"aprenderá"* a detectar los bordes de una animación dada. Es importante considerar que la animación de entrada es completamente diferente a las animaciones de entrenamiento, y el resultado esperado es que como salida el clasificador devuelva una versión que contenga solo los bordes de la animación de entrada.

  Es importante notar que esta idea se puede aplicar al procesamiento de imágenes, ya que en principio el clasificador funcionaría de manera similar, solo que en este caso no se tiene información sobre la evolución de los frames en el tiempo.
- Extendiendo la idea anterior, se propone usar el mismo enfoque al anterior pero para detectar objetos en animaciones o imágenes. En este caso se debe generar una conjunto de entrenamiento considerablemente grande para entrenar el clasificador, en donde se tiene información de la clase de objetos que se desea detectar, por ejemplo autos, personas, animales o cualquier otro objeto. El resultado esperado es que el clasificador pueda detectar, en una animación o imagen, los objetos que ha aprendido a identificar.
- Más aún, sería interesante utilizar el mismo enfoque anterior para que un clasificador pudiera detectar objetos en movimiento en una animación. Por ejemplo se podría generar un conjunto de datos de entrenamiento sobre personas o autos en movimiento, y el resultado esperado es que el clasificador, dado un video de entrada, pueda separar dichos objetos de su entorno.

Apéndice A

Implementación de Árboles Trie y sus Métodos

Un árbol *Trie* consiste en un árbol de búsqueda, en donde sus elementos se ordenan en base a su clave (*key*), también se considera un ordenamiento en base al nivel de profundidad [11, 52]. En esta implementación, los nodos en la estructura, son considerados como listas ordenadas. Es decir, que para cada nivel, en el árbol Trie, hay una lista ordenada con todas las claves del nivel correspondiente.

En este sentido, cada nodo del árbol Trie tiene tres elementos como se muestra en la figura A.1a y se describen a continuación:

- El valor de la clave.
- Un apuntador hacia el siguiente elemento de la lista, dicha lista está asociada al nivel correspondiente.
- Un apuntador al siguiente nivel, el cual se forma por una lista ordenada con las claves para dicho nivel. Entonces, este apuntador hace referencia al primer nodo de la lista del siguiente nivel. Para coincidir con el modelo nD-EVM, se asociará el nivel de profundidad del Trie, con la dimensión actual del nD-EVM.

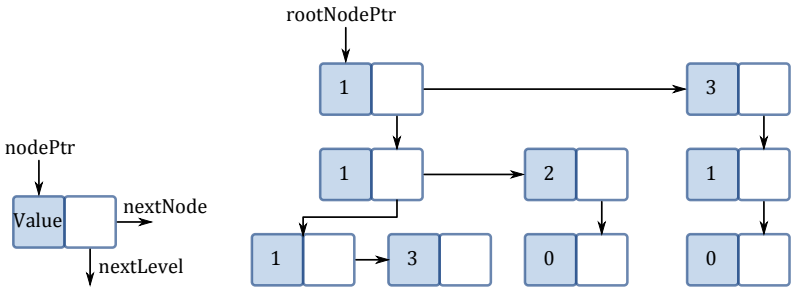
Para ilustrar cómo se forma un árbol Trie, supongamos que se tiene el siguiente conjunto de vértices y se desea almacenarlos en un árbol Trie:

$$\{1,1,1\},\{1,1,3\},\{1,2,0\},\{3,1,0\}$$

En la figura A.1b se muestra el Trie resultante de dichos vértices, en donde se observa que cuando se presentan elementos comunes en los vértices, en la lista asociada al nivel correspondiente, se almacena solo un nodo.

Es importante notar que la implementación presentada fue diseñada para soportar cualquier tipo de dato que proporciona el lenguaje *C++*, por ello en cada definición de algún método o clase se usan los macros *template<typename valueType>*, en donde *valueType* es el tipo de dato. Para representar una secuencia de frames se utiliza el tipo de dato *unsigned int*.

En el código A.1, se presenta la definición de la estructura básica para un nodo del árbol Trie, esta es la estructura *trieNode*. Como se observa, esta estructura consiste de tres elementos: el valor actual del nodo, un apuntador al siguiente nodo en la lista ordenada en la dimensión actual y un apuntador al primer nodo de la lista ordenada de la siguiente dimensión.



(a) Nodo de un árbol Trie y sus elementos.

(b) Ejemplo de un árbol Trie.

Figura A.1: Nodo de un árbol Trie y ejemplo de la formación de un Trie.

código A.1: Estructura de datos para implementar un nodo del árbol *Trie*

```
1 template<typename valueType>
2 struct trieNode{
3     valueType value;
4     trieNode *nextDim;
5     trieNode *nextTrieNode;
6 };
```

Con la definición de los nodos *trieNode* es posible implementar un árbol *TrieTree*. En el código A.2, se define la clase *TrieTree*, la cual cuenta con un apuntador al nodo raíz del árbol Trie, este es el nodo *rootNode*. Como se observa, también se cuenta con el apuntador doble *coupletIndex*, el cual se utiliza en la exploración de Couplets, el uso de este apuntador se analizará más adelante. También, se tiene la variable *isCouplet*, la cual es importante para determinar si el Trie actual corresponde a un Couplet, y es útil para evaluar si éste se elimina o no. Posteriormente, se definen los constructores y los métodos que se describirán posteriormente.

código A.2: Definición de la Clase *TrieTree*

```
1  template<typename valueType>
2  class TrieTree {
3  public:
4      trieNode<valueType> *rootNode = NULL;
5      trieNode<valueType> **coupletIndex;
6      bool isCouplet;
7
8      // - Constructores
9      // - Metodos
10 private:
11 };
```

A.1 Métodos Básicos para la clase TrieTree

En este apartado se describen algunos métodos fundamentales para el funcionamiento apropiado de la clase TrieTree. En el código A.3 se muestran dichos métodos y se describen a continuación:

- *resetCoupletIndex* (línea 2): Este método inicializa el apuntador *coupletIndex* al nodo raíz del árbol Trie. Este método es importante, ya que en la exploración de Couplets cuando se alcanza el último Couplet se reinicia este apuntador. Con ello es posible realizar otra exploración de Couplets cuando sea requerido por algún método.
- *isEmpty* (línea 7): Este método evalúa si el Trie se encuentra vacío y retorna un valor Booleano.
- *endTrie* (línea 15): Se evalúa si el apuntador *coupletIndex* ha llegado al final del árbol Trie, lo cual se utiliza en mayor medida para finalizar la ejecución de algún proceso.

- *putCouplet* (línea 23): Agrega el árbol Trie de un Couplet (n-1)D a un Trie que corresponde a un nD-EVM. Como se observa, este proceso se realiza al establecer el apuntador *coupletIndex* al nodo raíz del Trie del Couplet. Posteriormente se actualiza *coupletIndex* al siguiente nodo, lo cual permite agregar otro Couplet y así sucesivamente hasta que el proceso que realiza las llamadas finalice su ejecutión.
- *readCouplet* (línea 29): Obtiene el siguiente Couplet de un Trie en base a *coupletIndex*. Como se observa, se forma un nuevo objeto TrieTree considerando como nodo raíz al que actualmente contiene *coupletIndex*.
- *setCoord* (línea 40): Agrega un nodo con la coordenada *coord*, la cual corresponde al eje coordenado x_1.
- *getCoord* (línea 49): Obtiene la coordenada en el eje x_1 del Couplet que contiene *coupletIndex*.

código A.3: Métodos Básicos de la clase TrieTree

```
1  template<typename valueType>
2  void TrieTree<valueType>::resetCoupletIndex(){
3      coupletIndex = &rootNode;
4  }
5
6  template<typename valueType>
7  bool TrieTree<valueType>::isEmpty(){
8      if(rootNode == NULL)
9          return true;
10     else
11         return false;
12 }
13
14 template<typename valueType>
15 bool TrieTree<valueType>::endTrie(){
16     if(*coupletIndex == NULL)
17         return true;
18     else
19         return false;
20 }
21
22 template<typename valueType>
23 void TrieTree<valueType>::putCouplet(TrieTree* couplet){
24     (*coupletIndex) = couplet->rootNode;
25     coupletIndex = &((*coupletIndex)->nextTrieNode);
26 }
27
28 template<typename valueType>
29 TrieTree<valueType> *TrieTree<valueType>::readCouplet(){
30     if(!endTrie()){
31         TrieTree *couplet = new TrieTree((*coupletIndex)->nextDim);
32         coupletIndex = &((*coupletIndex)->nextTrieNode);
33         couplet->isCouplet = true;
34         return couplet;
35     }else
36         return NULL;
37 }
38
39 template<typename valueType>
40 void TrieTree<valueType>::setCoord(valueType coord){
41     trieNode<valueType>* coupletRoot = new trieNode<valueType>;
42     coupletRoot->value = coord;
43     coupletRoot->nextTrieNode = NULL;
44     coupletRoot->nextDim = rootNode;
45     rootNode = coupletRoot;
46 }
47
48 template<typename valueType>
49 valueType TrieTree<valueType>::getCoord(){
50     return (*coupletIndex)->value;
51 }
```

El método del código A.4 se utiliza para obtener la cantidad de vértices extremos en un árbol Trie. Como se observa, se realiza una exploración en profundidad (líneas 18 y 20) del árbol Trie y se realiza un conteo de los nodos hoja (líneas 13-16), ya que éstos representan un vértice extremo.

código A.4: Método para obtener la cantidad de vértices extremos de un Trie.

```
template<typename valueType>
valueType TrieTree<valueType>::size(){
    valueType trieSize = 0;
    if((&rootNode) == NULL){
        return 0;
    }
    size(rootNode,&trieSize);
    return trieSize;
}

template<typename valueType>
void TrieTree<valueType>::size(trieNode<valueType> *currentNode,valueType *trieSize){
    if((currentNode) == NULL){
        *trieSize = (*trieSize) +1;
        return;
    }

    size((currentNode)->nextDim,trieSize);

    if((currentNode)->nextTrieNode != NULL){
        size((currentNode)->nextTrieNode,trieSize);
    }
}
```

A.2 Método para Insertar un Vértice en el Árbol Trie

En el código A.5, se muestra el método que inserta un nuevo vértice en el árbol Trie. En donde el vértice a insertar se recibe en un arreglo, el cual tiene como longitud la dimensión máxima del árbol Trie. Por lo que, los elementos del arreglo de entrada están ordenados en base a la dimensión que les corresponde. Por ejemplo: para un arreglo de entrada con los elementos $\{3,4,5\}$, el cual tiene longitud 3, se generarán tres nodos: Un nodo con valor 3 en la dimensión 1, un nodo con valor 4 en la dimensión 2 y un nodo con valor 5 en la dimensión 3. También, se debe establecer el orden entre los nodos, mediante el apuntador *nextDim*.

Se insertará cada elemento del arreglo de entrada como un nodo en cada dimensión según el orden en este. Cabe mencionar que los nodos serán insertados sí y solo sí no hay otro nodo con el mismo valor en la dimensión correspondiente. Ya que se puede presentar el caso en que varios vértices en el árbol compartan un elemento para una dimensión específica, pero solo se debe almacenar un valor para todos. Por ejemplo si se tienen dos vértices: $\{2,4,5\}$ y $\{2,4,7\}$, entonces para las dimensiones 1 y 2 se almacenará solo un nodo por cada elemento, uno con valor 2 y otro con valor 4. No obstante para las demás dimensiones debe haber un nodo para cada uno de los elementos restantes, ya que son diferentes.

código A.5: Función para insertar un vértice en un árbol Trie

```
template<typename valueType>
void TrieTree<valueType>::insertVertex(valueType * inputKey,int length){
    trieNode<valueType> * prevNode = NULL;
    insertVertex(&prevNode,&rootNode,inputKey,length,0,0,length);
}

template<typename valueType>
bool TrieTree<valueType>::insertVertex(trieNode<valueType> **prevNode, trieNode<valueType>
    **currentNode,valueType* inputKey, int length, int prevDim, int currentDim, int matchCount){
    if(!(currentDim < length))
        if(matchCount == 0)
            return true;
        else
            return false;

    if(*currentNode == NULL){
        trieNode<valueType> *node = new trieNode<valueType>;
        node->value = inputKey[currentDim];
        node->nextDim = NULL;
        node->nextTrieNode = NULL;

        *currentNode = node;
        if(*prevNode != NULL)
            if(prevDim == currentDim)
```

```
25                  (*prevNode)->nextTrieNode = *currentNode;
26              else
27                  (*prevNode)->nextDim = *currentNode;
28          insertVertex(currentNode,&((*currentNode)->nextDim),inputKey,length,currentDim,currentDim
        +1,matchCount);
29      }else{
30          if((*currentNode)->value == inputKey[currentDim]){
31              bool vertexStatus = insertVertex(currentNode,&((*currentNode)->nextDim),inputKey,length
        ,currentDim,currentDim+1,matchCount-1);
32              if(vertexStatus){
33                  if(rootNode == (*currentNode)){
34                      if(rootNode->nextTrieNode != NULL){
35                          trieNode<valueType> *tempNode = rootNode;
36                          rootNode = rootNode->nextTrieNode;
37                          delete tempNode;
38                          tempNode = NULL;
39                      }else{
40                          rootNode == NULL;
41                          delete *currentNode;
42                          *currentNode = NULL;
43                      }
44                  }else{
45                      if((*prevNode)->nextDim == (*currentNode))
46                      {
47                          if((*currentNode)->nextTrieNode == NULL)
48                          {
49                              delete *currentNode;
50                              *currentNode = NULL;
51                              return true;
52                          }else
53                          {
54                              trieNode<valueType> *node = (*currentNode)->nextTrieNode;
55                              delete *currentNode;
56                              *currentNode = NULL;
57                              (*prevNode)->nextDim = node;
58                              return false;
59                          }
60                      }else
61                      {
62                          if((*currentNode)->nextTrieNode == NULL)
63                          {
64                              delete *currentNode;
65                              *currentNode = NULL;
66                          }else
67                          {
68                              trieNode<valueType> *node = (*currentNode)->nextTrieNode;
69                              delete *currentNode;
70                              *currentNode = NULL;
71                              (*prevNode)->nextTrieNode= node;
72                          }
73                          return false;
74                      }
75                  }
76              }
77          }else
78              if((*currentNode)->value > inputKey[currentDim]){
79                  trieNode<valueType> *node = new trieNode<valueType>;
80                  node->value = inputKey[currentDim];
81                  node->nextDim = NULL;
82                  node->nextTrieNode = (*currentNode);
83                  if(rootNode == (*currentNode))
84                      rootNode = node;
85                  else{
86                      if((*prevNode)->nextDim == (*currentNode)){
87                          (*prevNode)->nextDim = node;
88                      }else{
89                          (*prevNode)->nextTrieNode = node;
90                      }
91                  }
92
93                  insertVertex(&node,&node->nextDim,inputKey,length,currentDim,currentDim+1,
        matchCount);
94              }else
95                  insertVertex(currentNode,&((*currentNode)->nextTrieNode),inputKey,length,currentDim
        ,currentDim,matchCount);
96      }
97  }
```

Entonces, básicamente el método presentado en el código A.5, realiza una búsqueda para verificar

si existen nodos en la dimensión correspondiente, que tengan valores iguales a los del arreglo de entrada. Si no existe un nodo para algún elemento del arreglo de entrada, se generará un nodo con dicho valor. En este sentido, y de manera general para todos los métodos, la búsqueda se realiza primero en profundidad. Como se observa se tienen dos funciones *insertVertex*, la de la línea 2 se utiliza para la llamada general, por otro lado la de la línea 8 se utiliza para realizar la inserción de los nodos en el Trie. El método de la línea 8 recibe los siguientes parámetros:

- *prevNode*: Nodo Trie previo en la exploración.
- *currentNode*: Nodo Trie actual en la exploración.
- *inputKey*: Apuntador al arreglo del vértice que se va a insertar.
- *length*: Tamaño máximo del arreglo del vértice a insertar.
- *prevDim*: Dimensión previa en la exploración.
- *currentDim*: Dimensión actual en la exploración.
- *matchCount*: Contador de coincidencias en el Trie, usado para saber si el vértice que se va a insertar ya existe.

La condición de la línea 10, se utiliza para determinar si se ha explorado todo el arreglo del vértice de entrada, con ello se decide cuando detener la exploración. El parámetro *matchCount* indica cuántos nodos del arreglo de entrada ya existen en el Trie actual. Este parámetro, inicialmente tiene un número igual a la longitud del arreglo de entrada. Con ello, cuando hay una coincidencia de valores, se decrece este parámetro en una unidad. Así, si al finalizar la exploración del Trie, el parámetro *matchCount* es cero, quiere decir que el vértice que se intenta insertar ya existe.

La condición de la línea 16, indica cuando en la exploración se llega a un punto en donde el apuntador actual es nulo. Es decir que el valor del arreglo del vértice a insertar en la dimensión actual no existe. Por lo que se debe crear un nuevo nodo con el valor correspondiente. Sin embargo, existen tres posibles escenarios:

- Si el nodo previo es nulo, entonces se trata del nodo raíz. Lo cual se evalúa en la condición de la línea 23. Esta condición solo se utiliza para hacer ajustes en los apuntadores de los nodos previos.
- Si se llega por una exploración en anchura, entonces se debe actualizar el apuntador *nextTrieNode* del nodo previo (condición de la línea 24).
- En otro caso, se llega mediante una exploración por profundidad, entonces se debe actualizar el apuntador *nextDim* del nodo previo.

En la línea 30, se evalúa el caso en que se halla una coincidencia en el árbol Trie. A partir de este punto, se valuará el valor de verdad que retorna este método. Ya que si se retorna TRUE, se deben eliminar los nodos que ya existen. Con esto se resuelve de manera directa, el problema de eliminar vértices que no sean extremos. Ya que un vértice extremo se insertará un número impar de veces, con lo que si se preservará. Un vértice que no es extremo, se insertará un número par de veces, por lo que al final de la construcción del Trie, este vértice se eliminará.

En la línea 78, se realiza la exploración del Trie en la dimensión actual. Ello para evaluar si el valor de clave a insertar ya existe o no. Sin embargo, como el Trie es un árbol ordenado, entonces se debe evaluar el caso en donde hay un vértice con un valor mayor al que se va a insertar. Si se halla un nodo con un valor mayor, entonces se crea un nuevo nodo y se inserta antes de dicho nodo.

A.3 Comparación de dos Tries

En el código A.6 se muestra el método para comparar dos árboles Trie, como se observa se tienen dos definiciones, la de la línea 2 es la llamada principal de este método, y en la línea 10 se tiene el método que realiza la comparación como tal de los árboles Trie. Este método de la línea 10 recibe

como entrada los nodos raíz para cada árbole Trie y un apuntador a una variable de tipo booleana, en donde se registrará el estado de la comparación.

- *currentNode*: Apuntador al nodo actual del primer árbol Trie en la exploración.
- *otherCurrentNode*: Apuntador al nodo actual del segundo árbol Trie en la exploración.
- *comparison*: Variable para almacenar el valor de verdad de la comparación.

código A.6: Método para Comparar dos árboles Trie

```
template<typename valueType>
bool TrieTree<valueType>::compare(TrieTree *otherTrie){
    bool compare1 = true, compare2 = true;
    compare(&rootNode,&(otherTrie->rootNode),&compare1);
    compare(&(otherTrie->rootNode),&rootNode,&compare2);
    return compare1 and compare2;
}

template<typename valueType>
void TrieTree<valueType>::compare(trieNode<valueType> **currentNode,trieNode<valueType> **
     otherCurrentNode,bool *comparison){
    if((*comparison) == false)
        return;

    if((*currentNode) == NULL and (*otherCurrentNode) != NULL){
        (*comparison) = false;
        return;
    }

    if((*currentNode) != NULL and (*otherCurrentNode) == NULL){
        (*comparison) = false;
        return;
    }

    if((*currentNode) == NULL and (*otherCurrentNode) == NULL){
        (*comparison) = true;
        return;
    }else{
        if((*currentNode)->value != (*otherCurrentNode)->value){
            (*comparison) = false;
            return;
        }
        if((*currentNode)->nextDim != NULL){
            if((*otherCurrentNode)->nextDim != NULL)
                compare(&(*currentNode)->nextDim,&(*otherCurrentNode)->nextDim,comparison);
            else{
                (*comparison) = false;
                return;
            }
        }else
            if((*otherCurrentNode)->nextDim != NULL){
                (*comparison) = false;
                return;
            }

        if((*comparison) == false)
            return;

        if((*currentNode)->nextTrieNode != NULL){
            if((*otherCurrentNode)->nextTrieNode != NULL)
                compare(&(*currentNode)->nextTrieNode,&(*otherCurrentNode)->nextTrieNode,comparison
     );
            else{
                (*comparison) = false;
                return;
            }
        }else
            if((*otherCurrentNode)->nextTrieNode != NULL){
                (*comparison) = false;
                return;
            }

        if((*comparison) == false)
            return;
    }
    (*comparison) = true;
}
```

En este método del código A.6, se presentan los siguientes posibles escenarios:

- Si en algún momento de la exploración uno de los nodos Trie es nulo y el otro nodo no lo es (líneas 14 y 19), entonces se establece la variable de estado a FALSE. Con ello, se retorna en la llamada recursiva y el método terminará su ejecución.
- Si ambos nodos Trie, para un momento de la exploración son nulos, entonces se establece la variable de estado a TRUE (línea 24), ya que ambos nodos son nulos y por lo tanto son iguales.
- Si el valor de los nodos actuales en la exploración son diferentes, entonces se establece la variable de estado a FALSE (línea 29). En este caso, se retorna en la llamada recursiva y el método detendrá su ejecución. En caso de que los valores sean iguales, se continua con la exploración.

Otro punto importante, es que este método realiza la comparación basándose en la exploración del primer árbol Trie. Sin embargo, esta exploración no siempre determinará si ambos árboles son iguales, ya que el segundo árbol Trie puede tener los mismos nodos que el primer árbol Trie y otros más. No obstante, para solucionar esta problemática, se debe realizar la comparación en el otro sentido. Con ello la exploración será guiada por el segundo árbol Trie, y por ende será posible hallar dichas diferencias.

A.4 Método para Clonar un Árbol Trie

El método para clonar un Trie se muestra en el código A.7. Este método genera nodos con los mismos valores y la misma configuración que el Trie original. Al igual que los casos anteriores, la exploración se realiza primero en profundidad. Como se observa, se tienen dos definiciones de este método, el de la línea 2 es la llamada principal y el de la línea 13 es el método que realiza la exploración de los nodos como tal. Los parámetros que recibe este método son:

- *prevNode*: Nodo previo del Trie original en la exploración. Se utiliza para establecer la configuración del Trie original en el nuevo Trie.
- *currentNode*: Nodo actual del Trie original en la exploración.
- *copyPrevNode*: Nodo previo del Trie nuevo. Este nodo se actualizará en base al nodo previo del Trie original. Ello para preservar la configuración y ordenamiento en los nodos.
- *copyCurrentNode*: Nodo actual del nuevo Trie.

código A.7: Método para clonar un árbol Trie

```
1  template<typename valueType>
2  TrieTree<valueType> *TrieTree<valueType>::clone(){
3      trieNode<valueType> *copyRootNode = NULL;
4      trieNode<valueType> *prevNode = NULL;
5      trieNode<valueType> *copyPrevNode = NULL;
6
7      clone(&prevNode,&rootNode,&copyPrevNode,&copyRootNode);
8      TrieTree *newTrie = new TrieTree(copyRootNode);
9      return newTrie;
10 }
11
12 template<typename valueType>
13 void TrieTree<valueType>::clone(trieNode<valueType> **prevNode,trieNode<valueType> **currentNode,
       trieNode<valueType> **copyPrevNode,trieNode<valueType> **copyCurrentNode){
14     if((*currentNode) == NULL){
15         return;
16     }else{
17         trieNode<valueType> *node = new trieNode<valueType>;
18         node->nextDim = NULL;
19         node->nextTrieNode = NULL;
20         node->value = (*currentNode)->value;
21         (*copyCurrentNode) = node;
22
23         if((*prevNode) != NULL)
24             if((*prevNode)->nextDim == (*currentNode)){
25                 (*copyPrevNode)->nextDim = (*copyCurrentNode);
26             }else{
```

```
27                  (*copyPrevNode)->nextTrieNode = (*copyCurrentNode);
28              }
29
30          if((*currentNode)->nextDim != NULL)
31              clone(currentNode,&(*currentNode)->nextDim,copyCurrentNode,&(*copyCurrentNode)->nextDim
    );
32
33          if((*currentNode)->nextTrieNode != NULL){
34              clone(currentNode,&(*currentNode)->nextTrieNode,copyCurrentNode,&(*copyCurrentNode)->
    nextTrieNode);
35          }
36      }
37  }
```

El método del código A.7 realiza la exploración primero en profundidad del Trie original. En la línea 14 se evalúa si se ha llegado a un punto en donde el nodo actual es nulo, en este es el caso base de la llamada recursiva, ya que se detiene cuando se alcanza este punto. En otro caso se realiza la clonación como tal del contenido y configuración del nodo actual (líneas 17- 28). Por último en las líneas 30 y 33 se muestran las condiciones para continuar con la exploración en profundidad.

A.5 Eliminación de Vértices en un Árbol Trie

En el código A.8 se muestra el método para eliminar un vértice de un árbol Trie. El parámetro del vértice a eliminar se recibe en forma de arreglo. La exploración se realiza primero en profundidad, y se buscan los nodos que coincidan con los elementos del arreglo. Como se observa, se tienen dos definiciones de este método, en la línea 2 se tiene el método principal y en la línea 8 se tiene el método que realiza la búsqueda y eliminación de los nodos del vértice dado. Los parámetros de entrada son los siguientes:

- *prevNode*: Nodo previo en la exploración.
- *currentNode*: Nodo actual en la exploración.
- *key*: Arreglo que contiene el vértice que se desea eliminar.
- *currentDim*: Dimensión actual en la exploración.

código A.8: Método para eliminar un vértice del Trie

```
1  template<typename valueType>
2  void TrieTree<valueType>::removeVertex(valueType *key){
3      trieNode<valueType> * prevNode = NULL;
4      removeVertex(&prevNode,&rootNode,key,0);
5  }
6
7  template<typename valueType>
8  bool TrieTree<valueType>::removeVertex(trieNode<valueType> **prevNode,trieNode<valueType> **
       currentNode,valueType *key,int currentDim){
9      if((*currentNode) == NULL)
10         return true;
11
12     if((*currentNode)->value > key[currentDim])
13         return false;
14
15     if((*currentNode)->value == key[currentDim])
16     {
17         bool vertexStatus = removeVertex(currentNode,&(*currentNode)->nextDim,key,currentDim+1);
18         if(vertexStatus){
19             if(rootNode == (*currentNode)){
20                 if(rootNode->nextTrieNode != NULL){
21                     trieNode<valueType> *tempNode = rootNode;
22                     rootNode = rootNode->nextTrieNode;
23                     delete tempNode;
24                     tempNode = NULL;
25                 }else{
26                     rootNode == NULL;
27                     delete *currentNode;
28                     *currentNode = NULL;
29                 }
30
31                 return false;
```

```
32              }else{
33                  if((*prevNode)->nextDim == (*currentNode))
34                  {
35                      if((*currentNode)->nextTrieNode == NULL)
36                      {
37                          delete *currentNode;
38                          *currentNode = NULL;
39                          return true;
40                      }else
41                      {
42                          trieNode<valueType> *node = (*currentNode)->nextTrieNode;
43                          delete *currentNode;
44                          *currentNode = NULL;
45                          (*prevNode)->nextDim = node;
46                          return false;
47                      }
48                  }else
49                  {
50                      if((*currentNode)->nextTrieNode == NULL)
51                      {
52                          delete *currentNode;
53                          *currentNode = NULL;
54                      }else
55                      {
56                          trieNode<valueType> *node = (*currentNode)->nextTrieNode;
57                          delete *currentNode;
58                          *currentNode = NULL;
59                          (*prevNode)->nextTrieNode= node;
60                      }
61                      return false;
62                  }
63              }
64          }
65      }
66
67      if((*currentNode)->nextTrieNode != NULL)
68          return removeVertex(currentNode,&(*currentNode)->nextTrieNode,key,currentDim);
69
70      return false;
71  }
```

En el método A.8 se plantean cuatro posibles escenarios durante la exploración:

- Si en la exploración se llega al final del Trie, entonces el arreglo del vértice a eliminar si existe en el Trie y por lo tanto se retorna el valor *true* que indica que se debe intentar eliminar todos los nodos hallados durante la exploración, esta condición se evalúa en la línea 9. Cabe mencionar que como ya se explicó anteriormente, los vértices en el Trie pueden compartir nodos comunes, por ello se hace énfasis en que se intentará eliminar los nodos hallados.
- Cuando hay un nodo con un valor mayor que el elemento buscado, para la dimensión actual en la exploración, significa que el vértice buscado no existe (línea 12). Ello debido a que los nodos en el Trie están ordenados, consecuentemente si existe un nodo con valor mayor al valor buscado quiere decir que en los nodos previos no se halló una coincidencia. Por lo tanto la exploración termina y no se eliminan los elementos del vértice.
- Si durante la exploración se halla un nodo cuyo valor coincide con el elemento actual del arreglo (línea 15), entonces se espera el resultado de la ejecución de este método en la siguiente llamada recursiva para proceder a eliminar el nodo actual, ya que el nodo hallado es un candidato a ser eliminado. En el caso dado que el nodo actual se deba intentar eliminar (condición de la línea 17), se presentan dos situaciones adicionales:
  1. Si el nodo actual corresponde al nodo raíz del Trie, entonces se eliminará solo si éste no hace referencia a algún otro nodo (líneas 19-31).
  2. Por otro lado, cuando no se trata de un nodo hoja, entonces se evalúa si éste puede ser eliminado (condición de la línea 33). Ya que pueden existir otros nodos en la siguiente dimensión asociados con el nodo en cuestión. Si la lista de nodos de la siguiente dimensión del nodo actual esta vacía, entonces si se podrá eliminar.
  3. Si el nodo actual, que tiene un valor igual al buscado, es un nodo hoja entonces se elimina

(línea 50). Ya que en la siguiente llamada recursiva se llega al final del Trie con respecto a este nodo. La variable *vertexStatus* recibe un valor TRUE, con lo que se procede a eliminar este nodo.

- Por último, cuando no se halla una coincidencia en el nodo actual de la exploración, entonces se explora otro nodo en la misma dimensión (línea 67). Considerando que este nodo no sea el último elemento de la lista actual. Lo que significaría que no existe una coincidencia en la dimensión actual, con lo que se terminaría la exploración.

A.6 Operación XOR Regularizada

En este método se realiza la operación XOR regularizada, la cual consiste en tomar dos árboles Trie y aplicar la operación XOR a sus vértices, por ello en general se hace referencia a este método como *mergeXOR*. Esto significa que se eliminarán todos los vértices que sean comunes a ambos Tries. Este es otro enfoque al presentado en el algoritmo de Operaciones Booleanas (sección 3.7.1), ya que en este caso no se realiza el cálculo de Couplets y Secciones, sino se comparan directamente los vértices extremos de cada Trie.

En el código A.9, se muestra el método para realizar la operación mergeXOR regularizada. Como se observa, se tienen dos definiciones, en la línea 2 se tiene la llamada principal del método y en la línea 15 se presenta el método para la exploración de los vértices en el segundo Trie. La idea general del método mergeXOR consiste en clonar el Trie actual (línea 4) y pasarlo como la variable *resultTrie*, y posteriormente se realiza la exploración del segundo Trie para obtener sus vértices extremos, los cuales serán insertados en resultTrie. Como se puede notar, el método mergeXOR se apoya en la función *insertVertex* presentada en el código A.5 (línea 17), ya que los vértices comunes a ambos Tries serán eliminados de resultTrie.

código A.9: Operación XOR regularizada

```
1  template<typename valueType>
2  TrieTree<valueType> * TrieTree<valueType>::mergeXOR(TrieTree *otherTrie){
3      int dim = dimDepth();
4      TrieTree *xorTrie = clone();
5      valueType *key = new valueType[dim];
6
7      mergeXOR(&xorTrie,&(otherTrie->rootNode),&key,0);
8
9      delete [] key;
10
11     return xorTrie;
12 }
13
14 template<typename valueType>
15 void TrieTree<valueType>::mergeXOR(TrieTree **resultTrie,trieNode<valueType> **currentNode,
       valueType **key,int dim){
16     if((*currentNode) == NULL){
17         (*resultTrie)->insertVertex((*key),dim);
18         return;
19     }
20
21     (*key)[dim] = (*currentNode)->value;
22
23     mergeXOR(resultTrie,&(*currentNode)->nextDim,key,dim+1);
24
25     if((*currentNode)->nextTrieNode != NULL)
26         mergeXOR(resultTrie,&(*currentNode)->nextTrieNode,key,dim);
27 }
```

Bibliografía

[1] K. Ngan and H. Li, *Video Segmentation and Its Applications*. SpringerLink : Bücher, Springer, 2011.

[2] T. Meier and K. Ngan, "Automatic segmentation of moving objects for video object plane generation," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 8, pp. 525–538, Sep 1998.

[3] R. Pérez-Aguila, "Representing and visualizing vectorized videos through the extreme vertices model in the n-dimensional space nd-evm," *Journal Research in Computer Science*, vol. 29, pp. 65–80, 2007.

[4] K. N. Ngan and H. Li, "Semantic object segmentation," *IEEE Communications Society Multimedia Communications Technical Committee E-Letter*, vol. 4, no. 6, pp. 6–8, 2009.

[5] K.-S. Fu and J. Mui, "A survey on image segmentation," *Pattern recognition*, vol. 13, no. 1, pp. 3–16, 1981.

[6] S. Bhattacharyya and U. Maulik, *Soft Computing for Image and Multimedia Data Processing*. SpringerLink : Bücher, Springer Berlin Heidelberg, 2013.

[7] S. Bhattacharyya, U. Maulik, and P. Dutta, "Multilevel image segmentation with adaptive image context based thresholding," *Applied Soft Computing*, vol. 11, no. 1, pp. 946–962, 2011.

[8] T. Moeslund, *Introduction to Video and Image Processing: Building Real Systems and Applications*. Undergraduate Topics in Computer Science, Springer, 2012.

[9] Y. Zhang, *Advances in Image and Video Segmentation*. IGI Global research collection, IGI Global, 2006.

[10] A. Aguilera, *Orthogonal polyhedra: study and application*. PhD thesis, Universitat Politècnica de Catalunya, 1998.

[11] R. Pérez-Aguila, *Orthogonal polytopes: study and application*. PhD thesis, Universidad de las Américas, Puebla UDLAP, 2006.

[12] R. Pérez-Aguila, "Modeling and manipulating 3d datasets through the extreme vertices model in the n-dimensional space (nd-evm)," *Research in Computer Science*, vol. 31, pp. 15–24, 2007.

[13] R. Pérez-Aguila, "Towards a new approach for modeling volume datasets based on orthogonal polytopes in four-dimensional color space," *Engineering Letters*, vol. 18, no. 4, p. 326, 2010.

[14] R. Pérez-Aguila, "Efficient boundary extraction from orthogonal pseudo-polytopes: An approach based on the nd-evm," *Journal of Applied Mathematics*, vol. 2011, 2011.

[15] R. Pérez-Aguila, "Computing the discrete compactness of orthogonal pseudo-polytopes via their nd-evm representation," *Mathematical Problems in Engineering*, vol. 2010, 2010.

[16] T. Kohonen, *Self-Organizing Maps*. Physics and astronomy online library, Springer Berlin Heidelberg, 2001.

[17] S. Haykin, *Neural Networks and Learning Machines*. Neural networks and learning machines, Prentice Hall, 3rd ed., 2009.

[18] S. Samarasinghe, *Neural Networks for Applied Sciences and Engineering: From Fundamentals to Complex Pattern Recognition*. Taylor & Francis, 2006.

[19] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE*, vol. 78, pp. 1464–1480, Sep 1990.

[20] S. Theodoridis and K. Koutroumbas, *Pattern Recognition*. Elsevier Science, 2008.

[21] R. Pérez-Aguila, "Enhancing brain tissue segmentation and image classification via 1d kohonen networks and discrete compactness: an experimental study," *Engineering Letters*, vol. 21, no. 4, pp. 171–180, 2013.

[22] E. Bribiesca, "Measuring 2-d shape compactness using the contact perimeter," *Computers & Mathematics with Applications*, vol. 33, no. 11, pp. 1–9, 1997.

[23] R. S. Montero and E. Bribiesca, "State of the art of compactness and circularity measures," in *International Mathematical Forum*, vol. 4, pp. 1305–1335, 2009.

[24] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[25] L. Roberts and J. Tippett, "Machine perception of three-dimensional solids, optical and electro-optical information processing, 1965."

[26] S.-Y. Wan and W. E. Higgins, "Symmetric region growing," *Image Processing, IEEE Transactions on*, vol. 12, no. 9, pp. 1007–1015, 2003.

[27] R. M. Haralick and L. G. Shapiro, "Image segmentation techniques," *Computer vision, graphics, and image processing*, vol. 29, no. 1, pp. 100–132, 1985.

[28] I. Kokkinos and P. Maragos, "Synergy between object recognition and image segmentation using the expectation-maximization algorithm," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 31, no. 8, pp. 1486–1501, 2009.

[29] C. Gentile, O. Camps, and M. Sznaier, "Segmentation for robust tracking in the presence of severe occlusion," *Image Processing, IEEE Transactions on*, vol. 13, pp. 166–178, Feb 2004.

[30] B. Ko and H. Byun, "Frip: a region-based image retrieval tool using automatic image segmentation and stepwise boolean and matching," *Multimedia, IEEE Transactions on*, vol. 7, pp. 105–113, Feb 2005.

[31] V. Mezaris, I. Kompatsiaris, N. Boulgouris, and M. Strintzis, "Real-time compressed-domain spatiotemporal segmentation and ontologies for video indexing and retrieval," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 14, pp. 606–621, May 2004.

[32] T. Meier and K. N. Ngan, "Video segmentation for content-based coding," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 9, no. 8, pp. 1190–1203, 1999.

[33] C. L. Zitnick and S. B. Kang, "Stereo for image-based rendering using image over-segmentation," *International Journal of Computer Vision*, vol. 75, no. 1, pp. 49–65, 2007.

[34] D. Chai and K. Ngan, "Face segmentation using skin-color map in videophone applications," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 9, pp. 551–564, Jun 1999.

[35] L.-K. Liu, "Model-based video segmentation for vision-augmented interactive games," in *Electronic Imaging*, pp. 432–439, International Society for Optics and Photonics, 2000.

[36] M. Spivak, *Calculus On Manifolds: A Modern Approach To Classical Theorems Of Advanced Calculus*. Westview Press, 1971.

[37] S. Marchand-Maillet and Y. M. Sharaiha, *Binary digital image processing: a discrete approach*. Academic Press, 1999.

[38] E. Bribiesca, "A measure of compactness for 3d shapes," *Computers & Mathematics with Applications*, vol. 40, no. 10, pp. 1275–1284, 2000.

[39] R. Pérez-Aguila, *Una Introducción al Cómputo Neuronal Artificial*. El Cid Editor, 2012.

[40] S. American, *Mind and Brain: Readings from Scientific American Magazine*. W.H. Freeman, 1993.

[41] R. Colom, S. Karama, R. E. Jung, and R. J. Haier, "Human intelligence and brain networks," *Dialogues in clinical neuroscience*, vol. 12, no. 4, p. 489, 2010.

[42] J. Einenkel, U.-D. Braumann, L.-C. Horn, N. Pannicke, J.-P. Kuska, A. Schütz, B. Hentschel, and M. Höckel, "Evaluation of the invasion front pattern of squamous cell cervical carcinoma by measuring classical and discrete compactness," *Computerized Medical Imaging and Graphics*, vol. 31, no. 6, pp. 428–435, 2007.

[43] Z.-P. Lo, M. Fujita, and B. Bavarian, "Analysis of neighborhood interaction in kohonen neural networks," in *Parallel Processing Symposium, 1991. Proceedings., Fifth International*, pp. 246–249, IEEE, 1991.

[44] Z.-P. Lo, Y. Yu, and B. Bavarian, "Analysis of the convergence properties of topology preserving neural networks," *Neural Networks, IEEE Transactions on*, vol. 4, pp. 207–220, Mar 1993.

[45] H. Ritter, T. Martinetz, and K. Schulten, *Neural Computation and Self-organizing Maps: An Introduction*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1992.

[46] R. Ruiz-Rodríguez, "Implementación del evm (extreme verticesmodel) en java," Master's thesis, Universidad de las Américas, Puebla UDLAP, 2002.

[47] R. Ruiz-Rodríguez, "A 3d editor for orthogonal polyhedra based on the extreme vertices model," in *Décimo Congreso Internacional de Investigación en Ciencias Computacionales CIICC*, vol. 3.

[48] M. Nixon, *Feature Extraction and Image Processing*. Elsevier Science, 2013.

[49] P. Deitel and H. Deitel, *Java How to Program, 9th Edition: Java How to Program, 9th Edition Deitel*. Fonenix inc, 2011.

[50] AfricanSkyCAM, "Dji phantom aerial tour of the amboseli ecosystem," 2014. [En línea: `https://www.youtube.com/watch?v=BMrsy9C_re8`; accesado el día 7-Septiembre-2015].

[51] R. Cucchiara, C. Grana, M. Piccardi, and A. Prati, "Detecting moving objects, ghosts, and shadows in video streams," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 25, no. 10, pp. 1337–1342, 2003.

[52] K. Maly, "Compressed tries," *Communications of the ACM*, vol. 19, no. 7, pp. 409–415, 1976.

[53] Pixar, "Jack-jack attack," 2005. [En línea: `https://en.wikipedia.org/wiki/Jack-Jack_Attack`; accesado el día 7-Septiembre-2015].

[54] PoolShot.org, "Trickshots for beginners #3 - bilyar - pool trick shot & artistic billiard training lesson," 2014. [En línea: `https://www.youtube.com/watch?v=vkrfc65vanY`; accesado el día 7-Septiembre-2015].

[55] A. Prati, I. Mikic, M. M. Trivedi, and R. Cucchiara, "Detecting moving shadows: algorithms and evaluation," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 25, no. 7, pp. 918–923, 2003.

[56] M. C. C. Video, "Nione security megapixel cctv camera nv-nd752m-e [traffic]," 2010. [En línea: `https://www.youtube.com/watch?v=ukMFR0IQ3Yc`; accesado el día 7-Septiembre-2015].