UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

TraceMon: Sistema multiagente para Tracking multicámara en entornos Monitorizados

David García Bermejo

Febrero, 2012

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

TraceMon: Sistema multiagente para Tracking multicámara en entornos Monitorizados

Autor: David García Bermejo

Director: Javier Alonso Albusac Jiménez

Tutor: Carlos González Morcillo

Febrero, 2012

David García Bermejo

E-mail: davidgber10@gmail.com

TRIBUNAL:

Presidente:

Vocal 1:

Vocal 2:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	VOCAL 1	VOCAL 2	SECRETARIO
Fdo.:	Fdo.:	Fdo.:	Fdo.:

A mis padres y a mi Hermano, por su apoyo y preocupación, y a Fátima, por ayudarme moralmente cuando más lo necesitaba.

Resumen

La Vigilancia Inteligente (VI) emplea técnicas, métodos y algoritmos de Inteligencia Artificial sobre sistemas de monitorización de entornos. El principal objetivo de los Sistemas de Vigilancia Inteligente (SVI) puede definirse como la detección, clasificación y seguimiento (análisis de trayectorias y comportamiento) de objetos de forma automática.

En la actualidad existen diversos métodos de segmentación y *tracking* que ofrecen diferentes ventajas e inconvenientes. Así, es necesario elegir los métodos que mejor se adapten a cada entorno de ejecución, ofreciendo los mejores resultados en detección de objetos, posicionamiento 3D y evasión de oclusiones totales o parciales.

Como solución a este problema y, con el objetivo de desarrollar un SVI centrado sobre las capas de *segmentación* y *tracking* de una arquitectura de SVI multi-cámara genérica, se propone el presente proyecto fin de carrera: TraceMon, un sistema modular basado en el paradigma de diseño multi-agente que permite homogeneizar los dispositivos de entrada (archivos de vídeo, cámaras locales o remotas), y proporcionando una interfaz de usuario para un SVI multicámara.

TraceMon en su etapa de calibración, obtiene los parámetros intrínsecos (distancia focal, tamaño de píxeles y centro de la imagen) y extrínsecos (posición y rotación 3D) de cada cámara. Así, el sistema desarrollado es capaz de realizar un seguimiento en el espacio 3D del entorno que está monitorizando. TraceMon está construido empleando herramientas y estándares libres, lo que facilita la exportación de eventos (tales como detección de personas o vehículos) para que puedan ser analizados por otros sistemas.

Abstract

The Intelligent Surveillance(IS) uses techniques, methods and algorithms of Artificial Intelligence on environment monitoring systems. The main objective of the Intelligent Surveillance Systems (ISS) can be defined as the automatic detection, classification and monitoring (analysis the trajectories and behavior) of objects.

Nowadays, there are several segmentation and tracking methods with different advantages and disadvantages. Thus, it is necessary to choose the best methods in each environment, offering the finest results in the detection of objects, 3D positioning and eluding partial or total occlusions.

To solve this problem, and focusing on the tracking and segmentation layers of a generic ISS, this project called TraceMon is proposed. TraceMon is a modular system based on the paradigm of multi-agent design which eases the homogeneization of input devices (video files and local or remote cameras), and provides an user interface for a multi-camera ISS.

At the calibration stage, TraceMon gets the intrinsic (focal length, pixel size and image center) and extrinsic (position and 3D rotation) parameters of each camera. In this manner, the developed system is capable of tracking directly in the 3D environment. TraceMon is built using free software tools and open standards, making easy the exportation of events (such as the detection of people or vehicles) that can be analyzed by other external systems.

Índice general

Índice de figuras

Índice de listados

Lista de algoritmos

Agradecimientos

En primer lugar quisiera agradecer a mis padres y hermano que siempre me han apoyado y me han ayudado a luchar en los momentos más difíciles. También a los amigos que he tenido durante estos años de carrera, gracias por vuestra atención y amistad. En especial mención a mi amiga Fátima, por comprenderme y alegrarme en las situaciones más críticas. Y por último y no menos importante, quisiera agradecer a Carlos González Morcillo y Javier Alonso Albusac Jiménez por brindarme la oportunidad de realizar un proyecto con vosotros, gracias por vuestra atención, dedicación y compromiso.
Muchas gracias a todos.

1

Introducción

A lo largo de los últimos años se ha producido un crecimiento progresivo en el uso de sistemas de vigilancia, de tal forma que se considera una alternativa muy eficaz para controlar y reducir el índice de delitos cometidos en espacios públicos. Con esta fuerte demanda, se requiere de más personal para atender a la monitorización de las imágenes. Las empresas de seguridad buscan dotar de inteligencia a sus sistemas, con el objetivo de apoyar al personal de seguridad y mejorar la eficiencia en las tareas de vigilancia.

Una característica común de los Sistemas de Vigilancia Inteligente (SVI) es que para analizar si un objeto se está comportando correctamente, previamente se debe detectar (empleando técnicas de segmentación) y realizar su seguimiento (mediante métodos de *tracking*). Existen multitud de técnicas de *segmentación* y *tracking* con sus ventajas e inconvenientes asociados. Otro de los principales problemas es que la mayoría de los SVI no son entornos multicámara, de modo que son más sensibles a errores por oclusión total o parcial.

Con el propósito de resolver estos problemas, se propone el presente proyecto fin de carrera, TraceMon, un sistema multi-agente que permite una monitorización multicámara de entornos mediante el uso de técnicas de *segmentación* y de *tracking*.

1.1. Qué es un Sistema de Vigilancia Inteligente

Un **"Sistema de Vigilancia Inteligente"** es aquel que emplea técnicas, métodos y algoritmos de Inteligencia Artificial sobre sistemas de monitorización de vigilancia.

Según [Fer09], la mayoría de los SVI presentan una arquitectura abstracta, formada por 5 capas (figura 1.1) :

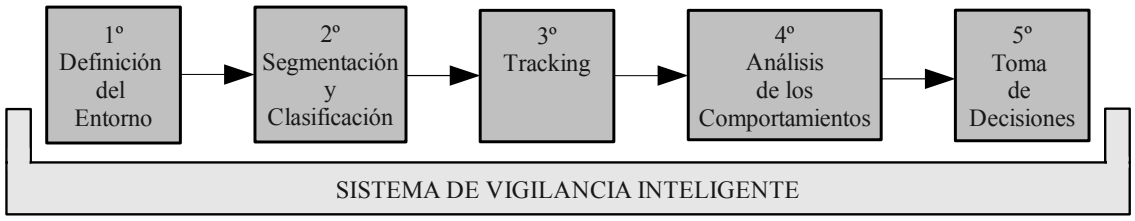

FIGURA 1.1: Arquitectura abstracta de un Sistema de Vigilancia Inteligente.

1. **Definición del entorno.**

   En esta primera capa se determina cuál va a ser el entorno y los elementos donde se va a desplegar el sistema de vigilancia inteligente.

2. **Segmentación y clasificación.**

   Una vez fijado el entorno, comienza esta capa donde se deberán detectar cuáles son los objetos que intervienen en la escena captada por la cámara y hacer una clasificación de éstos.

3. **Tracking o seguimiento.**

   En esta capa, el sistema debe hacer un seguimiento de los objetos que han sido detectados por la capa de segmentación y clasificación.

4. **Análisis de los comportamientos.**

   Una vez obtenida la información que indica los puntos del espacio por los que el objeto se ha estado moviendo, ésta debe de ser procesada en esta capa para realizar un estudio de cómo se ha comportado ese objeto y si se corresponde con los patrones de comportamiento que tiene asociado dicho objeto por su naturaleza.

5. **Toma de decisiones.**

   En esta última capa se elaboran las decisiones teniendo en cuenta la información procesada en las anteriores capas. Las decisiones típicas que se suelen abordar son activar alarmas, cerrar puertas, accionar mecanismos para controlar la incidencia, etc.

1.1.1. Evolución e impacto socio-económico

Hoy en día, se puede encontrar el uso de sistemas de vigilancia en multitud de espacios públicos (hoteles, hospitales, residencias, bancos, etc), en privados (hogares) y en medios de transporte (estaciones de tren, aeropuertos, carreteras, etc) entre otros. Esta fuerte demanda crea la necesidad de desarrollar aplicaciones de monitorización inteligente. Según la consultora *iSuppli Corp*[1], los ingresos globales aportados por sistemas de vigilancia tendrán un crecimiento del 13,2 % hasta llegar a superar los 9.000 millones a finales del 2011.

Según [VV05], existen tres generaciones diferentes:

- **1ª generación.** En esta generación, aparecen los primeros sistemas CCTV (Circuito Cerrado de Televisión) analógicos. Estos sistemas constan de un conjunto de cámaras conectadas a una serie de monitores, generalmente ubicados en una unidad de control. Aunque en algunas situaciones tienen un buen rendimiento, surgen problemas en el uso de las técnicas analógicas para la distribución y almacenamiento de las imágenes. Con el objetivo de corregir estos problemas, comienzan las investigaciones en el campo de la digitalización.

- **2ª generación.** Aparecen las primeras técnicas de vigilancia automatizada donde se combina la visión por computador y los sistemas de 1ª generación. Los beneficios de éstos comienzan a notarse, pero ante determinadas circunstancias no satisfacen las necesidades o requisitos. Se requieren nuevas aproximaciones, capaces de obtener algoritmos mucho más robustos para la visión por computador en tiempo real, mejores patrones de comportamientos y aprendizaje de las variaciones del entorno, y un acercamiento entre el análisis automatizado de la escena y su interpretación en el lenguaje natural.

- **3ª generación.** Se desarrollan técnicas robustas para una amplia gama de sistemas de vigilancia. Aparecen problemas en la distribución de la información, se requiere mucho más almacenamiento y hay una gran variedad de plataformas de sensores. Con el objetivo de resolver estos problemas, actualmente se está investigando sobre el uso de sistemas distribuidos en lugar de centralizados en este ámbito, en la compactación y fusión de los datos, en un framework de razonamiento probabilístico y en técnicas de vigilancia para sistemas multicámara.

[1] `http://www.infosecurityvip.com/newsletter/estadisticas_may11.html`

1.1.2. Problemática

El ámbito de este proyecto está centrado en las capas de segmentación, clasificación y de *tracking* (figura 1.1). En dichas capas existen multitud de problemas provenientes de la infinidad de situaciones que se pueden dar. Comenzando por la capa de segmentación, el principal problema es cómo poder detectar aquellos objetos que se están moviendo. Otro problema ligado a la detección de movimientos es que en los entornos reales se presentan movimientos constantes (las ramas de los árboles moviéndose, paneles luminosos, el propio agua de la lluvia, etc). Estos movimientos pueden considerarse "ruido" visual y deben ser ignorados en la capa de segmentación .

En la siguiente capa, se presentan problemas a la hora de clasificar correctamente los objetos. La correcta clasificación de un objeto requiere el modelado de las propiedades específicas de los objetos en base a las áreas en movimiento de la escena. Si el movimiento detectado no engloba totalmente al objeto, sino que lo recoge parcialmente se puede clasificar en un tipo distinto al que es en realidad. Este tipo de situaciones pueden dar lugar a problemas en la correcta clasificación de objetos.

En cuanto a la capa de seguimiento, existen problemas de precisión a la hora de obtener la posición del objeto identificado. Esta inexactitud, se puede dar en situaciones como:

- Peatones que se muevan con una distancia de separación entre ellos significativa y que dicha distancia se reduzca posteriormente a un valor cercano a cero. El sistema pasaría de seguir dos objetos independientes a uno mucho más grande (por ejemplo, como consecuencia de la situación explicada anteriormente).
- Que el sistema esté siguiendo un objeto y en ese momento este objeto se separe en varios objetos.
- En casos de detección parcial, el posicionamiento del objeto tendrá asociado un error mayor. Por ejemplo, si un peatón es detectado de cintura para arriba según la perspectiva de la cámara la estimación de su posición será errónea respecto a la real.
- En sistemas multi-cámara otro de los problemas que se dan es que el número de objetos detectados por las cámaras puede ser distinto, pero en realidad no se sabe si es fruto de algún error en la detección de los movimientos o se debe a que en cierta vista si existen más objetos debido al posicionamiento de las cámaras.
- Cuando los objetos se cruzan en sus trayectorias. Estas situaciones pueden dar lugar a errores en la posterior asignación de los identificadores de los objetos.

- Cuando se dan oclusiones totales de los objetos, en los que éstos desaparecen de la escena y pueden volver a aparecer.
- En la mayoría de los entornos existen zonas donde la visibilidad de las cámaras es peor que en otras. Esta reducción de la visibilidad puede introducir cierta incertidumbre en las posiciones de los objetos y también a la hora de clasificar y detectar los movimientos.

1.2. Objetivo principal

Teniendo en cuenta los problemas anteriores, el objetivo principal de este proyecto de fin de carrera **TraceMon**, es construir un sistema centrado en las capas *segmentación* y *tracking* de cualquier arquitectura genérica para la vigilancia inteligente multi-cámara. Con este propósito se estudiarán los principales algoritmos de segmentación para posteriormente implementar métodos para la detección de personas y vehículos.

Para solventar los problemas expuestos en la sección 1.1.2, el sistema que se va a desarrollar será multicámara para reducir en la mayor medida de lo posible el número de situaciones que provocan estos errores de precisión, y que suponen uno de los principales problemas en los SVI actuales. Al ser un sistema multicámara, para que la información obtenida de cada una de las cámaras pueda ser tratada como un conjunto común, el sistema a desarrollar tendrá la capacidad de obtener la localización 3D de los objetos que están captando las cámaras. De esta forma, empleando una función de distancia será posible decidir si un objeto es el mismo que están percibiendo diferentes cámaras.

TraceMon se desarrollará con el objetivo de ser una aplicación escalable y compatible con futuras versiones. Puesto que la vigilancia abarca una gran variedad de entornos, está aplicación se podría comunicar con nuevos algoritmos diseñados para detectar objetos abandonados, reconocimiento facial de personas, etc, así como en diversas aplicaciones centradas en otras capas del sistema vigilancia.

1.3. Estructura del documento

El presente documento ha sido redactado siguiendo la normativa del proyecto de fin de carrera de la Escuela Superior de Informática de la universidad de Castilla-La Mancha. Éste consta de los siguientes capítulos:

Capítulo 1: Introducción

En este capítulo se ha realizado una breve introducción del ámbito en el que se encuadra *TraceMon*.

Capítulo 2: Objetivos

En este capítulo se encuentran los objetivos que se pretenden conseguir en el desarrollo de este proyecto fin de carrera.

Capítulo 3: Antecedentes

En este capítulo se explican las bases matemáticas y el funcionamiento general de los principales sistemas de vigilancia inteligente, haciendo especial hincapié en las capas de segmentación y tracking.

Capítulo 4: Metodología de Desarrollo y Herramientas

En este capítulo se recoge la metodología utilizada para desarrollar este proyecto, junto con las herramientas necesarias para llevarlo a cabo.

Capítulo 5: Arquitectura de TraceMon

En este capítulo se recoge la descripción detallada de la elaboración del proyecto siguiendo la metodología elegida. En cuanto a la descripción de la arquitectura, ésta sigue un enfoque Top-Down, donde se pueden apreciar todos los módulos y submódulos que componen la solución.

Capítulo 6: Evolución y Resultados

En este capítulo se detallan las iteraciones, evolución, los problemas y los resultados que se han producido durante el desarrollo de este proyecto.

Capítulo 7: Conclusiones y Propuestas

En este capítulo se detallan las conclusiones obtenidas de la elaboración del proyecto, las propuestas de trabajo futuras, y algunas conclusiones personales sobre el desarrollo del proyecto.

Bibliografía

En esta sección se indican las referencias que se han utilizado en la elaboración de este proyecto.

Anexo

En esta sección se recoge un manual de usuario para hacer uso del prototipo funcional desarrollado.

2

Objetivos

El objetivo principal de este proyecto de fin de carrera es desarrollar **TraceMon** un sistema de *tracking* multicámara ubicado en las capas de *segmentación* y *tracking* de forma que pueda ser utilizado en cualquier sistema de vigilancia inteligente. A partir de este objetivo principal se pueden describir un conjunto de subobjetivos funcionales específicos que serán detallados a continuación.

- **Soporte para la gestión de fuentes de vídeo heterogéneas**. El sistema será desarrollado para soportar el *flujo de stream* proveniente de cámaras locales, de cámaras remotas o de archivos de vídeo. Esto nos facilitará la realización de pruebas deterministas del sistema.
- **Representación del entorno 3D**. El sistema permitirá representar el entorno sobre el que se está monitorizando como un mundo 3D. En este mundo, el usuario podrá ver las ubicaciones de las cámaras del sistema y los objetos que se van detectando y siguiendo.
- **Cálculo del posicionamiento 3D**. El sistema permitirá realizar el proceso de posicionamiento tridimensional de las fuentes de vídeo para obtener su posición sobre el entorno que se está monitorizando y posteriormente realizar un seguimiento de los objetos en el espacio 3D.
- **Soporte para pruebas deterministas**. Se realizará el modelado de un entorno virtual con el objetivo de testear algunos de los módulos que componen el sistema.

- **Soporte para el calibrado**. El sistema permitirá realizar el proceso de calibrado a las fuentes de vídeo conectadas en el sistema de vigilancia. Este proceso de calibrado consiste en el cálculo de los parámetros intrínsecos de la fuente que determinan la distorsión de la lente y la matriz de la cámara.

- **Soporte para la definición de áreas I/O**. El sistema ofrecerá la posibilidad de poder definir áreas de entrada/salida de las vistas de las fuentes de vídeo, con el fin de predecir cuando los objetos entran o abandonan el entorno.

- **Soporte para la definición de zonas fiables**. El sistema permitirá la posibilidad de indicar la fiabilidad de las zonas que aparecen en las vistas de las fuentes de vídeo. Definiendo estas zonas fiables se asociará un nivel de confianza a cada zona de la imagen.

- **Soporte para la gestión y comunicación de agentes**. El sistema utilizará agentes para llevar acabo los procesos de segmentación, *tracking*, clasificación y análisis de la información.

- **Despliegue de Agentes de aprendizaje**. El sistema dispondrá de agentes en fase de aprendizaje para que aprendan a distinguir aquellos objetos que permanecen inmóviles o mantienen un movimiento constante como las ramas de los árboles.

- **Despliegue de Agentes de segmentación**. El sistema dispondrá de agentes ya entrenados en distinguir entre fondo-primer plano para que se encarguen de la parte de segmentación.

- **Despliegue de Agentes de *tracking***. El sistema tendrá un algoritmo de *tracking* que se base principalmente en el posicionamiento 3D de los objetos detectados para poder realizar su seguimiento. Este algoritmo de *tracking* estará asociado a los agentes responsables del *tracking*.

- **Despliegue de Agentes de clasificación**. El sistema tendrá agentes para llevar acabo la fase de clasificación utilizando la información proveniente de la fase de *tracking*.

- **Despliegue de Agentes de fusión**. El sistema dispondrá de un agente que se encargue de fusionar la información procesada por el resto de agentes en las fases de segmentación, *tracking* y clasificación.

- **Interfaz gráfica parcialmente desacoplada**. El sistema dispondrá de una interfaz gráfica donde el usuario pueda realizar los procesos de calibración, posicionamiento, definición de áreas y la monitorización de lo que están captando las cámaras que

forman el sistema de vigilancia. En la medida de lo posible los módulos de detección y *tracking* estarán desacoplados de la interfaz gráfica de modo que pudieran ser reutilizados en otros proyectos.

- **Formatos de intercambio**. El sistema permitirá exportar información sobre el calibrado, posicionamiento, procesos de aprendizaje y de *tracking* para que se puedan comunicar con aplicaciones desarrolladas por terceros.
- **Gestión de datos de configuración**. El sistema permitirá cargar cierta información para evitar repetir determinados procesos como la calibración, posicionamiento, etc.
- **Escalabilidad**. El sistema debe de ser escalable y estar desarrollado para permitir la conexión de un número ilimitado de dispositivos de captura que forman el sistema de vigilancia.
- **Estándares libres**. El sistema será desarrollado utilizando estándares y software libre con el propósito de que *TraceMon* sea distribuido empleando alguna licencia libre y se potencie el desarrollo de futuras versiones por parte de la comunidad de usuarios.

3

Antecedentes

En este capítulo se pretende explicar al lector los componentes y tecnologías que están involucradas en las capas de segmentación y de *tracking* de todo sistema de vigilancia inteligente.

3.1. Introducción general

En este primer apartado de los antecedentes, se recoge la amplia gama de aplicaciones de los Sistemas de Vigilancia Inteligentes, junto con las arquitecturas (sección 3.1.2) que hay en el mercado para dar soporte a éstos, y en la (sección 3.1.3) un análisis de los componentes que intervienen en las capas de segmentación y *tracking* de todo sistema de vigilancia inteligente.

3.1.1. Aplicaciones de los Sistemas de Vigilancia Inteligente

Con el propósito de reducir el número de delitos y hacer los entornos mucho más seguros, hoy en día las áreas de aplicación de los SVI es muy variada. De esta manera, se puede decir que un SVI se verá aplicado en cualquier entorno en el que se requiera un control de la seguridad pública o personal.

Algunas de las aplicaciones de SVI más destacadas son:

- **Entorno urbanístico.** Con el objetivo de controlar y reducir el número de incidencias de transeúntes en las ciudades, en algunos países como Reino Unido se han desplegado SVI en sus calles, parques, plazas, etc.

- **Transporte público.** A raíz de los atentados producidos por el terrorismo, los políticos reforzaron las medidas de seguridad, de tal forma que se instalaron SVI en cualquier medio de transporte público, tales como, aeropuertos, estaciones de tren, metro, autobuses, etc.

- **En tráfico o estaciones de servicios.** Con el objetivo de reducir el número de víctimas que se producen en las carreteras, autovías, autopistas; es cada vez mayor el número de SVI que se utilizan para detectar el comportamiento incorrecto de cualquier vehículo. A parte de las carreteras, en las estaciones de servicios y de peaje también se han instalado SVI con el fin de reducir el número de conductores a la fuga para evitarse el cobro de dicho servicio.

- **Zonas de aparcamiento, almacenamiento y restringidas.** Como el número de delitos y robos es alto en almacenes donde se almacenan productos de alta gama, las empresas han instalado SVI con el fin de aumentar la seguridad en dichas zonas. En entornos industriales donde la presencia del ser humano puede estar expuesta a peligros los SVI son fundamentales para advertir a la planta del riesgo de algún objeto que pueda ser dañado. Estos SVI a parte de detectar objetos, también pueden ser entrenados para el reconocimiento del personal autorizado y ser mucho más precisos en su labor.

- **Entornos militares.** En actividades militares, el papel de los SVI para el espionaje de los enemigos y control de los vehículos que se encuentran en las zonas del perímetro, supone una ayuda muy importante para evitar posibles ataques del enemigo.

3.1.2. Arquitecturas y *frameworks* de SVI

Según los objetivos del proyecto (sección 1.2), el estudio de las arquitecturas y *frameworks* se centrará sobre sistemas multi-agente, aunque una arquitectura distribuida también podría ser otra alternativa. A continuación se explican tres importantes *frameworks* para sistemas multi-agente [RBR06] y uno para arquitecturas distribuidas.

- **TuCSoN** (Tuple Centre Spread over the Network) es un framework basado en un modelo y una infraestructura que ofrece servicios programables para dar soporte a

la comunicación y coordinación de los agentes. Es un proyecto de código abierto desarrollado en Java (*http://tucson.sourceforge.net*) y se compone de:

- De una plataforma que será instalada en los host que vayan a convertirse en los nodos de la infraestructura.
- Un conjunto de APIs para que los agentes puedan acceder a los servicios.
- Un conjunto de herramientas para la depuración, monitorización y análisis del comportamiento de los agentes.
- Su núcleo utiliza tuProlog que es el motor Prolog integrado perfectamente con Java. Éste permite interpretar el lenguaje declarativo de los agentes.

- **JADE** (Java Agent DEvelopment Framework) es un framework para el desarrollo de aplicaciones multiagente distribuidas. Se compone de un conjunto de herramientas gráficas para depurar y testear, APIs para el uso de servicios y su principal objetivo es dar soporte a la interoperabilidad del estándar FIPA (Foundation for Intelligent Physical Agents), junto con la posibilidad de poder ser utilizado en sistemas de recursos limitados como dispositivos móviles, PDAs, etc. El código fuente es abierto y se puede descargar en (*http://jade.tilab.com*).

- **DESIRE** (DEsign and Specification of Interacting REasoning components) es un framework para el desarrollo de aplicaciones multiagente desde el punto de vista de una arquitectura compositiva. Todo agente que se despliegue con este framework, contiene tres partes:

  - La definición de los procesos de razonamiento. Los procesos de razonamiento son definidos por un conjunto de componentes que interaccionan entre sí. Estos componentes se pueden dividir en otros componentes. Los componentes primitivos serán aquellos que no se puedan descomponer en otros.
  - El conocimiento del agente.
  - La definición de las relaciones entre los procesos de razonamiento y el conocimiento del agente.

- **ZeroC Ice** (ZeroC Internet Communications Engine) es un middleware muy adecuado para desplegar y desarrollar aplicaciones de sistemas distribuidos. El objetivo de éste es hacer que la invocación a objetos remotos sea vista por parte del programador como si se tratase de una invocación a objetos locales. La forma de llevar a cabo esta enmascaración es mediante un proxy en la parte del cliente que actúe de intermediario entre el cliente y el objeto remoto al que se quiere invocar. Y

en la parte del servidor se necesita un esqueleto que traduzca los eventos que se van produciendo en la red a las correspondientes invocaciones de los métodos remotos. Esta traducción es necesaria debido a que la comunicación entre el cliente y servidor, no deja de ser una comunicación que sigue un protocolo de red como TCP, UDP, etc. Debido a que ZeroC ICE permite que el cliente y el servidor estén implementados en distintos lenguajes de programación. El lenguaje para definir la interfaz que debe de ser idéntica a ambas partes se define mediante Slice: lenguaje de especificación de interfaces para ZeroC ICE.

3.1.3. Áreas y campos relacionados

Antes de diseñar y desarrollar un SVI Multi-agente basado en las fases de segmentación y de tracking, se debe de realizar un estudio de áreas relacionadas para comprender el funcionamiento de éste. Las áreas relacionadas que se tratarán en las siguientes secciones son:

- **Sistemas Multi-Agente**, una rama de la inteligencia artificial. Se realizará un estudio del funcionamiento y los componentes que forman un sistema multiagente.
- **Marco matemático**. Se realizará un estudio de las operaciones con vectores, puntos 3D, y matrices con el objetivo de poder representar el espacio 3D.
- **Raytracer**. Se realizará un estudio del funcionamiento y de los componentes que forman un trazador de rayos, con el objetivo de obtener las coordenadas 3D de los objetos que son capturados en la imagen 2D.
- **Herramientas y técnicas de visión por computador**. Se realizará una exposición de algunas de las herramientas existentes que serán útiles para desarrollar este proyecto, junto con el estudio de técnicas de clasificación y tracking para implementar un SVI.

3.2. Sistemas Multi-Agente

La inteligencia artificial es una ciencia reciente que surgió al terminar la segunda guerra mundial. Ésta trata de crear entidades inteligentes, de forma que sean útiles en muchas áreas (simulaciones de enfermedades, demostraciones de teoremas, operaciones de cirugía, estrategias de ajedrez, etc). Los sistemas multi-agente constituyen un tipo de

entidad inteligente. Éstos están formados por un conjunto de agentes que trabajan de forma independiente, comunicándose entre ellos para alcanzar un objetivo común.

Un agente [yPN04] es una entidad que percibe el entorno a través de sensores (cámaras de video, medidores de temperatura, de presión, etc), y con la información obtenida de los sensores lleva a cabo un procesamiento autónomo y racional para poder actuar mediante actuadores en dicho entorno (figura 3.1).

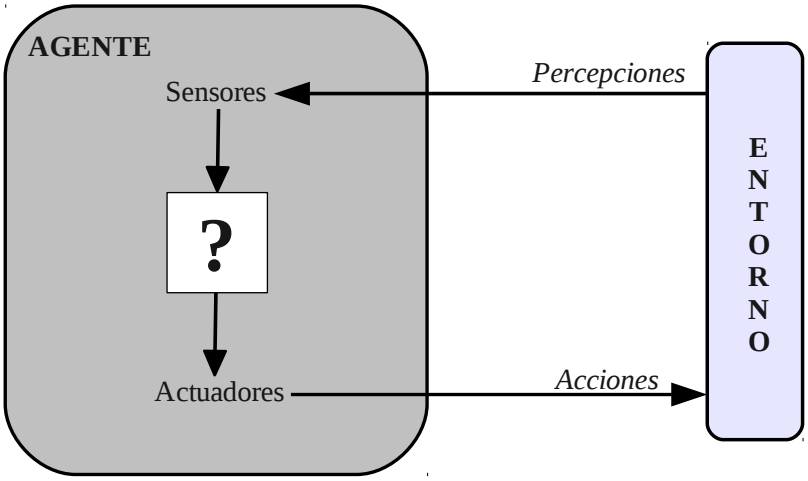

FIGURA 3.1: Definición de agente.

Si analizamos la figura 3.1, se puede ver que el agente recibe una serie de entradas (percepciones) y aplica a estas percepciones su conocimiento para obtener una serie de acciones. Luego todo conocimiento o comportamiento del agente viene definido en la **función del agente**. Esta función del agente se encarga de establecer una equivalencia de una percepción a una acción.

Para que el agente sea racional, es decir, tenga un buen comportamiento; su conocimiento debe de estar formado por cuatro factores:

- La definición del criterio de éxito. Esta definición le permitirá al agente distinguir qué acciones le llevaron a una situación correcta y cuáles no. Así podrá decidir sobre cuál acción es mejor.
- Conocimiento del entorno sobre el que está actuando. Toda información extra que se le pueda aportar al agente sobre el entorno en el que está actuando, le permitirá obrar mejor.
- El conjunto de acciones que puede realizar el agente.

- El conjunto de percepciones (entradas) que puede captar el agente.

En general, el comportamiento de un agente evoluciona de tal forma que en principio el agente no es autónomo. Esto es porque sus decisiones se basan en el conocimiento definido por el diseñador del agente. Pero a medida que va interaccionando con el entorno, éste irá aprendiendo hasta el punto que su conocimiento será independiente del definido por el diseñador.

A la hora de construir un agente [Dam05], otro de los factores importantes es tener una definición y a ser posible lo más completa sobre los entornos en los que va a trabajar el agente. Existen una serie de propiedades asociadas a los entornos, que nos permiten definirlos:

- **Observabilidad**. El entorno puede ser parcial o total. Se dice que es total, si el agente no tiene en cuenta estados intermedios, y a través de sus sensores es capaz de captar todos los aspectos que le llevan a tomar sus decisiones. Mientras que se dice de parcial, si los sensores del agente no son capaces de captar todos esos aspectos, debido a que existe ruido, éstos son imprecisos, etc.

- **Determinismo**. El entorno puede ser determinista o estocástico. Se habla de que el entorno es determinista, si el estado en el que se encuentra el entorno es el resultado de aplicar la acción del agente al estado anterior del entorno. Mientras se habla de estocástico, sino se cumple que el estado actual es el resultado del anterior y la acción del agente. Es decir, un entorno estocástico es aquel en el que hay un grado de incertidumbre sobre lo que puede pasar en el futuro, en resumen, no es predecible.

- **Temporal**. En cuanto a esta propiedad, el entorno puede ser episódico o secuencial. Se habla de un entorno episódico, si para elaborar la acción del agente, éste solo tiene en cuenta el episodio actual, y ninguno anterior a él. Mientras se habla de secuencial, si utiliza información de otros episodios anteriores.

- **Variabilidad**. Según esta propiedad el entorno puede ser estático o dinámico. Se dice que el entorno es dinámico, si el entorno puede cambiar mientras el agente razone para obtener una acción. En cuanto a estático, es cuando el entorno no varía, mientras el agente está razonando.

- **Estado**. Según el estado, el entorno puede ser discreto o continuo. Se habla de que un entorno es discreto, si todo posible comportamiento que sucede en el entorno, se puede expresar con un conjunto finito de estados. Si no se puede expresar con un conjunto finito de estados, entonces se habla de que el entorno es continuo.

A la hora de implementar un agente, se debe de tener en cuenta que existen cuatro posibles tipos de agentes.

- **Agente reactivo simple**. Este tipo de agente es el más sencillo que existe de implementar. Éste solo necesita la percepción actual por lo que no mantiene ningún histórico de percepciones para obtener una acción. El diseñador define una serie de *reglas de condición-acción* de tal forma que dada una percepción, se busca la regla que satisfaga dicha condición y se aplica su correspondiente acción.
- **Agente basado en modelos**. Este tipo de agente es más complejo y es utilizado en entornos parcialmente observables. Consiste en que el agente, aparte de lo que está percibiendo, también almacene un estado interno que contenga un histórico de las percepciones anteriores, de manera que ante una situación de oclusión, el agente pueda completar lo que no ve con el estado interno. Para actualizar la información del estado interno, se debe de definir un *modelo* y es mediante este modelo donde se le especifica al agente el funcionamiento del entorno y los efectos de sus acciones. En la figura 3.2, se puede ver como se combina la percepción actual con el estado interno, con la información sobre el funcionamiento del entorno, y con las consecuencias que se producen por las acciones del agente para dar lugar a la nueva definición del estado interno, de forma que éste se actualizará. Después de actualizar el estado interno se genera la acción mediante las reglas condición-acción.

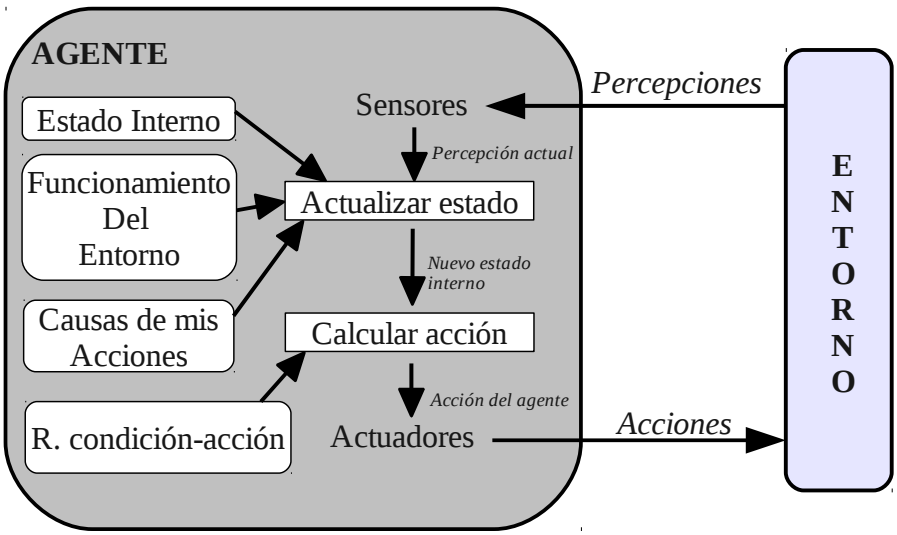

FIGURA 3.2: Definición de agente basado en modelos.

- **Agente basado en objetivos**. Este tipo de agente se diferencia del reactivo simple en la forma de calcular las acciones. Se hace uso de *objetivos* o *metas* en lugar de reglas

condición-acción. De manera, que la acción a realizar por el agente vendrá marcada por el estado actual del entorno y el objetivo o meta que se pretenda conseguir. Este tipo de agente se puede combinar con uno basado en modelos, y su funcionamiento sería el mismo que aparece en la (figura 3.2) salvo que en lugar de utilizar R. condición-acción utilizaría objetivos.

- **Agente basado en utilidad**. Este tipo de agente surge de la necesidad de solventar los problemas que tienen los agentes basados en objetivos. En algunas ocasiones, la propia naturaleza del problema a resolver conlleva a definir objetivos que pueden ser conflictivos. Según el refrán de "Todos los caminos llevan a Roma", el principal problema de un agente basado en objetivos, es que va a elegir un camino para cumplir dicho objetivo, pero no se asegura que éste sea el mejor. Mientras que un agente basado en utilidad, permite evaluar como de útil es el estado al que va a llegar si aplica dicha acción. De esta forma, el agente obrará siempre para encontrarse en el estado de más utilidad. Por lo tanto para calcular la acción, el agente hará uso de una *función de utilidad*. Y también éste se puede combinar con un agente basado en modelos, haciendo uso de la función de utilidad para calcular dicha acción.

3.3. Marco matemático

En esta sección se van a explicar los conocimientos matemáticos básicos que se han necesitado para comprender y desarrollar este *PFC*. Como la aplicación va a trabajar sobre un espacio euclídeo $\Re^3$, primero se explicarán los conceptos más básicos del espacio 3D; que son los *puntos* y *vectores*; y se concluirá con las *matrices* útiles para representar la información de la posición y rotación de los objetos 3D.

Por último destacar que, aunque los conocimientos que se van a explicar están adaptados para un espacio $\Re^3$, éstos se pueden generalizar para un espacio $\Re^n$.

3.3.1. Puntos y Vectores

El elemento más simple de un espacio euclídeo $\Re^3$, es el *punto*. Un *punto* representa una localización en el espacio $\Re^3$ y, como el espacio es de 3 dimensiones, éste está formado por tres valores ordenados (x, y, z). Tal como se aprecia en la figura 3.3, si calculamos la diferencia entre el punto b y el punto a obtendremos el vector $\vec{v}$. Por lo tanto, un *vector* [KS06] se puede definir como un segmento de línea con una dirección y sentido. Éste también viene representado por tres valores (v_x, v_y, v_z) de forma que se cumple:

$$\vec{v} = v_x\mathbf{x} + v_y\mathbf{y} + v_z\mathbf{z} \tag{3.1}$$

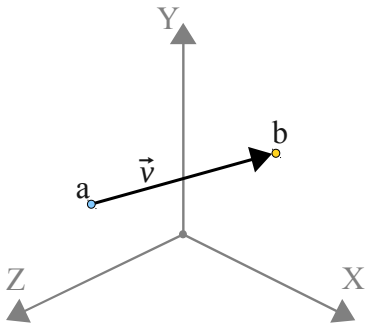

FIGURA 3.3: Definición de un vector.

La longitud o módulo del vector $\|\vec{v}\|$ se obtiene de la siguiente manera:

$$\|\vec{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2} \tag{3.2}$$

Si la longitud del vector es una unidad, se dice que el vector es *unitario* o está normalizado ($\hat{v}$). Para convertir un vector no unitario en unitario basta con hacer:

$$\hat{v} = \frac{\vec{v}}{\|\vec{v}\|} = (\frac{v_x}{\|\vec{v}\|}, \frac{v_y}{\|\vec{v}\|}, \frac{v_z}{\|\vec{v}\|}) \tag{3.3}$$

Se puede multiplicar un escalar α por un vector $\vec{v}$, y el resultado obtenido sería un vector con su misma dirección y sentido salvo que su longitud ha sido multiplicada por dicho escalar $(v_x\alpha, v_y\alpha, v_z\alpha)$. Esto se debe a que en realidad, un vector no lleva asociado una localización en el espacio, pese a que por motivos docentes, se suelen pintar en una localización determinada.

Suma de dos vectores

Dados dos vectores: $\vec{u} = (u_x, u_y, u_z)$, $\vec{v} = (v_x, v_y, v_z)$; la suma de estos vectores [KS06] genera un nuevo vector que se obtiene de la suma de las componentes de los dos vectores:

$$\vec{w} = \vec{u} + \vec{v} = (u_x + v_x,\ u_y + v_y,\ u_z + v_z) \tag{3.4}$$

En la figura 3.4 se puede ver el resultado de la suma. Tal como se puede apreciar, la suma de vectores cumple la propiedad asociativa y conmutativa.

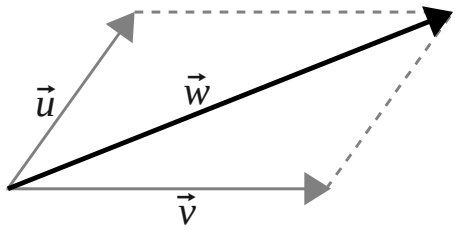

FIGURA 3.4: Suma de vectores.

Diferencia de dos vectores

Dados dos vectores: $\vec{u} = (u_x, u_y, u_z)$, $\vec{v} = (v_x, v_y, v_z)$; la diferencia de estos vectores [KS06] genera un nuevo vector que se obtiene de la resta de las componentes de los dos vectores:

$$\vec{w} = \vec{u} - \vec{v} = (u_x - v_x,\ u_y - v_y,\ u_z - v_z) \tag{3.5}$$

Obviamente la resta no cumple la propiedad conmutativa, pero como se puede ver en la figura 3.5 la resta se puede expresar como la suma del vector opuesto de $\vec{v}$.

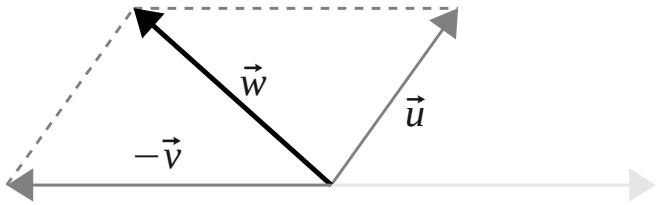

FIGURA 3.5: Diferencia de vectores, representado $\vec{w} = \vec{u} + (-\vec{v})$.

Producto escalar de dos vectores

Dados dos vectores: $\vec{u} = (u_x, u_y, u_z)$, $\vec{v} = (v_x, v_y, v_z)$; su producto escalar [KS06] es un escalar que viene dado por la siguiente expresión:

$$\vec{u} \cdot \vec{v} = u_x \cdot v_x + u_y \cdot v_y + u_z \cdot v_z = \|\vec{u}\| \cdot \|\vec{v}\| \cdot \cos(\theta) \tag{3.6}$$

Obviamente el producto escalar cumple la propiedad conmutativa. Y se puede definir como el producto de la longitud del primer vector, por la longitud de la proyección del segundo vector sobre el primero (ver figura 3.6)

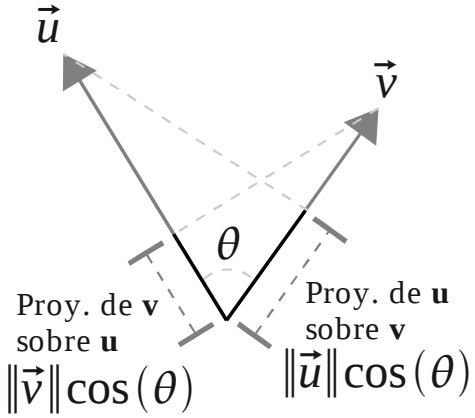

FIGURA 3.6: Representación del producto escalar de dos vectores.

Producto vectorial de dos vectores

Dados dos vectores: $\vec{u} = (u_x, u_y, u_z)$, $\vec{v} = (v_x, v_y, v_z)$; su producto vectorial $\vec{u} \times \vec{v}$ es otro vector ortogonal al plano definido por los vectores $(\vec{u}, \vec{v})$, y cuyo sentido es según el avance del giro de un sacacorchos hacia el camino más corto del vector $\vec{u}$ sobre el vector $\vec{v}$. Teniendo en cuenta esto último, se puede afirmar que el producto vectorial [KS06] no cumple la propiedad conmutativa. Se puede ver como el vector resultante de $\vec{u} \times \vec{v}$ es el mismo nada más que en sentido contrario al vector resultante de $\vec{v} \times \vec{u}$ (figura 3.7).

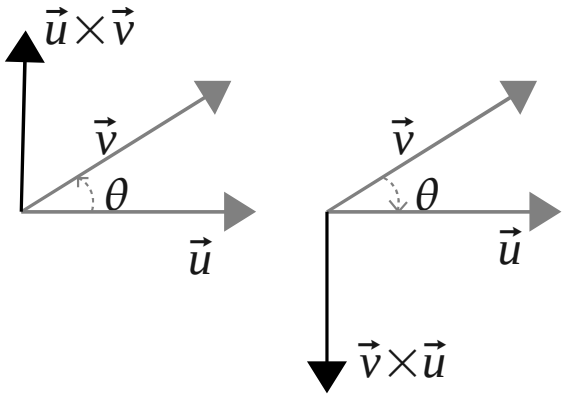

FIGURA 3.7: Representación del producto vectorial.

El producto vectorial viene dado por la siguiente expresión:

$$\vec{u} \times \vec{v} = (u_y v_z - u_z v_y,\ \ u_z v_x - u_x v_z,\ \ u_x v_y - u_y v_x) \tag{3.7}$$

Por último indicar que se cumple que la longitud del vector resultante del producto vectorial $\vec{u} \times \vec{v}$, es igual al área encerrada por el paralelogramo que forman dichos vectores $\vec{u}$ y $\vec{v}$ (ver figura 3.8).

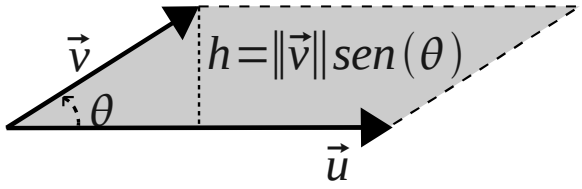

FIGURA 3.8: Propiedad del producto vectorial $\|\vec{u} \times \vec{v}\| = \|\vec{u}\|\|\vec{v}\| sen(\theta) =$ Área.

Proyección de un vector sobre un plano

Dado un plano π que está definido por un vector perpendicular a éste $\vec{n}$, para calcular la proyección de un vector $\vec{v}$ sobre el plano π se debe aplicar dos veces el producto vectorial. Para poder realizar la proyección correctamente, el vector perpendicular $\vec{n}$ debe de ser unitario ($\hat{n}$). Al hacer que el vector sea unitario, se consigue que el resultado de aplicar dos veces el producto vectorial genere un vector que será la proyección del vector $\vec{v}$ con su verdadera longitud. Si se aplica sobre $\vec{n}$ siendo no unitario, la longitud de la proyección obtenida no sería la correcta. Todo esto se debe a la propiedad explicada en la (figura 3.8).

$$\|\hat{n} \times \vec{v}\| = \|\hat{n}\|\|\vec{v}\| sen(\theta) = \|\vec{v}\| sen(\theta) \tag{3.8}$$

El resultado de calcular $\hat{n} \times \vec{v}$ generará un vector perpendicular a estos; por lo tanto proyectado en el plano y cuya longitud es la misma que la proyección de $\vec{v}$ sobre π. Como se puede ver en la figura 3.9, para que sea exactamente el vector proyección de $\vec{v}$ sobre π ($Proy_{\pi}\ \vec{v}$) se debe aplicar de nuevo el producto vectorial entre el resultado anterior y $\hat{n}$.

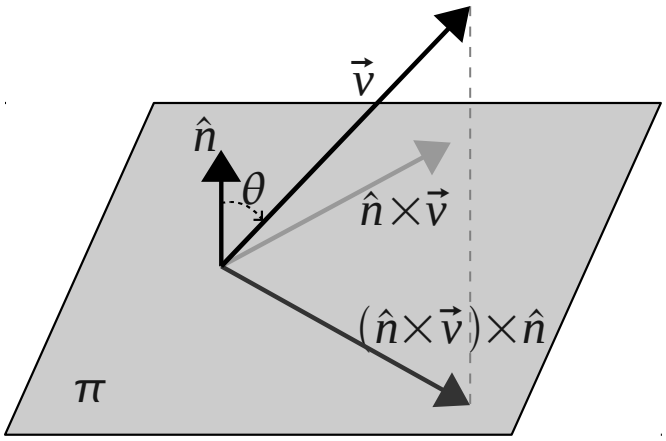

FIGURA 3.9: Proyección de $\vec{v}$ sobre π. $Proy_{\pi}\,\vec{v} = (\hat{n}\times\vec{v})\times\hat{n}$

3.3.2. Matrices

Una matriz [KS06] es una tabla de dos dimensiones que almacena un conjunto de escalares según un orden. Éste es el componente ideal para almacenar la información de posición y rotación del objeto en el espacio 3D (*matriz de transformación*). Para definir una matriz basta con especificar sus dimensiones; *m*: número de filas, *n*: número de columnas y el orden de sus elementos.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \tag{3.9}$$

Suma de matrices

Dadas dos matrices A, B de dimensiones $m \times n$, el resultado de $A + B$ es otra matriz con las mismas dimensiones $m \times n$, donde cada elemento es el resultado de la suma de los elementos de las dos matrices:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}_{m\times n} \quad B = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,n} \end{pmatrix}_{m\times n} \tag{3.10}$$

$$A+B=\begin{pmatrix} a_{1,1}+b_{1,1} & a_{1,2}+b_{1,2} & \cdots & a_{1,n}+b_{1,n} \\ a_{2,1}+b_{2,1} & a_{2,2}+b_{2,2} & \cdots & a_{2,n}+b_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1}+b_{m,1} & a_{m,2}+b_{m,2} & \cdots & a_{m,n}+b_{m,n} \end{pmatrix}_{m\times n} \tag{3.11}$$

Producto de matrices

Dadas dos matrices A, B; para efectuar su producto $A \cdot B$ se debe de cumplir que el número de columnas de A sea igual al número de filas de B. El resultado de $A \cdot B$ es otra matriz de dimensiones igual al número de filas de A $\times$ número de columnas de B. Por último indicar que el producto de matrices [KS06] no cumple la propiedad conmutativa.

$$A=\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}_{m\times n} \quad B=\begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,p} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,p} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,p} \end{pmatrix}_{n\times p} \tag{3.12}$$

$$A\cdot B=\begin{pmatrix} a_{1,1}b_{1,1}+\cdots+a_{1,n}b_{n,1} & \cdots\cdots & a_{1,1}b_{1,p}+\cdots+a_{1,n}b_{n,p} \\ a_{2,1}b_{1,1}+\cdots+a_{2,n}b_{n,1} & \cdots\cdots & a_{2,1}b_{1,p}+\cdots+a_{2,n}b_{n,p} \\ \vdots & \vdots & \vdots \\ a_{m,1}b_{1,1}+\cdots+a_{m,n}b_{n,1} & \cdots\cdots & a_{m,1}b_{1,p}+\cdots+a_{m,n}b_{n,p} \end{pmatrix}_{m\times p} \tag{3.13}$$

Determinante de una matriz

El determinante [KS06] de una matriz cuadrada A es un escalar obtenido a partir de uno de los muchos métodos que existen (*método de cofactores*):

$$\begin{aligned} \textit{Si A tiene dimensiones}: 1\times 1 \mapsto \quad & |A| = a_{1,1} \\ \textit{Si A tiene dimensiones}: n\times n;\ \textit{con } n>1 \mapsto \quad & |A| = \sum_{j=1}^{n} a_{i,j} \cdot Adjunto(A_{i,j}) \end{aligned} \tag{3.14}$$

donde i es una fila de la matriz, j las columnas de esa fila y $Adjunto(A_{i,j})$ es el adjunto del elemento $a_{i,j}$.

Un adjunto $Adjunto(A_{i,j})$ viene definido por la siguiente ecuación:

$$Adjunto(A_{i,j}) = \left|A_{i,j}\right| (-1)^{i+j} \tag{3.15}$$

donde $\left|A_{i,j}\right|$ es el determinante de la matriz A eliminando la fila y la columna señalada por i, j y manteniendo el orden de los elementos.

El resultado de aplicar éste método para una matriz 3×3, sería:

$$\begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{vmatrix} = a_{1,1} \begin{vmatrix} a_{2,2} & a_{2,3} \\ a_{3,2} & a_{3,3} \end{vmatrix} - a_{1,2} \begin{vmatrix} a_{2,1} & a_{2,3} \\ a_{3,1} & a_{3,3} \end{vmatrix} + a_{1,3} \begin{vmatrix} a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{vmatrix} \tag{3.16}$$

$$\begin{vmatrix} a_{2,2} & a_{2,3} \\ a_{3,2} & a_{3,3} \end{vmatrix} = a_{2,2} \left|a_{3,3}\right| - a_{2,3} \left|a_{3,2}\right| \tag{3.17}$$

$$\begin{vmatrix} a_{2,1} & a_{2,3} \\ a_{3,1} & a_{3,3} \end{vmatrix} = a_{2,1} \left|a_{3,3}\right| - a_{2,3} \left|a_{3,1}\right| \tag{3.18}$$

$$\begin{vmatrix} a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{vmatrix} = a_{2,1} \left|a_{3,2}\right| - a_{2,2} \left|a_{3,1}\right| \tag{3.19}$$

Transpuesta de una matriz

Dada la matriz A de dimensiones $m \times n$, su transpuesta A^t [KS06] es una matriz de dimensiones $n \times m$ donde se cumple que el elemento $a_{i,j}$ pasa a ser $a_{j,i}$.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}_{m \times n} \quad A^t = \begin{pmatrix} a_{1,1} & a_{2,1} & \cdots & a_{m,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{m,2} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1,n} & a_{2,n} & \cdots & a_{m,n} \end{pmatrix}_{n \times m} \tag{3.20}$$

Inversa de una matriz

Dada una matriz cuadrada A, se dice que A tiene *inversa* o es *invertible* [KS06] si existe una matriz B tal que:

$$AB = I \tag{3.21}$$

Siendo I la matriz *identidad*.[1] Existen muchas formas de resolver la ecuación 3.21, una de las más utilizadas es mediante la siguiente expresión:

$$A^{-1} = \frac{1}{|A|} Adjunta(A^t) \tag{3.22}$$

donde $Adjunta(A^t)$ sería la matriz adjunta de A^t. Los componentes de la matriz $Adjunta(A^t)$ se calculan aplicando la ecuación 3.15 para cada elemento de la matriz A^t.

3.4. *Raytracer* (trazador de rayos)

Raytracer [Suf07] es una técnica de gráficos por ordenador, que permite plasmar en una fotografía la visión captada por una cámara virtual sobre un mundo 3D. Si se atiende a su definición, éste toma un papel muy importante a la hora de desarrollar *TraceMon*. El por qué es tan importante, se debe a que el *raytracer*, se encarga de establecer la correspondencia de una coordenada 2-D de la propia fotografía, con una coordenada 3-D del mundo virtual.

Los componentes que emplea un *raytracer* son los siguientes:

- **El rayo** que se va a lanzar para determinar el objeto 3D y la coordenada 3D con la que interseca.
- **El mundo 3D** que contiene todos los objetos 3D que existen en la escena a fotografiar.
- **La cámara virtual** con la que se captará el mundo 3D.

[1] Matriz cuadrada, en cuya diagonal son todo 1, y el resto 0.

3.4.1. El rayo

El proceso de *raytracing* consiste en la emisión de rayos desde el punto de vista que proporciona la cámara. Para representar el concepto de *rayo* [Suf07], simplemente se necesita conocer el punto de *origen* desde donde es emitido el rayo y *la dirección* que va a tener el rayo; representada esta última por un vector unitario. Luego se puede calcular cualquier punto que pertenezca al rayo por la siguiente expresión:

$$Punto_{rayo} = origen + t \cdot dirección \tag{3.23}$$

donde t es un número real que multiplicándolo al vector unitario de la dirección se consigue que el rayo aumente la longitud de forma proporcional a este número. En la figura 3.10 se puede apreciar como el único punto del rayo es el origen cuando $t = 0$.

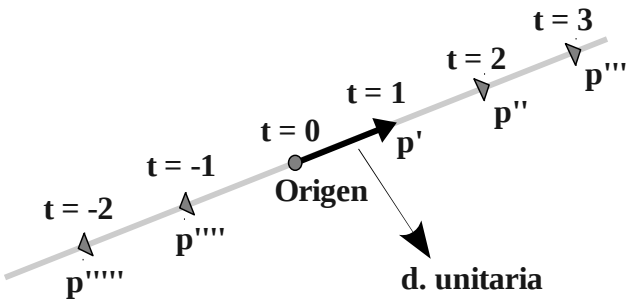

FIGURA 3.10: Representación de un rayo en el proceso de raytracing.

3.4.2. El mundo 3D

El *mundo 3D* representa el conjunto de objetos que forman la escena. La manera de incidir el rayo sobre los objetos dependerá de la forma que tengan éstos. Estudiar la intersección del rayo para cada una de las formas que presentan los objetos se convierte en una tarea imposible. Con el objetivo de simplificar esta tarea, la forma de un objeto se puede aproximar mediante el uso de triángulos. Por lo tanto, el mundo 3D puede ser representado por planos y triángulos.

Representación de un plano en un espacio tridimensional

La representación de un *plano* [Suf07] es similar a una hoja plana infinita. Se considera una superficie abierta, y para definirlo basta con dar un punto de éste y la normal[2] por dicho punto. En la figura 3.11 se aprecia como la normal determina la orientación del plano.

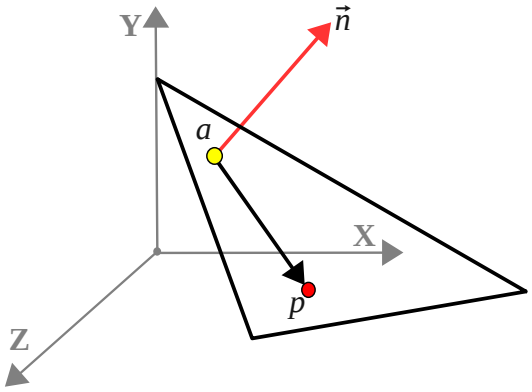

FIGURA 3.11: Representación de un plano en un espacio tridimensional.

Todo punto p que pertenezca al plano, debe cumplir que el producto escalar entre el vector $\vec{p}$ (vector formado por el punto p y el punto que a que define al plano), y el vector normal $\vec{n}$ sea 0:

$$(p-a)\cdot\vec{n} = \vec{p}\cdot\vec{n} = 0 \tag{3.24}$$

El punto de intersección entre el rayo y el plano pertenece a ambos. Es por eso que si se sustituye la ecuación 3.23 por el punto p se obtiene:

$$(origen + t\cdot dirección - a)\cdot\vec{n} = 0 \tag{3.25}$$

De la ecuación 3.25 se conoce todo excepto t que es la distancia desde el origen del rayo hasta la intersección con el plano. Luego si se despeja t, se obtiene:

$$t = \frac{(a-origen)\cdot\vec{n}}{dirección\cdot\vec{n}} \tag{3.26}$$

El valor de t puede ser $t > 0$ o $t < 0$. Como muestra la figura 3.12, si $t < 0$ implica que el objeto se encuentra detrás de la cámara que está captando la escena.

[2] Vector perpendicular a la superficie del plano.

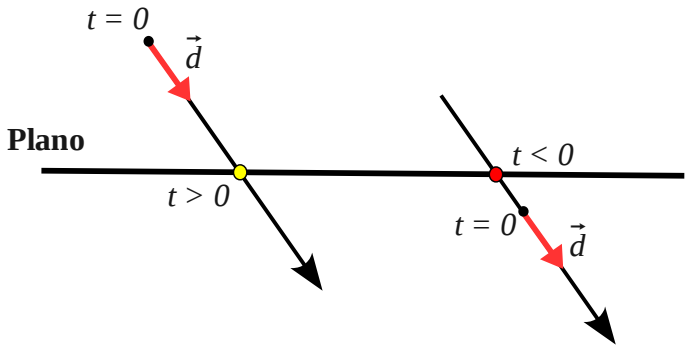

FIGURA 3.12: Intersección del rayo sobre el plano.

Representación de un triángulo en un espacio tridimensional

La representación de un *triángulo* (figura 3.13) está definida por tres puntos que se corresponden con cada uno de sus vértices. A partir de los vértices (a,b,c) se puede calcular el vector normal $\hat{n}$ del triángulo mediante:

$$\hat{n} = \frac{(b-a)\times(c-a)}{\|(b-a)\times(c-a)\|} \tag{3.27}$$

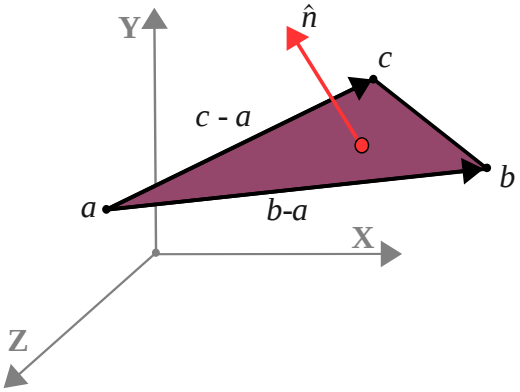

FIGURA 3.13: Representación de un triángulo en un espacio 3D.

La forma de calcular el punto de intersección de un rayo con un triángulo [Suf07], consiste en convertir el triángulo en un plano infinito. De esta forma, se aplica la intersección con dicho plano y luego se analiza si el rayo cayó en la región del plano donde se encuentra definido el triángulo. Para llevar a cabo esta transformación se hace uso de las coordenadas baricéntricas (α,β,γ). De tal forma que dado un triángulo con los vértices (a,b,c) todo punto p del plano cumple:

$$p(\alpha,\beta,\gamma) = \alpha\cdot a + \beta\cdot b + \gamma\cdot c \tag{3.28}$$

Para que el punto p pertenezca al triángulo se tiene que cumplir:

$$\alpha+\beta+\gamma=1 \tag{3.29}$$

Si despejamos la ecuación 3.29 en función de α, y sustituimos en la ecuación 3.28 se obtiene:

$$p(\alpha,\beta,\gamma)=a+\beta\cdot(b-a)+\gamma\cdot(c-a) \tag{3.30}$$

En la figura 3.14 se puede ver la representación del plano que engloba al triángulo que cumple la ecuación 3.30.

En ésta se aprecia que cuando $\beta=0$ da lugar al γ *eje* que tiene por ecuación:

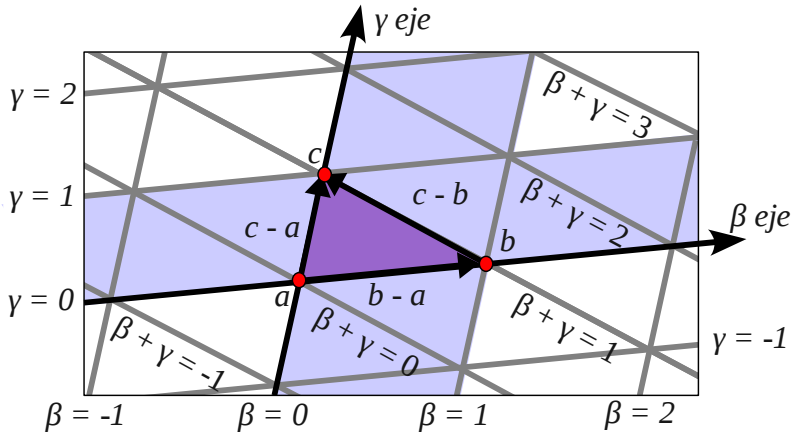

FIGURA 3.14: Transformación de un triángulo en un plano infinito.

$$p(\alpha,\beta,\gamma)=a+\gamma\cdot(c-a) \tag{3.31}$$

En la ecuación 3.31, si el γ pertenece al intervalo $[0,1]$, se está definiendo la arista **c-a** del triángulo.

Cuando $\gamma=0$ da lugar al β *eje* que tiene por ecuación:

$$p(\alpha,\beta,\gamma)=a+\beta\cdot(b-a) \tag{3.32}$$

En la ecuación 3.32, si el β pertenece al intervalo $[0,1]$, se está definiendo la arista **b-a** del triángulo.

Cuando $\beta + \gamma = 1$, si se sustituye por β en 3.30 se obtiene:

$$p(\alpha, \beta, \gamma) = b + \gamma \cdot (c - b) \tag{3.33}$$

En la ecuación 3.33, si el γ pertenece al intervalo $[0, 1]$, se está definiendo la arista **c-b** del triángulo.

Una vez descrita la representación del plano que incluye al triángulo, se procede a calcular la intersección del rayo con éste. Si se sustituye la ecuación 3.23, por $p(\alpha, \beta, \gamma)$ en la ecuación 3.30 se obtiene:

$$o + t \cdot d = a + \beta \cdot (b - a) + \gamma \cdot (c - a) \tag{3.34}$$

Si se despliega la ecuación 3.34 se obtienen tres ecuaciones, con tres incógnitas.

$$\begin{aligned} \beta \cdot (a_x - b_x) + \gamma(a_x - c_x) + t \cdot d_x &= a_x - o_x \\ \beta \cdot (a_y - b_y) + \gamma(a_y - c_y) + t \cdot d_y &= a_y - o_y \\ \beta \cdot (a_z - b_z) + \gamma(a_z - c_z) + t \cdot d_z &= a_z - o_z \end{aligned} \tag{3.35}$$

Las cuales se pueden resolver haciendo uso de las reglas de *Cramer*. De forma que la ecuación 3.35 se puede representar:

$$\begin{pmatrix} \mathbf{a} & \mathbf{b} & \mathbf{c} \\ \mathbf{e} & \mathbf{f} & \mathbf{g} \\ \mathbf{i} & \mathbf{j} & \mathbf{k} \end{pmatrix} \times \begin{pmatrix} \beta \\ \gamma \\ \mathbf{t} \end{pmatrix} = \begin{pmatrix} \mathbf{d} \\ \mathbf{h} \\ \mathbf{l} \end{pmatrix} \tag{3.36}$$

$$\begin{aligned} \mathbf{a} &= a_x - b_x, & \mathbf{b} &= a_x - c_x, & \mathbf{c} &= d_x, & \mathbf{d} &= a_x - o_x \\ \mathbf{e} &= a_y - b_y, & \mathbf{f} &= a_y - c_y, & \mathbf{g} &= d_y, & \mathbf{h} &= a_y - o_y \\ \mathbf{i} &= a_z - b_z, & \mathbf{j} &= a_z - c_z, & \mathbf{k} &= d_z, & \mathbf{l} &= a_z - o_z \end{aligned} \tag{3.37}$$

Y de su resolución se obtiene:

$$\beta = \frac{\mathbf{d}(\mathbf{fk}-\mathbf{gj})+\mathbf{b}(\mathbf{gl}-\mathbf{hk})+\mathbf{c}(\mathbf{hj}-\mathbf{fl})}{\mathbf{a}(\mathbf{fk}-\mathbf{gj})+\mathbf{b}(\mathbf{gi}-\mathbf{ek})+\mathbf{c}(\mathbf{ej}-\mathbf{fi})}$$

$$\gamma = \frac{\mathbf{a}(\mathbf{hk}-\mathbf{gl})+\mathbf{d}(\mathbf{gi}-\mathbf{ek})+\mathbf{c}(\mathbf{el}-\mathbf{hi})}{\mathbf{a}(\mathbf{fk}-\mathbf{gj})+\mathbf{b}(\mathbf{gi}-\mathbf{ek})+\mathbf{c}(\mathbf{ej}-\mathbf{fi})} \tag{3.38}$$

$$\mathbf{t} = \frac{\mathbf{a}(\mathbf{fl}-\mathbf{hj})+\mathbf{b}(\mathbf{hi}-\mathbf{el})+\mathbf{d}(\mathbf{ej}-\mathbf{fi})}{\mathbf{a}(\mathbf{fk}-\mathbf{gj})+\mathbf{b}(\mathbf{gi}-\mathbf{ek})+\mathbf{c}(\mathbf{ej}-\mathbf{fi})}$$

3.4.3. Representación de una cámara virtual en un espacio tridimensional

El último componente que interviene en el *raytracer* y no menos importante es la cámara virtual. A la hora de colocar una cámara virtual en un mundo 3D, se conoce de ella, el punto central de la cámara (*eye*), el punto hacia donde está mirando (*lookat*), y el vector $\vec{up}$ que indica la orientación de la cámara con respecto al eje $Y(0,1,0)$. Con esta información, toda cámara virtual está compuesta de:

- Base ortonormal.
- Plano vista.

Base ortonormal

Con la información del *eye*, *lookat*, y $\vec{up}$ se tiene que construir una *base ortonormal* [Suf07]. Una *base ortonormal*, esta formada por tres vectores unitarios $\hat{v},\hat{w},\hat{u}$ y perpendiculares entre sí (figura 3.15).

En la figura 3.15 se puede apreciar como $\hat{w}$ es el vector $\vec{e}$ $(eye - lookat)$ normalizado:

$$\hat{w} = \frac{\vec{e}}{\|\vec{e}\|} \tag{3.39}$$

Con $\hat{w}$ calculado, si se hace el producto vectorial de $\vec{up}$ con $\hat{w}$ se obtiene el vector $\hat{u}$:

$$\hat{u} = \frac{\vec{up} \times \hat{w}}{\|\vec{up} \times \hat{w}\|} \tag{3.40}$$

Y por último, para calcular $\hat{v}$ se hace el producto vectorial entre $\hat{w}$ y $\hat{u}$:

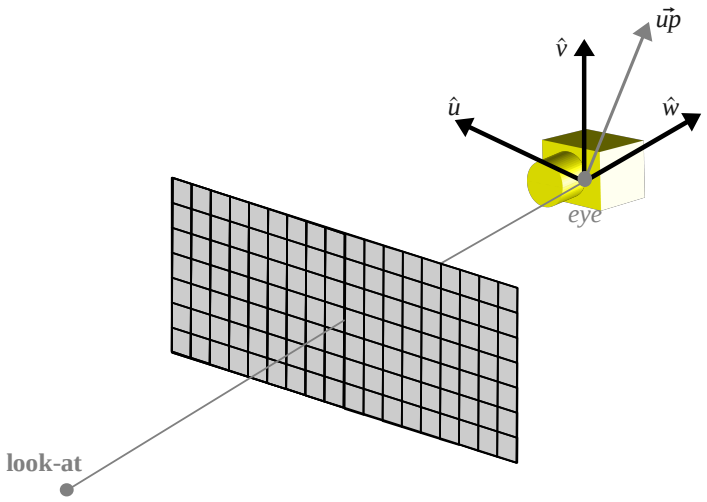

FIGURA 3.15: Construcción de la base ortonormal de una cámara.

$$\hat{v} = \hat{w} \times \hat{u} \tag{3.41}$$

Plano vista

El *plano vista* [Suf07], es el plano donde se va a renderizar la escena, en otras palabras la fotografía. Éste viene definido por el número de píxeles que componen la imagen $(ancho \times alto)$. Dado el número de píxeles a lo ancho y a lo alto, se puede calcular el tamaño del pixel $(p_{ancho} \times p_{alto})$.

Para calcular el tamaño del pixel, primero se debe obtener la relación que existe entre el ancho y el alto de la imagen. Para ello se obtiene como referencia la mayor magnitud del $(ancho \times alto)$ de la imagen. Un ejemplo:

$$Para\ una\ imagen\ (400 \times 300),\ \ ref = 400 \mapsto\ \ (\frac{400}{ref} \times \frac{300}{ref}) =\ imagen\ (1 \times 0.75)$$

Una vez calculadas las dimensiones de una imagen $(ancho_{imagen} \times alto_{imagen})$, el tamaño del pixel es tan simple como dividir cada una de las dimensiones por el número de píxeles que tiene en esa dimensión:

$$Pixel_{ancho} = \frac{ancho_{imagen}}{número_{pixeles\_ancho}} \qquad Pixel_{alto} = \frac{alto_{imagen}}{número_{pixeles\_alto}} \tag{3.42}$$

El plano vista juega un papel muy importante en el *raytracer* porque trata de lanzar rayos que sigan una trayectoria desde el centro de la cámara y el punto medio de cada uno de los píxeles de la imagen. Para calcular la trayectoria que sigue el rayo, existen multitud de formas. Como en la sección anterior se construyó una base ortonormal en el centro de la cámara, se va a hacer uso de ésta para calcular dicha trayectoria. Por lo general, el recorrido de los píxeles de una imagen, suele tener su origen en la esquina inferior izquierda ver figura 3.16.

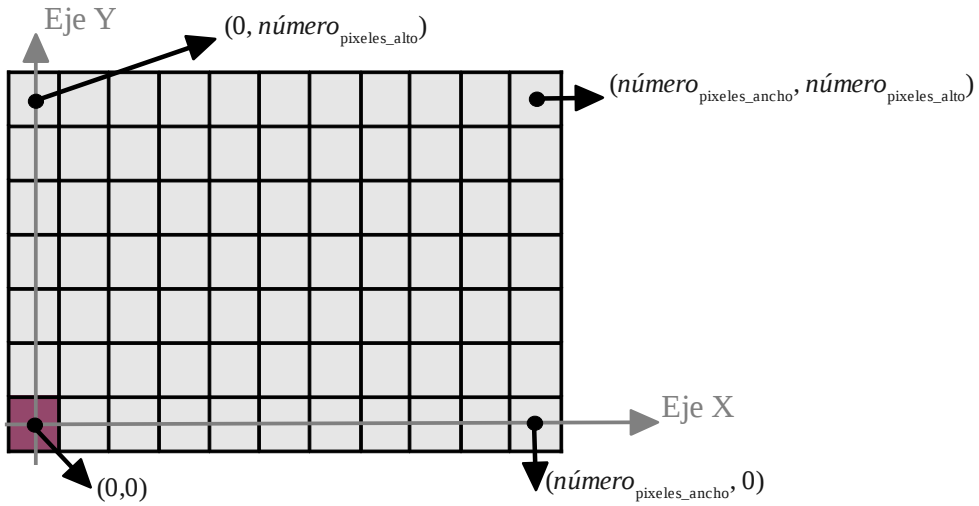

FIGURA 3.16: Recorrido general de los píxeles de una imagen.

Para calcular la trayectoria utilizando la base ortonormal, se debe trasladar este recorrido al que marca la figura 3.17. Esto se debe a que si se mira de frente la figura 3.15 se puede ver que el centro de la cámara se encuentra en el pixel central.

La ecuación que rige este cambio en el sistema de referencia es:

$$Dado\ un\ pixel\ = (x,y) \mapsto \quad Nuevo_{pixel} = (x - \frac{número_{pixeles\_ancho}}{2}, y - \frac{número_{pixeles\_alto}}{2}) \tag{3.43}$$

Teniendo en cuenta las expresiones 3.42 y 3.43, el rayo pasa por el punto medio de cada pixel ($+0.5$) y la *distancia focal* determina la longitud entre el centro de la cámara y el plano vista. El punto por el que pasa cada rayo, tiene las coordenadas:

$$P_{rayo} = (Pixel_{ancho} \cdot (Nuevo_{pixel\_x} + 0.5),\ \ Pixel_{alto} \cdot (Nuevo_{pixel\_y} + 0.5),\ \ -distancia_{focal}) \tag{3.44}$$

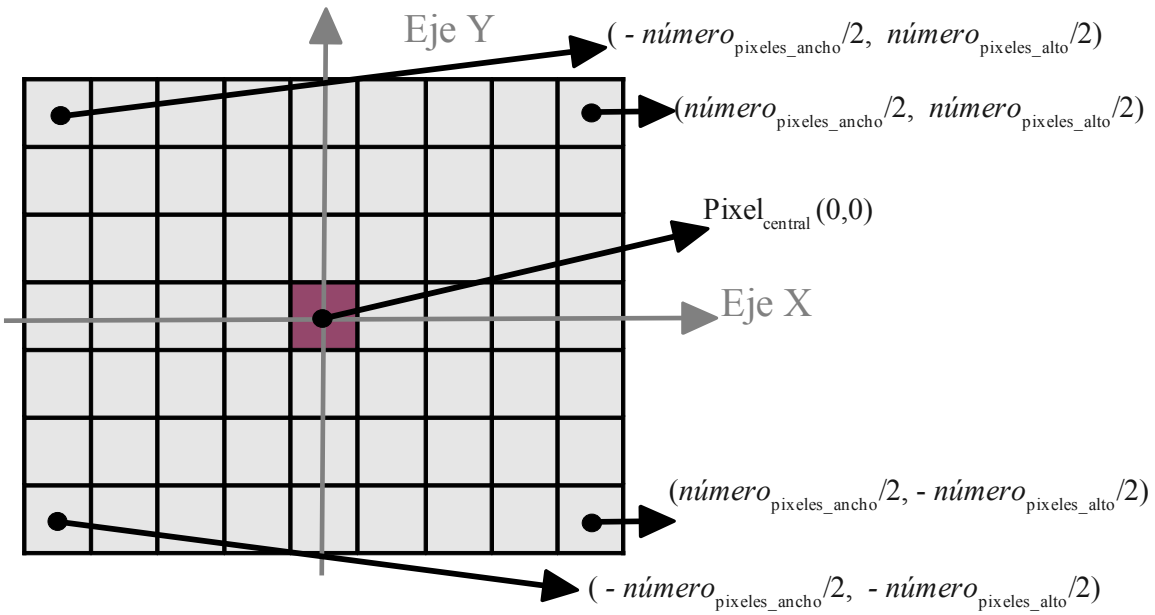

FIGURA 3.17: Cambio en el sistema de referencia de una imagen.

Se puede apreciar en la figura 3.15, que el $\hat{w}$ tiene sentido opuesto hacia donde está el plano vista, de ahí que sea $-distancia_{focal}$ la componente Z del P_{rayo}.

Con la ecuación 3.44 y la base ortonormal, se procede a calcular la trayectoria del rayo como una combinación lineal de los vectores de la base ortonormal.

$$Trayectoria_{rayo} = Punto\_rayo_x \cdot \hat{u} + Punto\_rayo_y \cdot \hat{v} + Punto\_rayo_z \cdot \hat{w} \qquad (3.45)$$

Según la ecuación 3.45 se obtiene un vector que define la trayectoria del rayo, pero para que pueda ser utilizado en el *raytracer*, éste debe de ser unitario como se especificó en la sección 3.4.1.

3.5. Técnicas de visión por computador

En esta sección se explicará el proceso de calibración de una cámara y se concluirá con un estudio sobre algunas de las técnicas utilizadas para la clasificación y seguimiento de los objetos.

3.5.1. Calibración de una cámara

El proceso de calibración de una cámara es de utilidad para conocer la posición y orientación de la cámara junto con su distancia focal y propiedades de la lente.

OpenCV (ver sección 3.6.1) proporciona una serie de funciones con las que se puede calibrar una cámara. Para calibrar una cámara se utiliza un patrón y tras una serie de tomas de ese patrón se obtiene la información sobre la calibración de la cámara. Esta información está formada por los *coeficientes de distorsión* de la lente, los *parámetros intrínsecos* de la cámara (matriz intrínseca) y los *parámetros extrínsecos* de la cámara (orientación-posición).

Coeficientes de distorsión de una lente

La lente de una cámara real no es perfecta y presenta pequeños fallos. Estos fallos son resultado de un ahorro de costes de los fabricantes de lentes. En lugar de fabricar lentes con una parábola ideal, se fabrican lentes esféricas que son más fáciles y más baratas.

En el proceso de fabricación de una cámara se pueden dar dos tipos de distorsiones:

- **La distorsión radial**. Este efecto de distorsión [Gar] se manifiesta sobre los bordes de la imagen (cuanto mayor sea la distancia sobre el centro de la imagen, más propenso será a que el rayo se desvíe). A este efecto se le denomina *efecto de ojo de pez* y viene caracterizado por una serie de Taylor donde solamente se necesitan los tres primeros parámetros $k1, k2$ y $k3$ para tener una perfecta aproximación de la distorsión ver figura 3.18.

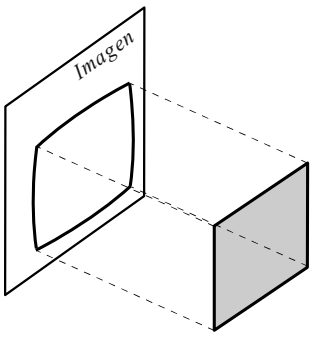

FIGURA 3.18: Ejemplo de distorsión radial.

Para corregir la distorsión radial se hace uso de las siguientes ecuaciones:

$$\begin{aligned} x_{corregido} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_{corregido} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \end{aligned} \tag{3.46}$$

donde r es la proximidad del pixel a corregir respecto al centro de la imagen.

- **La distorsión tangencial**. Este efecto [Gar] se debe a que la lente no se encuentra perfectamente paralela al plano vista de imagen ver figura 3.19. Está definido por dos parámetros $p1$ y $p2$ donde la corrección es llevada a cabo por las siguientes ecuaciones.

$$\begin{aligned} x_{corregido} &= x + [2p_1y + p_2(r^2 + 2x^2)] \\ y_{corregido} &= y + [p_1(r^2 + 2y^2) + 2p_2x] \end{aligned} \tag{3.47}$$

donde r es la proximidad del pixel a corregir respecto al centro de la imagen.

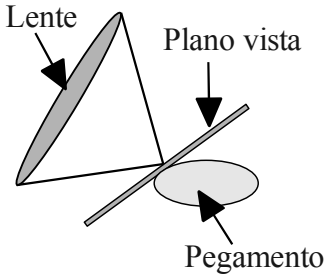

FIGURA 3.19: Causa de la distorsión tangencial.

Parámetros intrínsecos de una cámara

Los parámetros intrínsecos de la cámara se representan mediante una matriz de dimensiones 3×3. Esta matriz contiene información sobre la distancia focal de la cámara y el punto principal de la cámara (el centro de la imagen). La matriz de parámetros intrínsecos obtenida por *OpenCV* [BK08] tiene el siguiente aspecto:

$$\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{3.48}$$

donde el punto formado por las coordenadas (c_x, c_y) representa el centro de la imagen de esa cámara y f_x, f_y corresponden a la distancia focal expresada en unidades de píxeles. De forma que la distancia focal f se puede obtener:

$$f = \frac{f_x}{ancho\_de\_la\_imagen} \tag{3.49}$$

Orientación y posición de una cámara

OpenCV (sección 3.6.1) ofrece funciones que permiten obtener una estimación de la posición de la cámara. La posición de la cámara contiene información sobre el centro 3D de la cámara y sus rotaciones sobre el sistema de referencia del mundo. La información de rotación se representa mediante una matriz de dimensiones 3×3. En OpenCV [BK08] esta matriz se construye de la siguiente manera: $R = R_z(\theta) \cdot R_y(\varphi) \cdot R_x(\psi)$.

$$R_z(\theta) = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$R_y(\varphi) = \begin{pmatrix} \cos(\varphi) & 0 & -\sin(\varphi) \\ 0 & 1 & 0 \\ \sin(\varphi) & 0 & \cos(\varphi) \end{pmatrix} \tag{3.50}$$

$$R_x(\psi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\psi) & \sin(\psi) \\ 0 & -\sin(\psi) & \cos(\psi) \end{pmatrix}$$

Se puede representar de una forma conjunta el centro de la cámara y la matriz de rotación. Dicha forma da lugar a una matriz cuadrada de orden 4, denominada *matriz de transformación* (T).

$$T = \begin{pmatrix} R_{1,1} & R_{1,2} & R_{1,3} & 0 \\ R_{2,1} & R_{2,2} & R_{2,3} & 0 \\ R_{3,1} & R_{3,2} & R_{3,3} & 0 \\ C_x & C_y & C_z & 1 \end{pmatrix}_{4\times4} \tag{3.51}$$

Un detalle muy a tener en cuenta es que los datos obtenidos por OpenCV están expresados sobre un sistema de referencia colocado en la propia cámara. Si el sistema de referencia no se encuentra en la cámara, basta con aplicar la inversa a la matriz de transformación T^{-1} y se obtendrán los datos de acuerdo al sistema de referencia del entorno.

3.5.2. Métodos de segmentación

La segmentación consiste en detectar aquellos movimientos que aparecen en el entorno que se está monitorizando. Dependiendo de los componentes que formen el sistema de vigilancia, la información del movimiento puede ser más precisa y rica. Según el objetivo de este proyecto, el estudio de los métodos de segmentación se va a centar sobre cámaras de vídeo. Actualmente los métodos de segmentación se basan en la variación de los píxeles con el tiempo. Existen tres corrientes bien diferenciadas de métodos de segmentación:

Diferencia Temporal (*Temporal Difference*)

En [VV05] se proponen uno de los métodos más sencillos para detectar movimiento de objetos. Consiste en observar las diferencias en el umbral de los píxeles de dos *frames*[3] consecutivos. Éste tiene buen rendimiento en *entornos dinámicos*[4] porque es bastante adaptativo. Pero es más pobre a la hora de definir el contorno del objeto.

En [TH] se hace uso de un peso acumulativo W_{acum} con el fin de detectar los movimientos lentos en dos imágenes consecutivas $I(x,y,t)$, $I(x,y,t+1)$. El resultado de diferenciar las dos imágenes, es una imagen $I_{diferencias}$ de fondo negro resaltando los movimientos en color blanco. Para detectar esas variaciones en los píxeles se hace uso de un umbral T_d, de tal forma que se cumple:

$$I_{diferencias}(x,y,t+1) = \begin{cases} Blanco, & Si\ \ I_{acum}(x,y,t+1) > T_d \\ Negro, & Resto\ de\ casos \end{cases} \tag{3.52}$$

donde $I_{acum}(x,y,t)$ se inicializa a una imagen en negro, e $I_{acum}(x,y,t+1)$ es:

$$I_{acum}(x,y,t+1) = (1-W_{acum})I_{acum}(x,y,t) + W_{acum}|I(x,y,t+1) - I(x,y,t)| \tag{3.53}$$

Si se aplica este método a dos imágenes *imagen* 1 e *imagen* 2 el resultado que se obtendría puede verse en la figura 3.20.

[3]Un video tiene veinticinco imágenes por segundo. A estas imágenes se les denomina frames.
[4]Entornos que cambian periódicamente debido a nuevos sucesos en ellos.

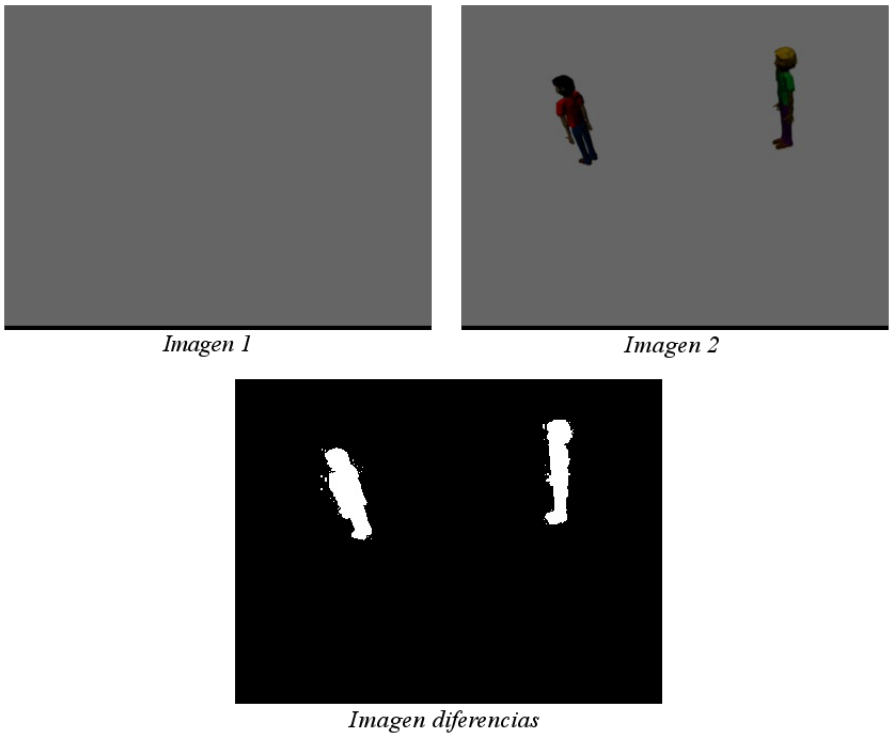

FIGURA 3.20: Aplicación del algoritmo "Diferencia Temporal".

Flujo óptico (*Optical Flow*)

Estos tipos de métodos son utilizados para detectar movimiento de objetos cuando las cámaras del sistema de vigilancia se están moviendo. Dado que el proyecto se va a desarrollar sobre sistemas de vigilancia con cámaras fijas, no se ha realizado un estudio profundo de éstos. En [JBB94] se describen algunos de estos métodos *Horn and Schunck* y *Lucas and Kanade*. Debido a que las cámaras se están moviendo si se aplicase un método de *diferencia temporal* un objeto estático se interpretaría en movimiento cuando es la propia cámara la que se mueve. La forma de evitar ese error consiste en calcular la velocidad que lleva la cámara a partir de un ligero desplazamiento en 2D. Si la vista de la cámara fuese aproximadamente a una vista de planta, su velocidad en 2D se asemejaría a la velocidad en el mundo real. Es por eso que estos algoritmos hacen suposiciones sobre una velocidad constante a cuya menor magnitud se les denomina *velocidad normal*.

Las características de estos algoritmos hacen que sean muy costosos computacionalmente e imposibles de utilizar en sistemas de tiempo real, un motivo más para no utilizar este tipo de métodos en este proyecto.

Sustracción del fondo (*Background subtraction*)

Estos tipos de métodos son muy utilizados en sistemas de vigilancia debido a que el entorno monitorizado se mantiene igual. Consiste en crear un modelo de lo que se considera como fondo del entorno, y una vez creado ese modelo se puede detectar cualquier movimiento sobre dicho fondo.

Según [BK08] el concepto de fondo no solo incluye aquello que permanece estático, sino también a los objetos que tienen movimiento constante como es el caso del movimiento de los árboles. Existe una gran variedad de éstos, siendo unos más rápidos que otros y comportándose mejor ante cambios bruscos de iluminación. Estos métodos están aprendiendo lo que es fondo y lo que no es, de forma que si un objeto que no se considera como fondo permanece inmóvil durante mucho tiempo pasará a formar parte del fondo. Cuando hay cambios bruscos de iluminación como es el caso de una habitación oscura y se enciende la luz, la mayoría de los métodos detectaría como movimiento toda la habitación. Por eso en estas situaciones se suelen compaginar con otros modelos para tratar estas falsas alarmas.

Un ejemplo de método de background es el *codebook*[BK08]. Éste tiene una fase de aprendizaje, en la que se está aprendiendo lo que es el fondo. Una vez que el fondo se ha aprendido, se coge el frame actual y se compara con el modelo de fondo. El resultado de esta comparación es una imagen binaria donde los movimientos son representados en blanco. Para mejorar esta imagen y eliminar el ruido que pueda tener, se aplican algoritmos de erosión y de agrupación de los movimientos que sean muy próximos. La mayoría de métodos background, una vez que aprenden el fondo, éstos no lo actualizan debido al consumo de memoria. Pero *codebook* permite actualizarse periódicamente y hacer limpieza de la información más antigua con el fin de liberar memoria.

3.5.3. Métodos de clasificación

Una vez que se ha detectado el movimiento, se procede a clasificarlo en un tipo de objeto. Los métodos de clasificación deben de tener un alto nivel de confianza, porque un error en una clasificación produciría errores catastróficos en las fases de análisis de comportamientos. Por ejemplo, si un vehículo es clasificado como una persona, su comportamiento será tratado como una persona en lugar de vehículo y provocará una sucesión de falsas alarmas en el entorno que se está monitorizando.

Existe una gran variedad de métodos de clasificación, pero se van explicar tres tipos genéricos. Estos métodos no son puramente de la rama de visión por computador y se mezclan con ramas de aprendizaje supervisado por conocimiento del entorno.

Clasificación basada en modelos

En [VV05] se clasifican los movimientos atendiendo a la afinidad de un modelo. De manera que estas técnicas consisten en construir un modelo 2-D o 3-D de cada uno de los tipos de objetos a clasificar. Toda la responsabilidad de estas técnicas recae en la construcción del modelo. Cuanto más rico y preciso sea la definición del modelo, mayor será la precisión del clasificador. La información que se utiliza a la hora de construir el modelo suele ser dimensiones, velocidad, zonas en las que se puede mover ese tipo de objeto, etc. La construcción de un modelo para clasificar entre personas y vehículos podría basarse en:

- **Relación de aspecto.** [MT] Consiste en obtener la relación que tienen las dos dimensiones del objeto detectado (ancho y alto). Si se analiza el ancho y el alto de los objetos a clasificar, las personas tienen más alto que ancho mientras que los vehículos tienen más ancho que alto. Luego la relación $\frac{alto}{ancho}$ será mayor para objetos personas que para objetos vehículos.

- **Complejidad de la forma del objeto.** [MT] Si se atiende a la forma de las personas y vehículos, se puede observar que la forma de las personas es más compleja que la de los vehículos. Para medir dicha complejidad se analiza la dispersión $\frac{perimetro^2}{area}$. Cuanto mayor dispersión, mayor complejidad tiene la forma.

- **Variación en el flujo del movimiento.** [Bro] Para clasificar entre personas y vehículos se puede construir también un modelo basado en la variación del movimiento. El movimiento de un vehículo es rígido mientras que en las personas el movimiento se atribuye a algunas partes del cuerpo (brazos y piernas). Para detectar personas basta con crear un modelo dividido en tres partes donde los movimientos se encuentren en la segunda y tercera parte del modelo.

- **Velocidad del objeto.** [Bro] Si se está trabajando en entornos donde la diferencia de velocidad entre las personas y vehículos es considerable, se puede construir un modelo donde se tenga en cuenta la velocidad con la que se mueve ese objeto y poder asociarlo al modelo que menos difiera en su velocidad.

- **Contorno del objeto.** [CG09] Es una técnica similar a la de la complejidad de la forma del objeto. Consiste en calcular el momento y el centro de masa asociado al contorno del objeto. Dada una imagen binaria formada por (x,y) píxeles, el momento $m_{p,q}$ de un contorno formado por n pixeles es:

$$m_{p,q} = \sum_{i=1}^{n} Imagen(x,y)x^p y^q$$

  Y las coordenadas que definen el centro de masa o centroide del contorno se obtienen mediante $\left(\frac{m_{1,0}}{m_{0,0}}, \frac{m_{0,1}}{m_{0,0}}\right)$. Atendiendo a los valores del centro de masa y a la variación del contorno con respecto a los ejes X e Y se pueden definir heurísticas de clasificación de los objetos.

Construidos estos modelos, la forma de aplicar este algoritmo es tan simple como dado un movimiento aplicarle los distintos modelos para ver cuál modelo es más afin a dicho movimiento.

Clasificación basada en histogramas

Los clasificadores más comunes se basan en histogramas. Si se construyen bien, éstos suelen tener un acierto del 97 % y un fallo del 3 %. Los clasificadores basados en histogramas consisten en obtener un histograma modelo para cada tipo de objeto. Para construir dicho histograma modelo tienen que aprender de un conjunto de imágenes las características del objeto. Una vez que se tiene construido el modelo, dada una imagen de movimiento se obtiene su histograma y se compara con cada uno de los histogramas modelo. El objeto será clasificado con el más afín a dicho modelo.

OpenCV [BK08] da soporte para crear clasificadores basados en histogramas (*haar classifier cascade*). El proceso de creación de un clasificador consta de tres fases:

1. **Repositorio de imágenes**. Consiste en obtener un repertorio amplio de imágenes con las que se va a entrenar el clasificador. Cuanto mayor sea el número y la variación de las imágenes mejor será la precisión del clasificador. Se tienen que encontrar imágenes que contengan al objeto a clasificar *(imágenes positivas)* e imágenes que no lo contengan *(imágenes negativas)*.

2. **Creación de la muestra**. Una vez que se tiene el repositorio de las imágenes, se tiene que recortar el objeto a clasificar de las imágenes positivas. Una vez que se tienen todos los recortes, se unen en una imagen, que constituirá la muestra para el entrenamiento del clasificador.

3. **Entrenamiento**. En esta fase, se le indican algunos parámetros que definan las propiedades del clasificador (robusto, uso de memoria en el entrenamiento, número de etapas, debilidad del clasificador, etc). Una vez definidas sus propiedades se procede a su construcción. El proceso de construcción del clasificador, tiene un coste computacional elevado y requiere paciencia. Tras finalizar este proceso, se dispone de un clasificador en XML listo para ser utilizado.

El clasificador XML consiste en un árbol binario orientado hacia la derecha. Cuando se tiene una imagen del movimiento ésta se convierte a una escala de grises y al tamaño adecuado para el clasificador. Después será analizada por el clasificador en busca de dicho objeto. El movimiento pertenecerá a dicho objeto solo si se llega hasta el último nodo hoja derecho (figura 3.21).

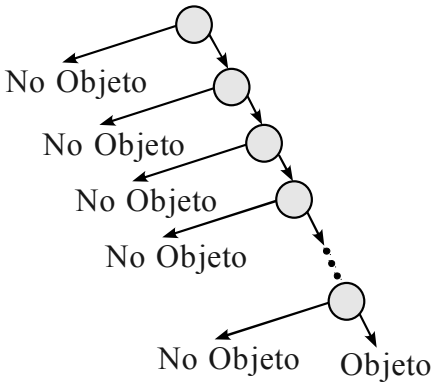

FIGURA 3.21: Representación de un clasificador en XML.

Clasificación basada en aprendizaje supervisado

Existe una gran variedad de algoritmos basados en un aprendizaje supervisado. Éstos se diferencian de la clasificación automática en que se le proporciona información ya válida. Esta información válida forma parte del conocimiento del sistema inteligente y es proporcionada por un experto.

Existen dos formas de realizar un aprendizaje supervisado [Mat96].

- Construir un clasificador basándose en el conocimiento de expertos. Se utilizan expertos en reconocer las características más discriminantes de los objetos a clasificar, y teniendo esas características se modelan mediante reglas. De tal forma que cuando un objeto quiera clasificarse en un tipo se evalúa el conjunto de reglas que deben de cumplirse para dicho tipo.

- Construir un clasificador basándose en un ejemplo. A veces existen dominios donde aquellas características son difíciles de obtener o todavía son desconocidas. Luego la única alternativa de construir un clasificador es basándose en ejemplos. Los seres humanos tenemos la capacidad de poder aprender basándonos en ejemplos. De dichos ejemplos se pueden deducir patrones asociados a cada tipo de objeto, de tal forma que se pueda construir un clasificador utilizando dichos patrones. En [Bro] han utilizado esta técnica para solventar el problema de la sensación de profundidad/lejanía. Esta sensación de lejanía o profundidad hace que los objetos vayan aumentando su tamaño a medida que se van acercando a la localización de la cámara que los capta. Mediante un estudio riguroso de los objetos a clasificar se ha obtenido información sobre sus tamaños, las zonas por donde se mueven, velocidades, histogramas que definen la silueta, etc. Toda esta información es recogida mediante patrones para construir el propio clasificador.

3.5.4. Métodos de tracking

El *tracking* es el proceso por el cual se realiza el seguimiento de los objetos, conociendo la proyección y perspectiva de la cámara que está realizando la monitorización. Teniendo en cuenta los parámetros de calibración de la cámara, distorsión y zoom existen una variedad de algoritmos que permiten obtener una estimación de la posición de los objetos que son captados en los frames de la cámara.

Los algoritmos de tracking se pueden clasificar en dos vertientes:

Algoritmos de tracking basados en modelos.

Estos algoritmos son los más utilizados debido a que son mucho más robustos que los basados en características. Realizan el seguimiento de un objeto del que previamente se conoce su modelo. Conociendo el modelo 2D o 3D del objeto se realiza un seguimiento de éste cuando aparece en la imagen captada por la cámara. En esta vertiente existen multitud de algoritmos, que los podemos agrupar en función del número de cámaras que van a realizar el seguimiento. Como estos métodos se basan en el modelo 2D o 3D del objeto a seguir, si el sistema de videovigilancia dispone de una cámara solamente se podría trabajar con un modelo 2D. Algunos algoritmos destacados cuando el sistema esta formado por *una sola cámara* son:

- **POSIT** [DD95] Este algoritmo es atribuido a Daniel DeMenthon y Larry S. Davis. Consiste primero en obtener una estimación de la matriz de rotación y posición del

objeto. Esto se realiza mediante un algoritmo llamado *POS (Pose from Orthography and Scaling)* que calcula la posición y rotación del objeto teniendo en cuenta la proyección ortográfica escalada de la cámara. Para ello se tienen en cuenta cuatro o más puntos que sean *no coplanares*[5]. Una vez que se tiene una estimación de la posición, ésta es refinada mediante el algoritmo *POSIT (POS with ITerations)* que consiste en seguir invocando al algoritmo POS corrigiendo la distorsión de la imagen e ir refinando la estimación de la posición hasta que el error cometido sea menor que un umbral.

- **Lowe's pose estimator** [AE10] Este algoritmo trata de calcular la rotación y posición del objeto simplificando las ecuaciones de la proyección. Teniendo en cuenta las coordenadas 3D reales del objeto y viendo su correspondencia con las coordenadas 2D detectadas en el frame se obtienen un conjunto de ecuaciones. De la resolución de ese sistema de ecuaciones se obtiene la posición y rotación del objeto expresados sobre el sistema de referencia de la cámara.

En cuanto a los algoritmos basados en modelos utilizados en sistemas formados por *varias cámaras* son:

- **Reconstrucción proyectiva-euclidea** [Ver] Consiste en obtener las proyecciones de las cámaras a partir de las diferentes vistas que ofrecen. Este algoritmo permite obtener una estimación de la proyección de la cámara mediante una reconstrucción proyectiva. La reconstrucción proyectiva consiste en juntar cada una de las vistas de las cámaras basándose en varias correspondencias entre ellas. Esta reconstrucción y mosaico de cada una de las vistas se realiza por medio de *homografías* [Lil03]. De forma que una vez que se tiene la reconstrucción proyectiva, si se conocen las coordenadas 3D de determinados puntos se pueden obtener las proyecciones de las cámaras.

Algoritmos de tracking basados en características.

Los métodos basados en características realizan seguimientos basándose en las características o partes de los objetos. Éstos son más eficientes que los basados en modelos, pero son menos robustos. Para hacer uso de estos algoritmos primero se deben definir esas características. Estas características pueden ser:

- **Basado en regiones** [KB90] Este tipo de algoritmo es utilizado en algoritmos de segmentación de tipo *background*. Consisten en mantener un fondo dinámico y se

[5]Los puntos son no coplanares si no existe un plano capaz de contenerlos a todos.

realiza el seguimiento de aquellas regiones resultantes de los cambios entre el fondo dinámico y el frame actual. Las ventajas de estos algoritmos son que permiten realizar el seguimiento de objetos en entornos cambiantes, pero no tienen una fiabilidad buena cuando se producen oclusiones ni tampoco obtienen información sobre la posición 3D de los objetos.

- **Basado en contornos activos** [IB96] Éstos se basan en realizar un seguimiento de los contornos/bordes bien definidos de los objetos. Los basados en contornos activos son más eficientes que los basados en regiones y funcionan bien bajo oclusiones parciales de los objetos. Pero la desventaja recae en que se debe realizar una buena inicialización de los contornos que se siguen debido a que son muy sensibles a dicha inicialización.

- **Basado en rasgos** [PN94] Realizan el seguimiento de los objetos basándose en los rasgos de los objetos. Estos rasgos se pueden definir de una forma global basándose en el área, perímetro, rango de colores, etc. O de una forma local, en el que se atiende a los segmentos de líneas, curvas y vértices. La principal desventaja de esta clase de algoritmos es que la distorsión de la lente perjudica en el reconocimiento de estos rasgos y produce errores en el seguimiento. Además tampoco permite la obtención de la posición 3D de los objetos seguidos.

Estos algoritmos para hacerlos más robustos se suelen combinar con algún otro algoritmo de predicción de la posición. De esta forma se hacen más robustos cuando se dan oclusiones, o cuando la velocidad que llevan los objetos es elevada. Uno de los algoritmos de predicción de la posición más utilizado es el *filtro de Kalman* [Pet00]. El *filtro de kalman* es un algoritmo recursivo en el que dada una región donde un algoritmo basado en características detecto movimiento, permite obtener un grado de confianza en indicar que en esa región del siguiente frame existirá un contorno o un rasgo que se estaba siguiendo. Este algoritmo permite la predicción del estado futuro de la trayectoria del objeto. Su funcionamiento consiste en dar una predicción teniendo en cuenta la información hasta ese momento y luego observar los resultados que se producirían de esa predicción para reducir el error cometido.

3.6. Herramientas de visión por computador y de diseño 3D

En esta sección se darán a conocer algunas de las herramientas actuales para utilizar técnicas de visión por computador y gráficos-3D.

3.6.1. OpenCV

OpenCV (*Open Source Computer Vision Library*[6]) [BK08] es una biblioteca abierta para aplicaciones en tiempo real que requieran de soporte de visión por computador. Da soporte para aplicaciones desarrolladas en lenguajes *C* y *C++*. Está diseñado para que la gente pueda desarrollar aplicaciones de visión por computador de forma rápida y sencilla.

OpenCV abarca un gran ámbito de áreas de aplicación (calibración de cámara, visión por computador, interfaces de usuario, seguridad, robótica, etc). Su nacimiento surgió del proyecto de un grupo de investigación de Intel para desarrollar aplicaciones de uso intensivo de la CPU. Uno de los investigadores cuando estaba visitando universidades, se dio cuenta del potencial que tenían algunos grupos universitarios (MIT Media Lab) en visión por computador. De forma que se decidió dar la oportunidad a éstos y así nació la idea de OpenCV.

OpenCV está formado por cuatro componentes:

- **CV**. Este componente contiene los algoritmos básicos para el procesamiento de las imágenes y de alto nivel en la visión por computador.
- **MLL**. Este componente es una librería de aprendizaje, contiene tanto herramientas para clúster como una variedad de clasificadores estadísticos.
- **HighGUI**. Este componente se encarga de las operaciones de entrada y salida, junto con las operaciones de almacenamiento y reproducción de imágenes y videos.
- **CXCore**. Éste es el núcleo sobre el que se sustentan los anteriores componentes, y en él se encuentran las estructuras básicas de datos, el soporte para XML, y funciones de dibujado.

[6]En español, biblioteca de visión por computador abierta

3.6.2. Blender

Blender [Chr06] es un programa de código abierto para el desarrollo de animación, renderizado y videojuegos. Fue desarrollado por la *Blender Foundation*[7] como la mayoría de programas de desarrollo de videojuegos, su curva de aprendizaje es costosa pero una vez que se adquiere destreza en un par de semanas dicha curva desaparece (figura 3.22). Esta herramienta será de gran utilidad durante el desarrollo de *TraceMon* para poder probar y refinar la funcionalidad de la aplicación. Se hace uso de *Blender* para que se tenga un control del entorno monitorizado de forma que la información de posicionamiento de las cámaras y los resultados del *raytracer* puedan ser contrastados con la información proporcionada por *Blender*.

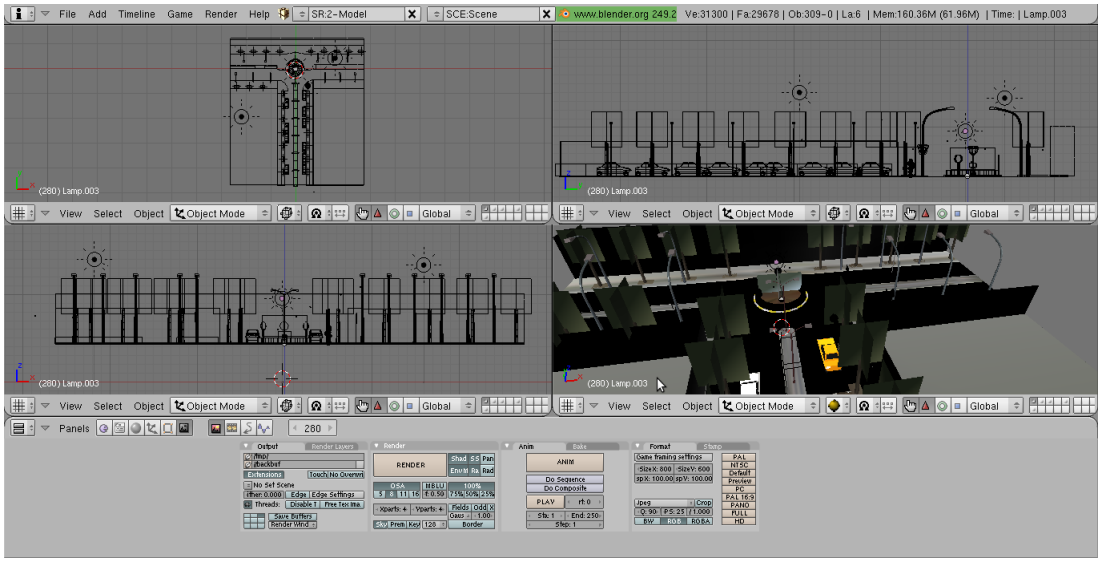

FIGURA 3.22: Representación de la interfaz de Blender.

3.6.3. OpenGL

OpenGL (*Open Graphic Library*[8]) [DS10] es una *API*[9] para poder representar objetos y realizar operaciones sobre éstos en aplicaciones 3D. Esta diseñado para una gran variedad de plataformas hardware, y aunque no proporciona instrucciones para construir objetos complejos (casas, personas, coches, etc), si proporciona primitivas básicas para poder modelarlos (líneas, puntos, polígonos, curvas, etc). A parte de estas primitivas, existen

[7] Una organización independiente con el objetivo de crear un acceso público de la tecnología 3D.

[8] En español, biblioteca gráfica abierta.

[9] Interfaz de programación de aplicaciones, que proporciona cierta capa de abstracción acta para ser utilizada en cualquier desarrollo de aplicación.

multitud de primitivas más para modelar las texturas de los objetos, el color, la orientación de los objetos, etc.

OpenGL utiliza una sintaxis que lo hace independiente del lenguaje de programación, y su funcionamiento es como el de una máquina de estados. Mantiene un gran número de estados, los cuales se pueden habilitar o deshabilitar con *glEnable()* y *glDisable()*. Entre ellos destacar:

- El estado que mantiene el color actual, de forma que todo lo que se dibuje en ese momento tendrá el color que marque este estado.
- El estado que mantiene el punto de vista y las transformaciones de proyección. Éste contiene dos matrices que representan la transformación que se le va aplicar a la perspectiva y a los objetos que se van a pintar. Para modificar dichas matrices y recuperar antiguas matrices, OpenGL implementa un sistema de pilas, donde estas matrices *MODELVIEW_MATRIX* y *PROJECTION_MATRIX* se pueden apilar o desapilar con los comandos: *glPushMatrix()* y *glPopMatrix()*.
- El estado que mantiene propiedades de los materiales sobre los objetos. Se crean materiales asociándole un identificador, y éste se puede reutilizar indicando dicho identificador.
- El estado que mantiene la información sobre las propiedades de las fuentes de luz.

Y por último destacar que a diferencia de otras APIs para aplicaciones-3D, OpenGL utiliza un sistema de referencia donde el *eje Z* es saliente al plano formado por los *ejes X e Y* ver figura 3.23.

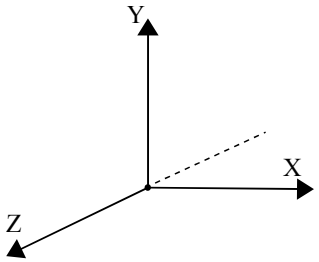

FIGURA 3.23: Representación del sistema de referencia de OpenGL.

4

Método de trabajo

En este capítulo se describe el ciclo de vida que se ha utilizado para desarrollar este proyecto junto con el conjunto de herramientas para llevarlo a la práctica.

4.1. Metodología de desarrollo

Para la elección de la metodología se ha pensado en un **modelo en cascada incremental** [Som05]. Los motivos de la elección por este modelo son porque permite un desarrollo iterativo e incremental, no se requieren de todos los requisitos en un principio, ofrece una planificación sencilla y existe un contacto entre el alumno y los directores del proyecto para ir refinando los prototipos hasta que se concluye con el producto final eliminando el riesgo de que el alumno y los directores del proyecto no estén conformes.

Dicho modelo se ajusta perfectamente en el ámbito de este proyecto porque el alumno y los directores del proyecto en un principio tienen ideas generales de lo que quieren, pero a medida que se van viendo prototipos van surgiendo nuevas ideas o requisitos. En la figura 4.1 se puede ver la estructura que sigue este modelo.

Se puede ver que la primera fase de esta metodología es la fase de análisis. En dicha fase se debe de plasmar mediante algún documento los requisitos funcionales que va a cumplir la aplicación. Para ello se realizó un *diagrama de casos de uso* donde se recogían todos esos requisitos funcionales (figura 4.2).

Teniendo ese diagrama de casos de uso se pasa a la fase de realizar un diseño preliminar. A la hora de realizar este diseño preliminar se era consciente de que una mala decisión en

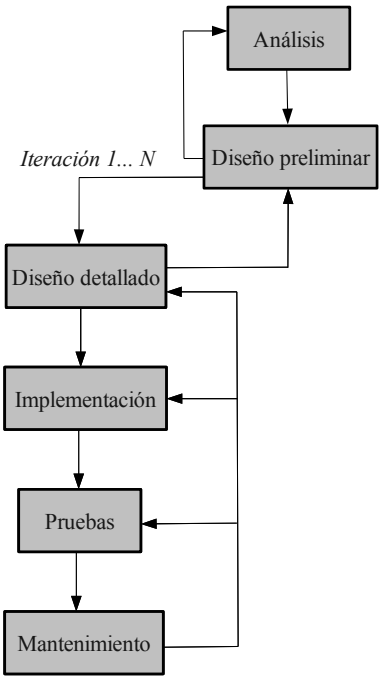

FIGURA 4.1: Estructura de un modelo en cascada incremental.

éste, repercutiría negativamente en las siguientes fases de diseño, implementación, pruebas y mantenimiento. Luego se optó por seguir un *patrón arquitectónico multicapa* para poder separar y agrupar responsabilidades.

Este patrón consiste en descomponer el sistema en una serie de capas en las que se interrelacionan solo con sus adyacentes. En general el patrón *multicapa* presenta tres capas:

- **Presentación**: Capa en la que se encuentran las interfaces gráficas por las que el usuario interacciona con la aplicación.
- **Dominio**: Capa en la que se encuentra toda la lógica del ámbito de la aplicación.
- **Persistencia**: Capa en la que se encuentra toda la lógica relacionada con la persistencia. La persistencia consiste en guardar el estado en el que se encuentra la aplicación de tal forma que, si existen apagones de luz, la información del sistema no se pierda.

Si se atiende al diagrama de casos de uso de *TraceMon* no existe ninguno relacionado con la capa de persistencia. Luego el primer diseño preliminar que surgió de clasificar los requisitos en capas se puede ver en la figura 4.3.

Se puede ver en la capa de presentación que existe una *interfaz gráfica principal* que es conectada con todas las subinterfaces gráficas que permiten al usuario realizar cada uno de

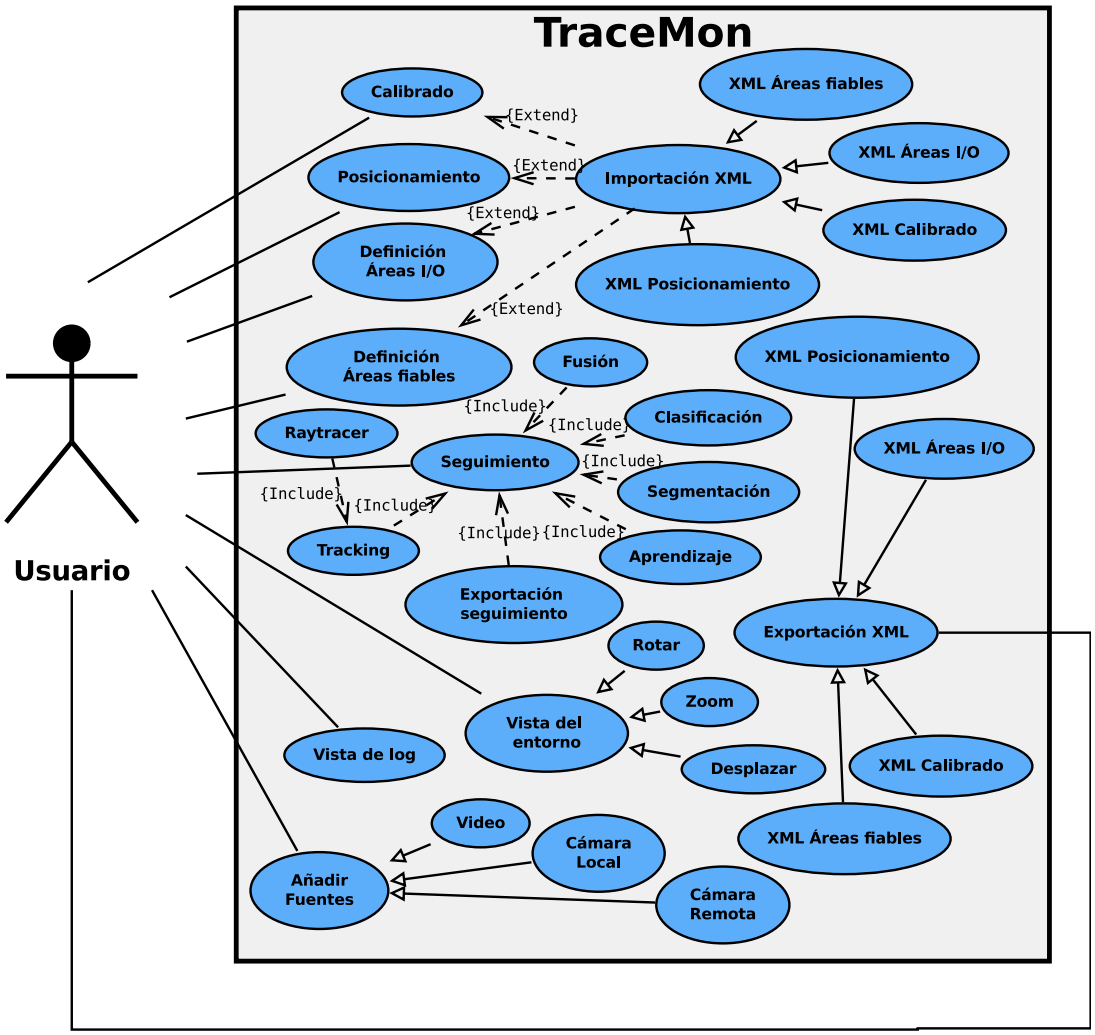

FIGURA 4.2: Diagrama de casos de uso.

los procesos de importación, exportación, inserción de fuentes, calibrado, posicionamiento, selección de áreas y tracking. Es mediante esta interfaz por donde el usuario solicita al sistema el proceso a realizar y éste le responderá con la interfaz gráfica correspondiente.

Y en cuanto a la capa de dominio existe un módulo encargado de la *gestión de los agentes* que está conectado con cada uno de los agentes que realizan los procesos que son solicitados por el usuario desde la capa de presentación.

Una vez que se tuvo este diseño preliminar comenzaron las iteraciones de la metodología escogida. Los detalles de las iteraciones y planificación durante el desarrollo de la aplicación se describen en el capítulo 6. Y en cuanto a los detalles de los patrones utilizados, diagramas de diseño más detallados, diagramas de secuencia y la especificación de los algoritmos de los procesos se describen en el capítulo 5. Por último los resultados obtenidos de las pruebas se describen también en el capítulo 6.

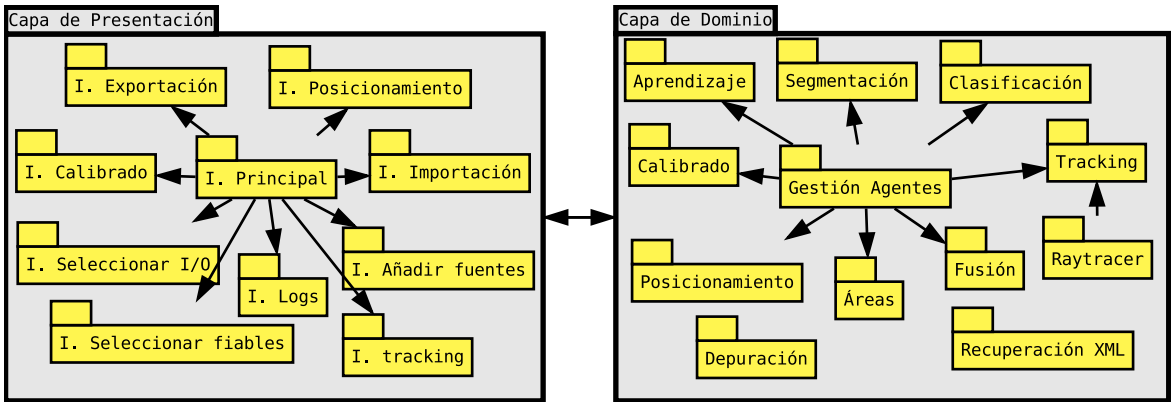

FIGURA 4.3: Diseño preliminar siguiendo un patrón multicapa.

4.2. Herramientas

4.2.1. Lenguaje de programación

TraceMon se ha implementado haciendo uso del lenguaje de programación **C++** [BE04]. Se ha optado por C++ porque es un lenguaje de programación que permite hacer uso de bibliotecas fundamentales para los procesos de calibración, posicionamiento y segmentación. Es un lenguaje eficiente para poder desarrollar aplicaciones en tiempo real y está orientado a objetos permitiendo una capa de abstracción, encapsulación y modularización en la implementación.

4.2.2. Hardware

El desarrollo y las pruebas de *TraceMon* se han realizado sobre una arquitectura hardware: *i386* de 64 bits y con 4GB de memoria RAM. En cuanto a los componentes hardware para el sistema de videovigilancia se han utilizado una webcam modelo *Logitech Sphere AF*[1], una cámara remota modelo *Axis207W*[2] y otra cámara remota modelo *Axis213r*[3].

4.2.3. Software

En esta sección se detallan todas las herramientas, aplicaciones y bibliotecas utilizadas para el desarrollo de *TraceMon*.

[1] http://www.logitech.com
[2] http://www.axis.com/products
[3] http://www.axis.com/products

Herramientas de diseño y modelado 3D

- **Blender 3D**: Se ha utilizado esta herramienta para modelar y diseñar entornos 3D con el fin de realizar pruebas al sistema mientras se está desarrollando. Esto es fundamental porque desplegar toda la infraestructura para un sistema de este tipo es complejo y primero debe de asegurarse que los módulos funcionan correctamente sobre un entorno totalmente controlado. La versión que se ha utilizado *2.49b*[4].

Software de desarrollo

- **Eclipse**: Se ha utilizado como principal entorno de desarrollo proporcionando soporte para proyectos en C++. La versión que se ha utilizado es *Europa*[5].
- **Make**: Herramienta que se ha utilizado para el proceso de compilación de los archivos fuente del proyecto. La versión que se ha utilizado es la *3.81*[6].
- **GCC**: Es una colección de compiladores, en especial se ha utilizado *g++* como compilador para el proyecto. La versión que se ha utilizado es *4.4.5*[7].
- **GDB**: Se ha utilizado como depurador del código y así detectar errores de implementación. La versión que se ha utilizado es *7.2* [8].
- **Emacs**: Se ha utilizado como herramienta principal para redactar la documentación. La versión que se ha utilizado es *23.1.1* [9].
- **Glade3**: Se ha utilizado como herramienta para diseñar las interfaces gráficas del proyecto. La versión que se ha utilizado es *3.6.7* [10].

Sistema Operativo

- **Ubuntu**: Se ha utilizado como sistema operativo para desarrollar este proyecto. Todas las herramientas y aplicaciones se han utilizado bajo este sistema operativo. En particular se ha utilizado la versión *10.10 de Maverick Meerkat* la cual es considerada una versión estable.

[4] http://download.blender.org/release/
[5] http://eclipse.org/europa/
[6] http://ftp.gnu.org/gnu/make/
[7] http://gcc.gnu.org/gcc-4.4/
[8] http://www.gnu.org/software/gdb/
[9] http://packages.ubuntu.com/natty/emacs23
[10] http://ftp.gnome.org/pub/GNOME/sources/glade3/3.6/

Control de Versiones

- **ProyectHosting**: Es un servicio de *google code* que permite alojar todo proyecto de software libre. Y mediante la herramienta *RabbitVCS SVN* se ha llevado a cabo la gestión y el control de las versiones del proyecto.

Documentación

- **Doxygen**: Se ha utilizado para generar la documentación del código fuente de *TraceMon*. La versión que se ha utilizado es *1.7.1* [11].
- **LaTeX**: Herramienta[12] que se ha utilizado para la elaboración de la memoria de este proyecto.
- **LibreOffice Draw**: Aplicación que se ha utilizado para generar las imágenes vectoriales de este documento. La versión que se ha utilizado es *3.3.2*[13].
- **Gimp**: Herramienta[14] que se ha utilizado para generar las imágenes no vectoriales de este documento.
- **Dia**: Herramienta[15] que se ha utilizado para la creación de diagramas UML. La versión que se ha utilizado es *0.97.1*.

Bibliotecas

- **OpenCV**: Biblioteca que se ha utilizado para los procesos de calibrado, posicionamiento, segmentación, *tracking* y tratamiento de las imágenes. La versión que se ha utilizado es *2.1.0*[16]. Para la instalación de esta biblioteca sobre *Ubuntu 10.10* se ha seguido el tutorial de `http://www.samontab.com/web/2010/04/installing-opencv-2-1-in-ubuntu/`.
- **OpenGL**: Biblioteca que se ha utilizado para representar en 3D el entorno monitorizado, junto con las cámaras y los objetos detectados. Esta biblioteca contiene un conjunto de dependencias de forma que se ha seguido para su instalación el tutorial de

[11] http://www.icewalkers.com/Linux/Software/57870/Doxygen.html
[12] https://help.ubuntu.com/community/LaTeX
[13] http://www.ubuntu-es.org/node/143190
[14] http://packages.ubuntu.com/maverick/gimp
[15] http://packages.ubuntu.com/maverick/dia
[16] http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.1/OpenCV-2.1.0.tar.bz2/download

`http://www.nemediano.com.mx/2007/instalando-las-librerias-de-opengl-y-glut-en-ubuntu/`.

- **TinyXml**: Biblioteca[17] que se ha utilizado para realizar la exportación e importación de la información en archivos XML.
- **PThreads**: Biblioteca[18] que se ha utilizado para la gestión y control de los hilos de ejecución de los propios agentes que intervienen en las fases del sistema de vigilancia.
- **GTK+**: Biblioteca[19] que se ha utilizado para la creación y la gestión de las interfaces gráficas del proyecto. Para la instalación de esta biblioteca se ha seguido el tutorial de `http://developer.gnome.org/gtk3/stable/gtk-building.html`.
- **GtkGLExt**: Biblioteca que se ha utilizado para integrar OpenGL en las interfaces GTK+ del proyecto. Para la instalación de esta biblioteca se requieren instalar los paquetes *libgtkglext1-dev*, *libgtkglext1*, *libgtkglextmm-x11-1.2-0* y *libgtkglextmm-x11-1.2-dev*.

[17] http://sourceforge.net/projects/tinyxml/
[18] http://staff.science.uva.nl/ bterwijn/Projects/PThread/
[19] http://ftp.gnome.org/pub/gnome/sources/gtk+/3.0/

5

Arquitectura de TraceMon

En este capítulo se describe la arquitectura de *TraceMon* siguiendo un enfoque *Top-Down*. Este enfoque consiste en partir de un esquema de los módulos que forman la arquitectura e ir entrando en detalle en cada uno de ellos.

Basándose en el diagrama de casos de uso (figura 4.2) y en el diseño preliminar (figura 4.3) se obtuvo el esquema de la arquitectura de *TraceMon* (ver figura 5.1).

Este esquema consta de cinco módulos:

1. **Módulo de entrada**. Contiene toda la funcionalidad relacionada con la entrada de información a la aplicación. El usuario puede indicar el número de fuentes a utilizar, archivos de calibración, posicionamiento, definición de áreas I/O o fiables y cualquier evento de ratón para poder realizar acciones sobre la aplicación. Como es de esperar esta información debe de ser procesada y por ello este módulo se comunica con el módulo de procesamiento y módulo de gestión de agentes.

2. **Módulo de procesamiento**. Este módulo se encarga de realizar cada una de las funcionalidades que debe cumplir la aplicación. Durante la realización de estas funcionalidades este módulo será gestionado por el módulo de gestión de agentes y requerirá de la información que le proporcione el módulo de entrada en determinados momentos. Entre las funcionalidades que debe realizar se encuentran el calibrado, posicionamiento, definición de áreas, segmentación, seguimiento, tracking, fusión y depuración.

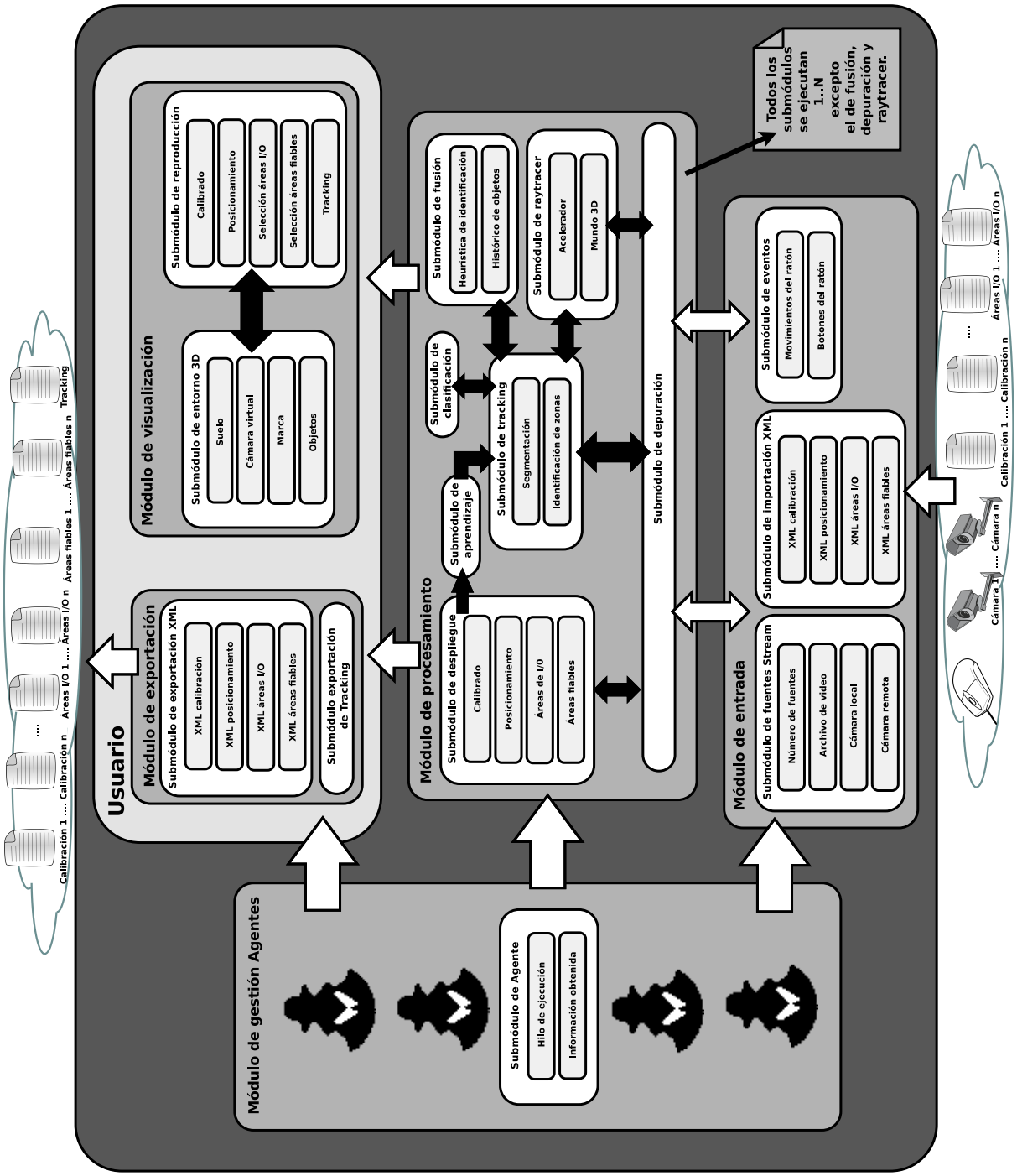

FIGURA 5.1: Esquema general de la arquitectura.

3. **Módulo de gestión de agentes**. Este módulo es el encargado de la gestión de los agentes. Para ello los agentes son componentes que forman este módulo y son los encargados de recoger la información del módulo de entrada, procesarla mediante los algoritmos y procesos del módulo de procesamiento y exportar o visualizar la información obtenida de cara al usuario.

4. **Módulo de visualización**. Este módulo se encarga de visualizar la información que le van indicando los subprocesos del módulo de procesamiento. Estos subprocesos son controlados y gestionados por el módulo de gestión de agentes el cuál también se encarga de gestionar la información que se tiene que visualizar. En cuanto a esa información que se pretende mostrar al usuario son los resultado obtenidos de cada una de las fases de calibración, posicionamiento, áreas definidas, objetos detectados, el entorno a monitorizar, etc.

5. **Módulo de Exportación**. Contiene la funcionalidad relacionada con la exportación de la información de los procesos hacia el exterior de la aplicación. Dicha información le es proveniente de la comunicación con el módulo de procesamiento y de la gestión por el módulo de gestión de agentes.

5.1. Módulo de entrada

El módulo de entrada es el encargado de recibir y percibir los eventos y la información que el usuario quiere que procese la aplicación. Este módulo se descompone en tres submódulos: *submódulo de fuentes de stream*, *submódulo de importación XML* y *submódulo de eventos*. Entre los submódulos del módulo de entrada no hay conexiones entre ellos, pero sí hay conexión con el módulo de procesamiento para que éste pueda procesar la información que el usuario va proporcionando al módulo de entrada.

5.1.1. Submódulo de fuentes de *stream*

Este submódulo es el encargado de ejecutar la funcionalidad sobre el registro de las fuentes *stream* que va a tener el sistema de *tracking*. La forma de interaccionar con este submódulo consiste en especificar el número de fuentes con las que trabajar y una vez indicado el número de fuentes, el usuario puede indicar el tipo de éstas (*archivo de vídeo*, *cámara remota* y *cámara local*).

El principal problema a la hora de diseñar una aplicación que permita un número ilimitado de fuentes *stream* es que el espacio de visualización de éstas viene limitado por la resolución de pantalla. Para solucionar este problema se pensó en limitar el número de fuentes *stream* a tres. Luego la aplicación trabaja con un máximo de tres fuentes debido a la limitación de la resolución, pero este submódulo está diseñado e implementado para poder trabajar con un número indefinido de fuentes gracias a que éste utiliza estructuras dinámicas.

Después de especificar el número de fuentes con las que trabajar, se especifican sus tipos. Estos tipos pueden ser *archivo de vídeo*, *cámara remota* y *cámara local*. Luego para el diseño de este submódulo se tenía que tener en mente la utilización de una capa de abstracción que permitiese manejar las fuentes independientemente del tipo que fuesen. Es decir, la forma de obtener la información de la fuente debía de ser igual para todos sus tipos. Para solventar este problema se utiliza *openCV* (sección 3.6.1). Éste ofrece el objeto *CvCapture* el cual permite trabajar con la fuente de forma independiente de su tipo. Lo único que se diferencia es la forma de inicializar este objeto *CvCapture*.

Para una fuente de tipo *archivo de vídeo* se realiza mediante:

```
1 CvCapture* capture = cvCreateFileCapture("Ruta_del_archivo_video");
```

Para una fuente de tipo *cámara remota* se realiza mediante:

```
1 CvCapture* capture = cvCreateFileCapture("URL_camara_remota");
```

Para una fuente de tipo *cámara local* se realiza mediante:

```
1 CvCapture* capture = cvCaptureFromCAM(Indice_del_dispositivo);
```

Por otro lado existe una pequeña restricción a la hora de elegir los tipos de fuentes. La razón es que no tiene sentido estar realizando un seguimiento de una fuente en tiempo real con otras fuentes en diferido. Por lo tanto, no se puede trabajar con fuentes de *archivo de vídeo* y *cámara remota o local* al mismo tiempo. Es decir, si a la propia aplicación se le indica una primera fuente de tipo *archivo de vídeo*, el resto de fuentes que sean conectadas a la aplicación deben de ser de tipo *archivo de vídeo*. Y al contrario si se añade una fuente de tipo *cámara remota o local* el resto de fuentes tienen que ser de tipo *cámara remota o local* pero no se puede añadir ninguna de *archivo de vídeo*.

Para diseñar esta restricción, el submódulo se basa en el *estado actual* de las fuentes que tiene el sistema. Cuando se pretende añadir una fuente al sistema, se consultan los estados actuales del número de fuentes a añadir para detectar si ya existe alguna fuente añadida. Si existe alguna entonces se chequean sus tipos y así la propia aplicación indica las opciones del tipo de fuente que debe de ser la nueva fuente a añadir.

Un diagrama que representa el funcionamiento de este submódulo se puede ver en la figura 5.2.

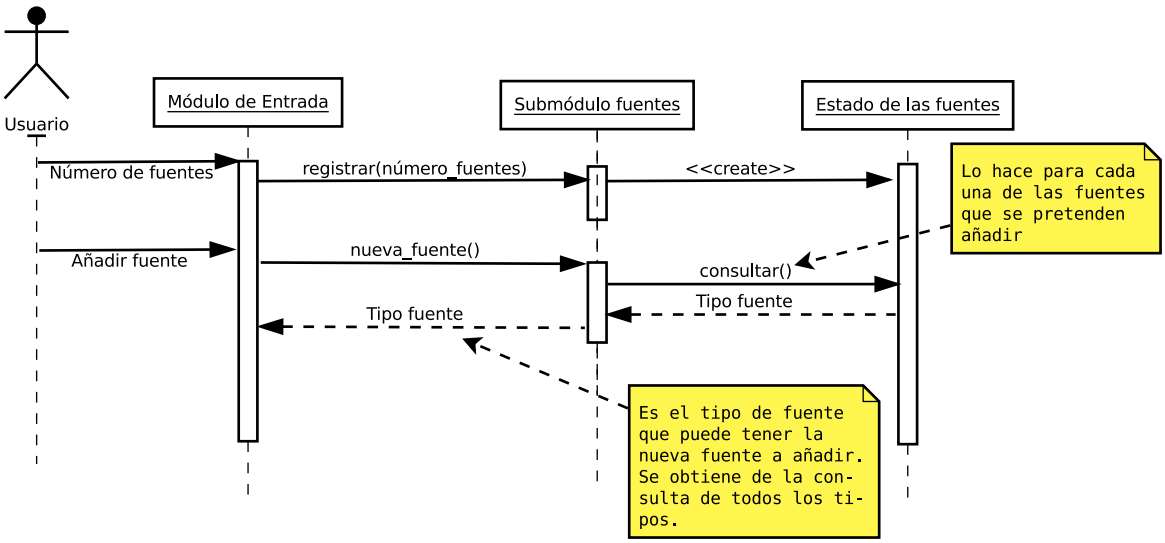

FIGURA 5.2: Diagrama de interacción del submódulo fuentes *Stream*.

5.1.2. Submódulo de importación de archivos XML

Submódulo que se encarga de permitir que el usuario pueda especificar archivos XML con información referente a los procesos de calibrado, posicionamiento, definición de áreas I/O y áreas fiables. Con esto se consigue evitar que el usuario tenga que repetir dichos procesos si simplemente ya los había realizado y el sistema que se desplegó sigue siendo el mismo. Por lo tanto la funcionalidad de este submódulo consiste en poder interpretar la información recogida en los archivos XML. Para ello está diseñado con el objetivo de detectar posibles errores en dicha información. Para llevar a cabo esta lectura se ha hecho uso de la biblioteca *TinyXML*[1] que ofrece una interfaz cómoda para poder realizar la lectura de los elementos del archivo XML. También el uso de esta biblioteca facilita la tarea de detectar errores en dicha estructura.

A continuación se describe la estructura que deben seguir estos archivos XML para que puedan ser procesados de forma correcta.

[1] http://sourceforge.net/projects/tinyxml/

Archivo XML de calibración

Los datos de calibración vistos en la sección 3.5.1 están compuestos por la matriz de la cámara y los parámetros de distorsión. Por lo tanto para almacenar dicha información se pensó en la siguiente estructura:

```
<Calibration>
  <Distortion>
    <!--Distortion values obtained from the calibration.-->
    <radial_distortions_K1>
      <!--valor en decimal-->
    </radial_distortions_K1>
    <radial_distortions_K2>
      <!--valor en decimal-->
    </radial_distortions_K2>
    <tangential_distortions_P1>
      <!--valor en decimal-->
    </tangential_distortions_P1>
    <tangential_distortions_P2>
      <!--valor en decimal-->
    </tangential_distortions_P2>
    <radial_distortions_K3>
      <!--valor en decimal-->
    </radial_distortions_K3>
  </Distortion>
  <Camera_Matrix>
    <!--The values of the camera matrix obtained from the calibration.-->
    <focal_length_in_pixels_of_the_X_axis>
      <!--valor en decimal-->
    </focal_length_in_pixels_of_the_X_axis>
    <matrix_element_0_1>
      0,000000
    </matrix_element_0_1>
    <principal_point_x>
      <!--valor en decimal-->
    </principal_point_x>
    <matrix_element_1_0>
      0,000000
    </matrix_element_1_0>
    <focal_length_in_pixels_of_the_Y_axis>
      <!--valor en decimal-->
    </focal_length_in_pixels_of_the_Y_axis>
    <principal_point_y>
        <!--valor en decimal-->
    </principal_point_y>
    <matrix_element_2_0>
      0,000000
    </matrix_element_2_0>
    <matrix_element_2_1>
      0,000000
    </matrix_element_2_1>
    <matrix_element_2_2>
      1,000000
    </matrix_element_2_2>
  </Camera_Matrix>
</Calibration>
```

LISTADO 5.1: Formato del archivo XML de calibración

En cuanto a los mecanismos de detección de error, éstos detectan si falta algún componente y si la información que contiene el componente es correcta, es decir, es un valor decimal.

Archivo XML de posicionamiento

En la sección 3.5.1 se explicó cómo se iba a representar la información sobre la posición y orientación de la cámara. Esta información está compuesta por la situación donde se encuentra el centro de la cámara y la matriz de rotación $3x3$ de la cámara. Por lo tanto para almacenar dicha información se pensó en la siguiente estructura:

```
<Position>
  <Camera_Center>
    <!--Center values obtained from the 3D position.-->
    <camera_center_x>
      <!--valor en decimal-->
    </camera_center_x>
    <camera_center_y>
      <!--valor en decimal-->
    </camera_center_y>
    <camera_center_z>
      <!--valor en decimal-->
    </camera_center_z>
  </Camera_Center>
  <Rotation_Matrix>
    <!--The values of the rotation matrix obtained from the calibration.-->
    <rotation_matrix_0_0>
      <!--valor en decimal-->
    </rotation_matrix_0_0>
    <rotation_matrix_0_1>
      <!--valor en decimal-->
    </rotation_matrix_0_1>
    <rotation_matrix_0_2>
      <!--valor en decimal-->
    </rotation_matrix_0_2>
    <rotation_matrix_1_0>
      <!--valor en decimal-->
    </rotation_matrix_1_0>
    <rotation_matrix_1_1>
      <!--valor en decimal-->
    </rotation_matrix_1_1>
    <rotation_matrix_1_2>
      <!--valor en decimal-->
    </rotation_matrix_1_2>
    <rotation_matrix_2_0>
      <!--valor en decimal-->
    </rotation_matrix_2_0>
    <rotation_matrix_2_1>
      <!--valor en decimal-->
    </rotation_matrix_2_1>
    <rotation_matrix_2_2>
      <!--valor en decimal-->
    </rotation_matrix_2_2>
  </Rotation_Matrix>
</Position>
```

LISTADO 5.2: Formato del archivo XML de posicionamiento

En cuanto a los mecanismos de detección de error, éstos son los mismos que para el archivo de calibración. Detectan si falta algún componente y si la información que contiene el componente es correcta.

Archivo XML de áreas I/O

Antes de empezar a describir la estructura que se ha utilizado para almacenar la información de las áreas de entrada/salida. Se va a indicar como está formada un área y los tipos que existen. Las áreas son definidas sobre los píxeles de la vista de la fuente *stream*. Por lo tanto éstas vienen definidas por el conjunto de píxeles que forman dicha área. El hecho de especificar las áreas de entrada/salida sobre las vistas que tiene cada fuente permite que la aplicación conozca las zonas donde los objetos aparecen y desaparecen. Esto es de gran utilidad en el proceso de asignación de identificadores a los objetos detectados y eliminación de dichos identificadores cuando los objetos abandonan el entorno. En cuanto a la forma de clasificar estas áreas se pueden dar cuatro tipos de zonas:

1. **Zona de entrada**. Representa a las áreas de la vista en las que los objetos aparecen en el entorno por primera vez.
2. **Zona de salida**. Representa a las áreas de la vista en las que los objetos desaparecen al salir del entorno.
3. **Zona de entrada y salida**. Representa a las áreas de la vista que se comportan como entrada y salida.
4. **Zona neutra**. Representa a las áreas que no son de entrada ni salida.

Teniendo en cuenta la forma en que se describen las áreas y los tipos de zonas a las que pueden pertenecer, el diseño que se ha utilizado para la estructura del archivo XML es el siguiente:

```
<Input_area>
  <!--Definition of the pixels that belong to the input area.-->
  <Pixel>
    <X>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </X>
    <Y>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </Y>
  </Pixel>
    ...
    ...
  <Pixel>
    <X>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </X>
    <Y>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </Y>
  </Pixel>
</Input_area>
<Output_area>
  <!--Definition of the pixels that belong to the output area.-->
  <Pixel>
    <X>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
```

```
27        </X>
28        <Y>
29          <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
30        </Y>
31      </Pixel>
32        ...
33        ...
34      <Pixel>
35        <X>
36          <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
37        </X>
38        <Y>
39          <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
40        </Y>
41      </Pixel>
42    </Output_area>
43    <Input_Output_area>
44      <!--Definition of the pixels that belong to the input-output area.-->
45      <Pixel>
46        <X>
47          <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
48        </X>
49        <Y>
50          <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
51        </Y>
52      </Pixel>
53        ...
54        ...
55      <Pixel>
56        <X>
57          <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
58        </X>
59        <Y>
60          <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
61        </Y>
62      </Pixel>
63    </Input_Output_area>
```

LISTADO 5.3: Formato del archivo XML de áreas I/O

Como se puede apreciar la zona neutra no aparece porque se sobreentiende que el resto de píxeles que no se encuentran asociados a ninguna otra zona corresponden a la zona neutra. En cuanto a los mecanismos de detección de error se detecta que los componentes estén bien formados y que los valores de los píxeles sean enteros positivos y que estén dentro del tamaño de la vista de la fuente *stream*. Para conocer los detalles sobre el proceso de definición de estas áreas ver sección 5.2.1

Archivo XML de áreas fiables

En los sistemas de monitorización multicámara suele pasar que existen unas zonas que son más visibles que otras, o que las fuentes *stream* tienen diferentes calidades de imagen o que incluso la vista y la perspectiva de una determinada fuente ofrece mejor información que otras vistas. Para permitir que determinadas partes de una vista sean más importantes que otras se ha pensado en utilizar un *parámetro de fiabilidad*. Este parámetro es de gran utilidad porque a la hora de mezclar la información de los objetos detectados por las fuentes, se valorará más la información que provenga de vistas con mayor fiabilidad. Por ejemplo

si una fuente presenta información de que ha detectado movimientos en una zona con baja fiabilidad, frente a otra que presenta la información de los mismos movimientos pero con mejor fiabilidad, se tendrá en cuenta mucho más ésta última que la otra. Los criterios utilizados para fusionar la información teniendo en cuenta la fiabilidad se describen en la sección 5.2.6.

Se decidió representar el parámetro de fiabilidad con once grados de fiabilidad. Dichos grados van en incrementos de 0.1 dentro del intervalo $[0, 1]$. Siendo la fiabilidad máxima 1 y la fiabilidad mínima 0.0. La fiabilidad 0.0 permite ignorar completamente lo que ocurra en dicha zona. Algunos criterios en los que se suelen basar los expertos para definir la fiabilidad adecuada:

- Ignorar o asignar fiabilidad 0.0 a cualquier objeto del entorno que presente en determinadas situaciones movimiento. Como por ejemplo los árboles que se pueden mover cuando hace viento, las partes que aparezcan del cielo que debido a los movimientos de las nubes provocarían movimientos, las fachadas de los edificios que tengan ventanas. Las ventanas son objetos que deben de obviarse porque reflejan la luz dependiendo de la hora del día y a los resultados.

- Las zonas que no se pretendan monitorizar asignar fiabilidad 0.0 con el fin de ignorarlas. Por ejemplo se está monitorizando el recinto de una urbanización y según la vista de la fuente también es captado parte del patio del vecino, el cuál no se pretende monitorizar.

- Las zonas en las que se produzcan oclusiones parciales de los objetos deben de indicarse como zonas de baja fiabilidad. Ejemplos de estas zonas se dan cuando los objetos a detectar pasan por debajo de farolas, árboles, entre vehículos, etc; provocando que éstos sean vistos parcialmente o divididos en partes por esas oclusiones.

- Las zonas lejanas o zonas donde la visibilidad disminuye deben de indicarse como zonas de baja fiabilidad. Ejemplos de estas zonas se dan cuando los objetos pasan por zonas donde la iluminación es menor, haciendo que los píxeles que lo definen se fusionen con el entorno debido a esa cierta oscuridad.

- Dependiendo de la perspectiva y vista de la cámara indicar la fiabilidad de las zonas. Este criterio es importante, puesto que aparte de detectar los movimientos se obtienen también las dimensiones del objeto. Si la vista de una fuente es una vista de pájaro pues el cálculo de la altura de los objetos detectados será incorrecto. Luego se deben

asignar las fiabilidades teniendo en cuenta cómo de correcta va a ser la información que dará la perspectiva de la fuente en determinadas zonas.

Teniendo en cuenta la forma en que se describe la fiabilidad de las áreas, el diseño que se ha utilizado para la estructura del archivo XML es el siguiente:

```
<Reliability_0.0>
    <!--Definition of the pixels that belong to the area with reliability 0.0.-->
  <Pixel>
    <X>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </X>
    <Y>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </Y>
  </Pixel>
    ...
    ...
</Reliability_0.0>
<Reliability_0.1>
    <!--Definition of the pixels that belong to the area with reliability 0.1.-->
  <Pixel>
    <X>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </X>
    <Y>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </Y>
  </Pixel>
    ...
    ...
</Reliability_0.1>
<Reliability_0.2>
    <!--Definition of the pixels that belong to the area with reliability 0.2.-->
  <Pixel>
    <X>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </X>
    <Y>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </Y>
  </Pixel>
    ...
    ...
</Reliability_0.2>
....
....
....
<Reliability_0.8>
    <!--Definition of the pixels that belong to the area with reliability 0.8.-->
  <Pixel>
    <X>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </X>
    <Y>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </Y>
  </Pixel>
    ...
    ...
</Reliability_0.8>
<Reliability_0.9>
    <!--Definition of the pixels that belong to the area with reliability 0.9.-->
  <Pixel>
    <X>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </X>
    <Y>
      <!--Posicion del pixel sobre el eje horizontal. Debe de ser un entero y positivo.-->
    </Y>
  </Pixel>
```

```
66      ...
67      ...
68  </Reliability_0.9>
```

LISTADO 5.4: Formato del archivo XML de áreas fiables

Como se puede apreciar la zona con fiabilidad 1 no aparece porque se sobreentiende que la fiabilidad de la vista a priori es 1 y es sobre la vista cuando se definen el resto zonas que no tengan fiabilidad 1. En cuanto a los mecanismos de detección de error son los mismos que para las zonas de entrada/salida.

5.1.3. Submódulo de eventos

Este submódulo se encarga de tratar los eventos que el usuario va realizando sobre la interfaz gráfica de la aplicación. El submódulo de eventos ha sido diseñado para que permita percibir eventos sobre la posición del puntero del ratón y detectar *el movimiento del ratón* ante determinadas zonas de la interfaz gráfica. Y por otro lado también percibe *los botones del ratón* que el usuario ha pulsado. La información sobre los eventos del movimiento y los botones del ratón es comunicada al módulo de procesamiento para que realice las acciones correspondientes al proceso actual.

5.2. Módulo de procesamiento

Módulo principal que recoge la información del módulo de entrada para procesarla en cada uno de los procesos. Éste se utiliza en el módulo de gestión de agentes para tener un control y una coordinación entre los agentes que se encargan de los procesos.

Una vez que se ha indicado el número de fuentes y sus tipos en el módulo de entrada, se pasa al submódulo de despliegue para realizar el despliegue del sistema de tracking multi-cámara. En este despliegue intervienen agentes para el proceso de calibrado, posicionamiento y selección de áreas de I/O y fiables. Tras realizar este despliegue, comienza un agente encargado del proceso de aprendizaje donde aprende a diferenciar lo que forma parte del fondo y lo que no, para que posteriormente se puedan desempeñar los procesos de *tracking* que conllevan a la segmentación, identificación de los objetos, clasificación en un tipo, estimación de las posiciones de los objetos y fusión con la información averiguada por otros agentes.

Las características de estos submódulos los hacen complejos y es importante que antes de ser probados con un entorno real estén previamente depurados. De esta depuración se

encarga el módulo de depuración. A continuación se procede con la descripción de los detalles de cada uno de estos submódulos.

5.2.1. Submódulo de despliegue

El submódulo de despliegue contiene la descripción e implementación de los procesos que se tienen que realizar para desplegar el sistema y poder realizar posteriormente el *tracking* multi-cámara.

Calibrado

En este apartado se describe en qué consiste el proceso de calibrado. El objetivo del proceso de calibrado es obtener la información sobre los coeficientes de distorsión de la lente y la matriz de la cámara (ver sección 3.5.1). Como bien se especificó en esa sección, para calibrar la fuente de *stream* se necesita hacer uso de un patrón.

Por lo tanto la primera decisión de diseño fue la elección de un patrón. Tras realizar una serie de investigaciones se descubrió que la biblioteca *OpenCV* trabaja bastante bien con el uso de *tableros de ajedrez* como patrón para calibrar las fuentes. Luego el proceso de calibrado está diseñado e implementado para calibrar las fuentes haciendo uso de *tableros de ajedrez* como patrón.

Una vez elegido el patrón a utilizar, se preguntará: Cómo se obtienen los coeficientes de distorsión y la matriz de la cámara. El funcionamiento consiste en realizar una comparación con los puntos reales representados en el tablero y los puntos correspondientes del tablero vistos a través de la fuente *stream*. Para un tablero 8×5 sus esquinas interiores se pueden ver de color rojo en la figura 5.3 y resultan ser $7 \times 4 = 28$ *esquinas*. Las coordenadas reales de estas esquinas se pueden calcular sabiendo lo que mide el lado del subcuadrado.

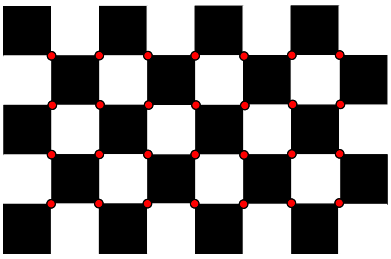

FIGURA 5.3: Esquinas internas de un tablero de ajedrez.

Conociendo las coordenadas reales del tablero, solamente se necesita algún algoritmo para detectar dichas esquinas interiores en la vista de la fuente *stream*. *OpenCV* ofrece

una función llamada *cvFindChessboardCorners* que se encarga de detectar el número de esquinas indicadas sobre la imagen especificada. A estas esquinas detectadas en la imagen es conveniente pasarle un proceso que mejore la precisión en esa detección. Este proceso de mejora de la precisión consiste en hacer una subdivisión del pixel, para ello se utiliza la función *cvFindCornerSubPix*. Los resultados de aplicar estas funciones sobre una fuente de *stream* se pueden ver en la figura 5.4.

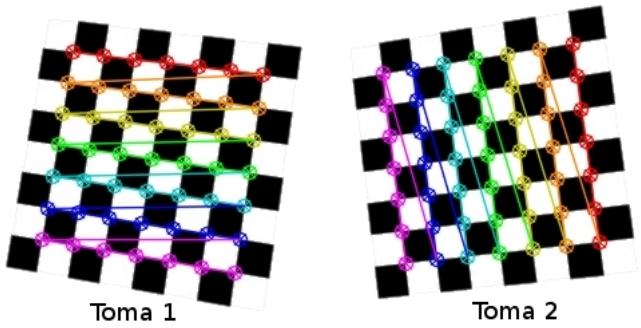

FIGURA 5.4: Esquinas internas detectadas por OpenCV.

Resumiendo, el proceso de calibrado consiste en comparar las coordenadas reales con las detectadas, pero se necesitan varias imágenes sobre las que comparar. *OpenCV* recomienda que se tomen entre doce y veinte muestras del tablero de ajedrez para obtener buenos valores en el proceso de calibración. A la hora de utilizar un conjunto de tomas, se puede dar el caso que se muestra en la figura 5.4 en el que para tableros cuadrados *OpenCV* varía el orden de detección de las esquinas del tablero. Esto supone un problema, puesto que si se cambia el orden de detección de las esquinas se está cometiendo error en el proceso de calibrado. Luego para solventar este problema, la aplicación está diseñada para que visualice lo que *OpenCV* ha detectado como esquinas y es el propio usuario quien decide si dar por buena esa toma o no. De esta forma la aplicación realizará un correcto proceso de calibración tanto para tableros regulares como irregulares.

Una vez que se dispone del número de tomas correctas del tablero de ajedrez, el proceso de calibración obtiene los coeficientes de distorsión y la matriz de calibración haciendo uso de la función *cvCalibrateCamera2*. El algoritmo 1 representa en pseudocódigo el proceso de calibrado y en la figura 5.5 se puede ver el diagrama de interacción del proceso de calibrado.

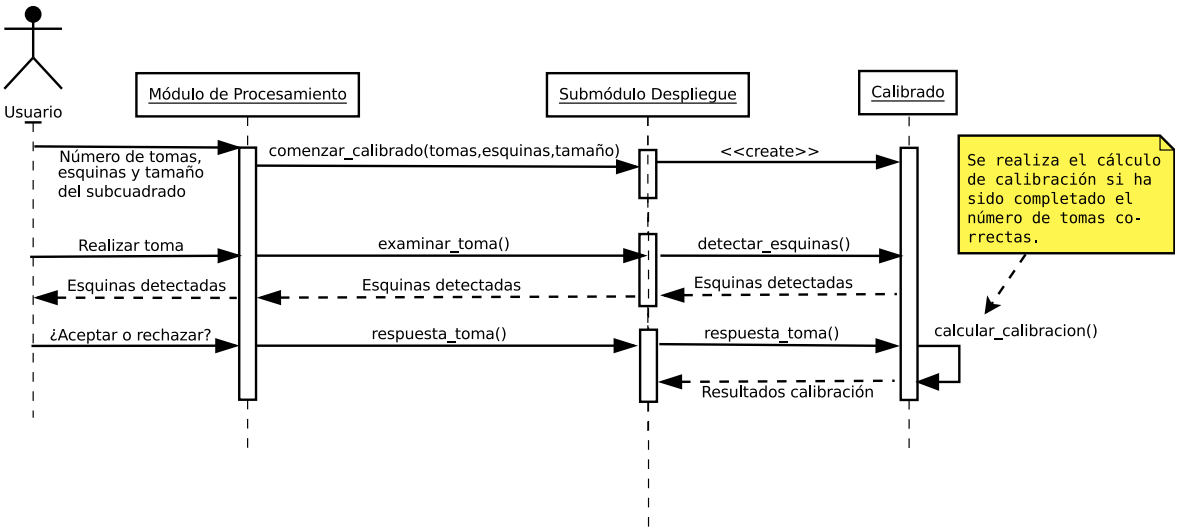

FIGURA 5.5: Diagrama de interacción del proceso de calibrado.

Algoritmo 1 Algoritmo de calibración de una fuente *stream*

Entrada: $Total\_tomas$: Número de tomas correctas para calibrar la fuente.
$Esquinas_h$: Número de esquinas internas para detectar sobre el eje horizontal.
$Esquinas_v$: Número de esquinas internas para detectar sobre el eje vertical.
$Lado$: El tamaño del lado del subcuadrado.
$Toma$: La toma que el usuario quiere que se analice en ese momento.
Salida: Los parámetros intrínsecos de la fuente *stream*.
Los coeficientes de distorsión de la fuente *stream*.

1: **mientras** $Tomas\_analizadas \neq Total\_tomas$ **hacer**
2: Analizar $Toma$ para encontrar las esquinas. Haciendo uso de las funciones *cvFindChessboardCorners* y *cvFindCornerSubPix*.
3: **si** $Esquinas\_encontradas \neq Esquinas_h \times Esquinas_v$ **entonces**
4: $Toma$ incorrecta.
5: **si no**
6: Mostrar esquinas encontradas y esperar *respuesta* del usuario.
7: **si** $respuesta =$ **cierto entonces**
8: Incrementar $Tomas\_analizadas$.
9: **fin si**
10: **fin si**
11: Recibir nueva toma para analizarla.
12: **fin mientras**
13: Calcular la distorsión y parámetros intrínsecos con *cvCalibrateCamera2*.
14: **devolver** parámetros intrínsecos y distorsión de la fuente.

Posicionamiento

Otro de los procesos que se tienen que realizar en el despliegue de un sistema de monitorización multicámara es el posicionamiento de las cámaras. Puesto que es un sistema multicámara en el que se debe contrastar la información de las fuentes como un único conjunto, es decir, lo captado por cada fuente debe estar expresado en un sistema de referencia global al de todas las fuentes. Para conseguir esto se aplica un proceso de posicionamiento con el objetivo de conocer la posición y orientación real de cada fuente calibrada con respecto a un patrón captado por todas las demás fuentes. Dicha patrón representa el centro de coordenadas de ese sistema de referencia global. La forma de obtener dicha información es mediante un proceso similar al de calibrado debido a que éste también realiza una comparación entre las coordenadas reales y visuales del patrón en la propia captura de la fuente. Luego lo primero de todo fue decidir cuál iba a ser el patrón. Sobre este aspecto se decidió pensar en utilizar un *cuadrilátero*. La ventaja de elegir un cuadrilátero como patrón está en que la mayoría de los entornos presentan objetos estáticos con forma de cuadrilátero y éstos se podrían utilizar como dicho patrón; por ejemplo las líneas que definen un aparcamiento de un vehículo, o pasos de cebra, etc. A este patrón para diferenciarlo del tablero de ajedrez se le suele llamar *marca*. En la figura 5.6 se puede ver un ejemplo.

FIGURA 5.6: Ejemplo de posicionamiento de una cámara a partir de una marca.

Por lo tanto el posicionamiento consiste en indicar las coordenadas reales de dicha marca y compararlas con las coordenadas virtuales detectadas en la imagen de la fuente *stream*. Para detectar dichas coordenadas no se podía utilizar la función *cvFindChessboardCorners* de *OpenCV* debido a que si el entorno presenta una gran cantidad de cuadriláteros dicha función podría indicar coordenadas que no fuesen las indicadas. Para solucionar este problema se decidió en que fuese el usuario, él que especificase aquellas coordenadas virtuales. De esta forma se consigue que el proceso de posicionamiento sea aplicable para cualquier entorno que presente cuadriláteros.

Es muy importante remarcar que para que el proceso de posicionamiento tenga éxito, el orden en que se van especificando las coordenadas virtuales debe ser el mismo que el utilizado en la especificación de las coordenadas reales.

Una vez que se tiene la información de las coordenadas reales y virtuales, y la información resultante del proceso de calibrado, *OpenCV* ofrece la función *cvFindExtrinsicCameraParams2* que permite obtener la posición y orientación de la marca con respecto a un centro de coordenadas fijado en el centro de la fuente. Es decir, lo que se obtiene con *cvFindExtrinsicCameraParams2* no es la posición y orientación de la fuente sino de la propia marca, pero dicha posición y orientación no es la que se le proporciona como información de entrada porque su centro de coordenadas es distinto.

En la figura 5.7 se representa lo que *cvFindExtrinsicCameraParams2* recibe como entrada y lo que genera como salida. Se puede ver que en realidad dicha función sí calcula la posición y orientación de la fuente para así situar el centro de coordenadas y expresar los resultados en base a este centro de coordenadas.

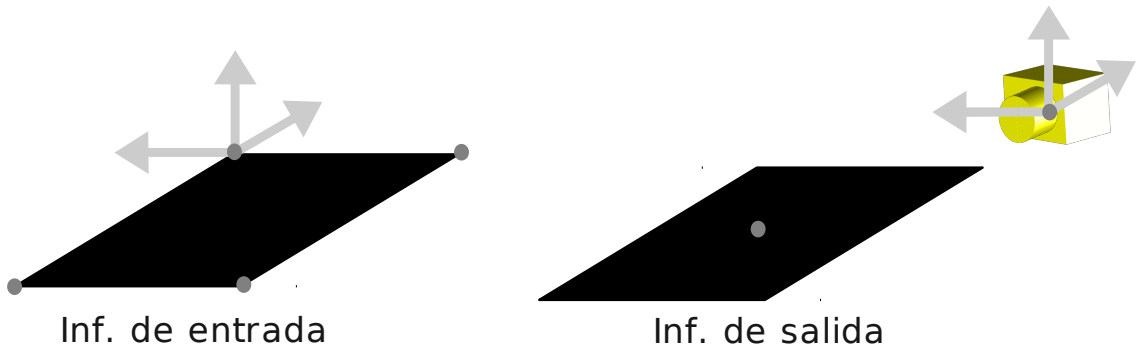

FIGURA 5.7: Resultados obtenidos por *cvFindExtrinsicCameraParams2*.

Explicada la información que se obtiene de la función *cvFindExtrinsicCameraParams2* se puede dar cuenta que dicha información debe ser transformada. La información obtenida directamente de *cvFindExtrinsicCameraParams2* no sirve porque lo que se pretende con el posicionamiento es obtener las posiciones y orientaciones de las fuentes con respecto a la marca de referencia. Es decir, la marca de referencia representa el centro de coordenadas del mundo a monitorizar y las posiciones de las cámaras y orientaciones se deben expresar con respecto a la marca (centro de coordenadas) si no se hace de esta forma el entorno visto por cada fuente sería totalmente propio para dicha fuente.

Para solucionar este problema basta con realizar la inversa de la matriz de transformación obtenida por la función *cvFindExtrinsicCameraParams2*. Con esto lo que se consigue

es cambiar el centro de coordenadas hacia la propia marca y obtener la posición y orientación de la fuente con respecto al centro de coordenadas de la marca.

Toda la explicación del proceso de posicionamiento es formalizada con los algoritmos 2 y 3.

Algoritmo 2 Algoritmo de posicionamiento de una fuente *stream*

Entrada: *C_reales* : Las coordenadas 3D que definen la marca.
C_virtuales : Las coordenadas 2D que definen la marca en la captura de la fuente.
Salida: La matriz de transformación que representa la posición y la orientación de la fuente con respecto a la marca.

1: Obtener la posición y el vector de rotación mediante la función *cvFindExtrinsicCameraParams2*.
2: Analizar los resultados de la posición y el vector de rotación mediante la función *cvProjectPoints2*.
3: Mostrar los resultados del análisis.
4: Convertir el vector de rotación en matriz de rotación mediante la función *cvProjectPoints2*.
5: Construir una matriz 4×4 que almacene la rotación y posición con **el algoritmo 3**.
6: Realizar la inversa de la matriz del anterior paso para obtener la matriz de transformación.
7: **devolver** Matriz de transformación.

Para realizar la construcción de una matriz 4×4 que almacene la rotación y posición obtenidas mediante la función *cvFindExtrinsicCameraParams2* se utiliza el algoritmo 3.

Algoritmo 3 Algoritmo de construcción de la matriz 4×4 que almacene la rotación y posición

Entrada: *Posicion* : Es un vector 3×1 que contiene la posición de la cámara.
Rotaciones : Matriz rotación 3×3 que contiene las rotaciones de la cámara.
Salida: $Matriz$: La matriz 4×4 que representa la posición y la orientación.

1: **para** $fila = 1$ hasta 3 **hacer**
2: **para** $columna = 1$ hasta 3 **hacer**
3: $Matriz[fila][columna] = Rotaciones[columna][fila]$
4: **fin para**
5: $Matriz[fila][4] = 0.0$
6: **fin para**
7: **para** $columna = 1$ hasta 3 **hacer**
8: $Matriz[4][columna] = Posicion[columna][1]$
9: **fin para**
10: $Matriz[4][4] = 1.0$
11: **devolver** $Matriz$.

El diagrama de interacción del proceso de posicionamiento se puede ver en la figura 5.8.

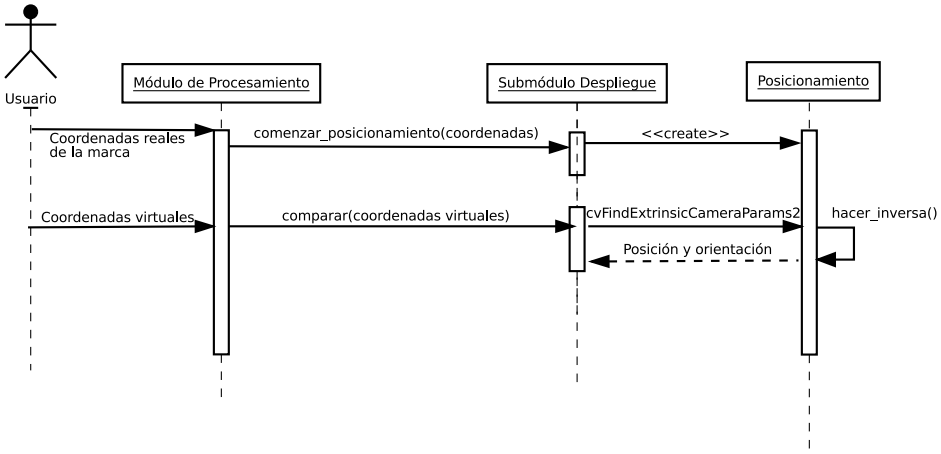

FIGURA 5.8: Diagrama de interacción del proceso de posicionamiento.

Definición de áreas I/O

En este apartado se describe el proceso sobre la definición de áreas de entrada y salida. La definición de estas zonas de entrada y salida permite a la aplicación conocer cuando los objetos pueden aparecer o desaparecer de la escena. Por lo tanto esta información es muy útil para tenerla en cuenta en los algoritmos de seguimiento.

Se decidió que la aplicación proporcionase una forma cómoda de definir áreas. Un área es creada a partir de la lista de puntos que definen su contorno. Para poder definir áreas se pensó en diseñar e implementar una solución similar a cuando se pinta sobre papel. Primero se coge el lápiz y se comienza a trazar el contorno del área hasta que finalmente se despega el lápiz del papel y en ese momento ya se tiene dibujado el contorno. De esta técnica se puede dar el caso de que los extremos del contorno no se encuentren unidos, si es así simplemente se unen entre ellos y queda perfectamente definido el contorno del área como se puede ver en la figura 5.9.

Implementando esta forma de definir áreas se consigue una buena flexibilidad para dibujar áreas con cualquier forma. También para dar una mayor flexibilidad en el proceso de definición de áreas, la aplicación ha sido diseñada para que el usuario pueda remover áreas ya creadas y seleccionarlas. Cuando el usuario ha definido el contorno debe de crear el área especificando el tipo:

- Zona de entrada: Se ha decidido representar las zonas de entradas de color verde.
- Zona de salida: Se ha decidido representar las zonas de salida de color rojo.
- Zona de entrada y salida: Se ha decidido representar las zonas de entrada y salida de color azul.

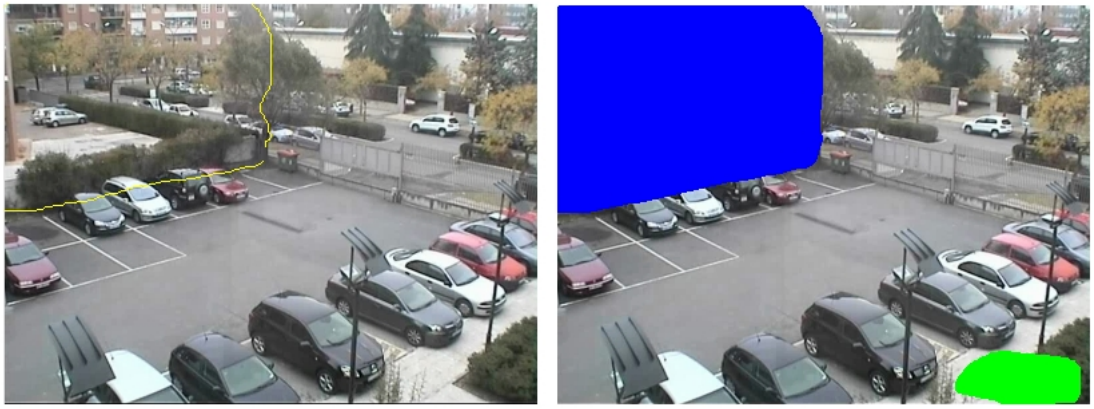

FIGURA 5.9: Herramienta para la definición de áreas sobre la vista de la fuente."Ver en Anexo B imagen a color"

Una vez que el usuario ha definido las áreas de entrada y salida se procede a crear la máscara. Esta máscara debe estar diseñada para que las operaciones de detección de si un objeto pertenece o no a un área sean eficientes. Para ello se pensó en crear una máscara con las mismas dimensiones que la imagen sobre la que se han definido las áreas, inicializarla toda a ceros y aquellos píxeles que pertenezcan a zona de entrada representarlos con un *uno*, los que pertenezcan a zona de salida con un *dos* y los que pertenezcan a entrada y salida con un *tres*.

Teniendo la imagen en negro con las zonas de entrada, de salida y de entrada/salida dibujadas en verde, rojo y azul. Para crear la máscara se hace uso del algoritmo 4 donde dicha imagen es separada en los tres canales *RGB*.

La forma de realizar las operaciones de lectura de esta máscara se describen en las secciones posteriores.

Definición de áreas fiables

En cuanto a la definición de áreas fiables, este proceso reutiliza toda la parte del proceso de creación de áreas descrito en el anterior apartado; y las únicas diferencias son la utilización de los colores para representar los tipos de áreas y la construcción de la máscara. Para representar la diferencia entre las áreas se utiliza una gama de colores grises siendo el color gris oscuro para las áreas más fiables y el gris claro para las áreas menos fiables.

A la hora de construir la máscara se crea con las dimensiones de la imagen y se inicializa con todo dieces (representando que todo es 100 % fiable). Para aquellos píxeles que pertenezcan a zona 0.0 fiables se representan con un *cero*, los que pertenezcan a zona

Algoritmo 4 Algoritmo de construcción de la máscara de entrada y salida

Entrada: *Imagen* : Es la imagen en negro con las regiones de entrada, salida y entrada/salida.
Salida: *Mascara* : La matriz que representa la máscara de áreas de I/O.

1: **para** $fila = 1$ hasta $alto\_Imagen$ **hacer**
2: **para** $columna = 1$ hasta $ancho\_Imagen$ **hacer**
3: $Mascara[fila][columna] = 0$
4: **fin para**
5: **fin para**
6: Separar *Imagen* en tres canales *red*, *green* y *blue*.
7: **para** $fila = 1$ hasta $alto\_Imagen$ **hacer**
8: **para** $columna = 1$ hasta $ancho\_Imagen$ **hacer**
9: $Mascara[fila][columna] = (red[fila][columna] >> 7) * 2 + (green[fila][columna] >> 7) * 1 + (blue[fila][columna] >> 7) * 3$
10: **fin para**
11: **fin para**
12: **devolver** *Mascara*.

0.1 fiable con un *uno*, los que pertenezcan a zona 0.2 fiable con un *dos*, los que pertenezcan a zona 0.3 fiable con un *tres*, etc.

A diferencia de la construcción de la máscara de áreas I/O, la imagen en negro con las zonas de fiabilidad resaltadas en escala de grises no es separada en los tres canales si no que se compara el valor del pixel para conocer cuál es su valor de fiabilidad. El algoritmo 5 es utilizado para la construcción.

Algoritmo 5 Algoritmo de construcción de la máscara de zonas fiables

Entrada: *Imagen* : Es la imagen en negro con las regiones fiables en escala de grises.
Salida: *Mascara* : La matriz que representa la máscara de las regiones fiables.

1: **para** $fila = 1$ hasta $alto\_Imagen$ **hacer**
2: **para** $columna = 1$ hasta $ancho\_Imagen$ **hacer**
3: $Mascara[fila][columna] = 10$
4: **fin para**
5: **fin para**
6: **para** $fila = 1$ hasta $alto\_Imagen$ **hacer**
7: **para** $columna = 1$ hasta $ancho\_Imagen$ **hacer**
8: Obtener en $valor$ la fiabilidad para el pixel $Imagen[fila][columna]$.
9: $Mascara[fila][columna] = valor \times 10$
10: **fin para**
11: **fin para**
12: **devolver** *Mascara*.

5.2.2. Submódulo de aprendizaje

Este submódulo está relacionado con el proceso de segmentación del submódulo de tracking. Y será utilizado por un agente para llevar a cabo el proceso de aprendizaje. En el capítulo de antecedentes (sección 3.5.2) se describieron algunas de las técnicas de segmentación. A la hora de decantarse por un algoritmo de segmentación, se decidió utilizar un algoritmo de tipo *background*. Este algoritmo trata de separar el fondo de lo que forma parte del primer plano. Pero para separar el fondo necesita primero una fase de aprendizaje para aprender lo que es fondo.

Como en los anteriores procesos se ha hecho uso de la biblioteca *OpenCV* se investigó si ésta ofrecía soporte para llevar a cabo la fase de aprendizaje. *OpenCV* ofrece un algoritmo de background denominado *codebook*.

La forma de aprender lo que es fondo consiste en almacenar los valores de colores que ha tenido cada pixel durante un intervalo de tiempo. *Codebook* no trabaja con colores en formato *RGB* sino en formato *YUV*. Los motivos por los que trabaja con colores en formato *YUV* se debe a que según los estudios, las variaciones en el fondo se realizan sobre el nivel de brillo en lugar de a nivel de color, y por lo tanto el formato *YUV* permite trabajar con los colores teniendo en cuenta su brillo.

Para detectar esas variaciones del color de cada pixel se utilizan cajas. Las cajas van agrupando los colores cercanos que ha tenido ese pixel. De tal forma que dado un color nuevo se mira el historial de cajas que tiene ese pixel y si de este historial hay alguna caja que lo pueda englobar, es decir, los colores pertenecientes a esa caja son muy parecidos al nuevo pues se actualiza dicha caja con el nuevo color. Si no existe ninguna caja próxima a dicho color, se crea una nueva caja para ese pixel que contenga ese nuevo color.

El algoritmo que analiza si una caja contiene colores parecidos a un nuevo color, utiliza dos umbrales (uno máximo y otro mínimo). De tal forma que el color es asociado a la caja si los valores para cada uno de sus canales se encuentran dentro del intervalo:

$$[umbral\_minimo - valor\_minimo\_caja,\ umbral\_maximo + valor\_maximo\_caja]$$

Por último indicar de este algoritmo de aprendizaje que la memoria consumida por este proceso es alta. Para evitar que el consumo se dispare, *codebook* permite almacenar en cada caja cuándo ésta fue actualizada. De esta forma a medida que va pasando el tiempo, se pueden eliminar aquellas cajas que fueron creadas en su momento pero que no han sido actualizadas desde hace mucho tiempo.

Todo lo explicado anteriormente corresponde al algoritmo *codebook* de *OpenCV* y se puede encontrar su código fuente en la página: `https://code.ros.org/trac/opencv/browser/trunk/opencv/src/cvaux/cvbgfg_codebook.cpp?rev=1334`.

El algoritmo que rige el comportamiento del submódulo de aprendizaje y hace uso de *codebook* es el que se describe a continuación:

Algoritmo 6 Algoritmo del submódulo de aprendizaje

Entrada: *Imagen* : Es la imagen que ha captado la fuente y se utilizará para aprender.
Salida: **cierto** Si la fase de aprendizaje continúa y **falso** si ha finalizado la fase de aprendizaje.

1: **si** $contador\_frames \neq n\_frames\_aprendizaje$ **entonces**
2: Convertir *Imagen* en formato YUV mediante el uso de la función *cvCvtColor*.
3: Actualizar el modelo de *codebook* con la imagen en formato YUV mediante el uso de *cvBGCodeBookUpdate*.
4: Incrementar $contador\_frames$.
5: **devolver cierto** .
6: **si no**
7: Limpiar las cajas que fueron antiguamente utilizadas mediante la función *cvBGCodeBookClearStale*.
8: Restablecer el $contador\_frames$ a 1.
9: **devolver falso** .
10: **fin si**

5.2.3. Submódulo de tracking

El submódulo de tracking ha sido diseñado para encargarse del proceso de segmentación e identificación de las zonas donde se encuentran los objetos detectados. Este submódulo es el encargado de realizar el seguimiento de los objetos aplicando las máscaras de entrada/salida y utilizando el submódulo del raytracer para poder averiguar las posiciones de los objetos detectados. Se comunica con el submódulo de clasificación para que se realice la clasificación de los objetos que se han detectado. Y una vez que se ha completado la información de clasificación, posiciones, identificación de la zona, dimensiones del objeto se establece comunicación con el agente que se encarga de realizar la fusión de la información utilizando el submódulo de fusión.

Segmentación

Una vez que el agente ha aprendido el fondo mediante el submódulo de aprendizaje, comienza el proceso de segmentación. El proceso de segmentación consiste en tres pasos:

1. **Obtener diferencias**. En este paso se obtienen las diferencias comparando el frame actual con la información del aprendizaje que tiene el agente. Para realizar estas diferencias se hace uso de la función *cvBGCodeBookDiff* la cual genera una imagen binaria. En esta imagen binaria los píxeles blancos representan los movimientos que se han detectado y los negros aquello que pertenece al fondo. En la figura 5.10 se puede ver el resultado de aplicar este primer paso.

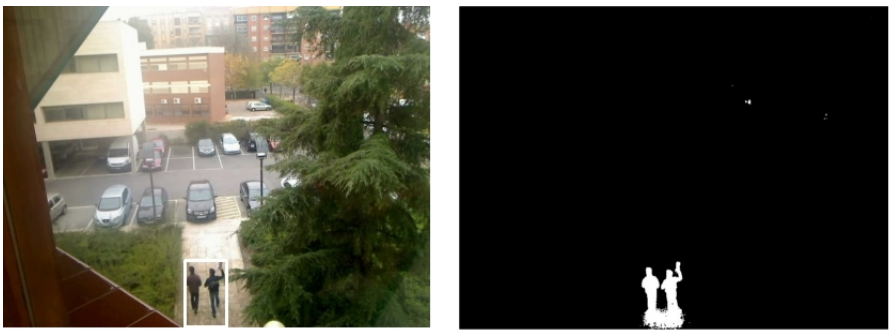

FIGURA 5.10: Resultados del primer paso del proceso de segmentación.

2. **Eliminar ruido**. Teniendo esa imagen binaria del paso anterior, se debe eliminar ruido de ella haciendo que los píxeles aislados sean eliminados. Hay que tener cuidado con este paso puesto que se pierde precisión y si el umbral de lo que se quiere eliminar no está bien ajustado, se pueden eliminar detecciones que no son ruido. Para realizar este paso se ha utilizado la función *cvSegmentFGMask* y en la figura 5.11 se pueden ver los resultados obtenidos. La imagen de la izquierda representa una salida del paso anterior, donde se pueden apreciar píxeles en blanco aislados. Tras aplicar la eliminación del ruido se obtiene la imagen de la derecha, donde se puede ver que el ruido que existía ha desaparecido y se ha reforzado mucho más la silueta de los movimientos detectados.

3. **Conexión mediante rectángulos**. Una vez que se tiene la imagen binaria limpia de ruido es utilizada para envolver con rectángulos los movimientos detectados. Los motivos por los que envolver los movimientos en rectángulos son porque la forma del rectángulo permite envolver cualquier silueta de movimiento y porque su definición es sencilla (basta con conocer la esquina superior derecha y la esquina inferior izquierda). Para envolver mediante cajas los contornos blancos de la imagen binaria

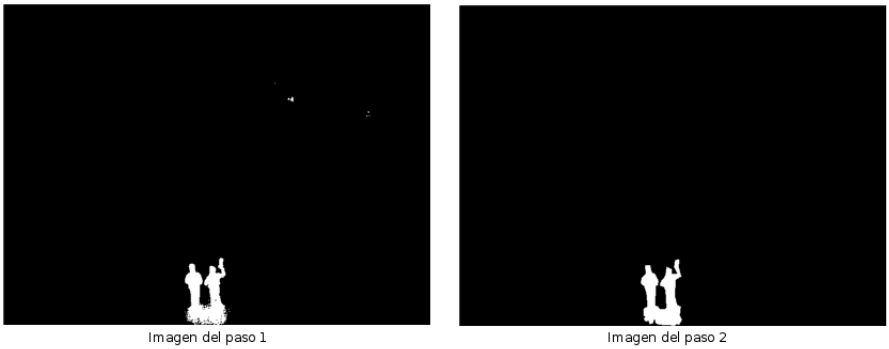

FIGURA 5.11: Resultados del segundo paso del proceso de segmentación.

se ha utilizado la función *cvFindContours*. Esta función devuelve una secuencia de los rectángulos que se han encontrado. Pero el uso de esta función a veces introduce ciertos errores a la hora de conectar en rectángulos. Es por ello que se ha diseñado un algoritmo complementario de conexión que ayuda a corregir estos errores. Para explicar el funcionamiento de este algoritmo se va a mostrar el error cometido por la función *cvFindContours* y el resultado de la corrección por dicho algoritmo. En la figura 5.12 se puede apreciar que el algoritmo complementario debe unir los dos rectángulos en uno. Para ello éste simplemente busca los rectángulos que prácticamente se encuentran muy cercanos en el eje Y. Si existen dos rectángulos que están muy próximos sobre el eje Y se considera que deben unirse.

FIGURA 5.12: Error en la conexión de rectángulos.

Los algoritmos 7 y 8 son los encargados de formalizar lo descrito en estos tres pasos.

Algoritmo 7 Algoritmo para la segmentación

Entrada: *Imagen* : Es la imagen que ha captado la fuente.
Salida: *Rectangulos* : Los rectángulos de los movimientos que se han detectado.

si $contador\_frames = n\_frames\_limpieza$ **entonces**
    Limpiar las cajas que fueron antiguamente utilizadas mediante la función *cvBGCodeBookClearStale*.
    Restablecer el $contador\_frames$ a 1.
fin si
Convertir *Imagen* a formato YUV.
Obtener diferencias con el fondo aprendido mediante la función *cvBGCodeBookDiff*.
Eliminar el ruido mediante la función *cvSegmentFGMask*.
Obtener rectángulos envolventes de los movimientos utilizando *cvFindContours*.
Eliminar rectángulos que tengan fiabilidad 0 haciendo uso del algoritmo 9.
Conectar los rectángulos mediante el algoritmo 8.
devolver *Rectangulos*.

Algoritmo 8 Algoritmo para conectar rectángulos

Entrada: *Rectangulos* : Es la secuencia de rectángulos que ha detectado la función *cvFindContours*.
Salida: *Nuevos_rectangulos* : Los nuevos rectángulos resultandes de la conexión de algunos de ellos.

para todo *rec* de la secuencia *Rectangulos* **hacer**
    para todo *rec*1 de la secuencia *Rectangulos* **hacer**
        si $rec \neq rec1$ Y **entonces**
            si $(rec1.y \geq rec.y$ Y $rec1.y \leq rec.y + umbral)$ O $(rec1.y + rec1.alto \geq rec.y - umbral$ Y $rec1.y + rec1.alto \leq rec.y + rec.alto)$ **entonces**
                *rec*1 es candidato para *rec*
            fin si
        fin si
    fin para
fin para
para los candidatos obtenidos **hacer**
    Se crea un nuevo rectángulo con dimensiones que englobe a los candidatos. Para ello se buscan coordenadas mínimas y máximas en los candidatos.
fin para
devolver *Nuevos_rectangulos*.

Identificación de zonas

En este apartado se describe el algoritmo que permite detectar si el rectángulo que contiene el movimiento pertenece a una zona de entrada/salida y la fiabilidad de la zona. Para ello requiere la información de las máscaras de I/O y fiabilidad. Puesto que se va a ejecutar muchas veces debe ser eficiente, por lo que se ha pensado en leer la información de las dos máscaras al mismo tiempo y simplemente leer solo la porción de píxeles que encierra el rectángulo y no toda la máscara.

La forma de determinar la fiabilidad y la zona a la que pertenece el rectángulo se realiza mediante el uso de contadores para cada uno de los tipos de zonas y fiabilidades. El funcionamiento del algoritmo consiste en recorrer cada uno de los píxeles del rectángulo de movimiento y si el pixel leído es *blanco* entonces examinar en las máscaras y actualizar contadores dependiendo del tipo I/O y fiabilidad que indiquen los píxeles de las máscaras. Tras finalizar de recorrer los píxeles del rectángulo, el rectángulo pertenece a la zona y fiabilidad que marquen los contadores máximos. Todo lo descrito anteriormente se recoge en el siguiente algoritmo:

Algoritmo 9 Algoritmo para la identificación de zonas

Entrada: *Rectangulo* : Es el trozo de imagen donde se encuentra el movimiento expresado en píxeles blancos.
Mascara_io : La máscara que representa las zonas de I/O.
Mascara_fiables : La máscara que representa las zonas fiables.
Salida: *Area* : El tipo de área de I/O a la que pertenece el movimiento.
Fiabilidad : La fiabilidad que tiene ese movimiento por pertenecer a una zona fiable.

1: $contadores\_io[4] = \{0,0,0,0\}$
2: $contadores\_fiable[11] = \{0,0,0,0,0,0,0,0,0,0,0\}$
3: **para** $fila = 1$ hasta *alto_Rectangulo* **hacer**
4: **para** $columna = 1$ hasta *ancho_Rectangulo* **hacer**
5: **si** $Rectangulo[fila][columna] = Blanco$ **entonces**
6: Incrementar $contadores\_io[Mascara\_io[fila][columna]]$.
7: Incrementar $contadores\_fiable[Mascara\_fiables[fila][columna]]$.
8: **fin si**
9: **fin para**
10: **fin para**
11: $Area =$ Indice del máximo de *contadores_io*.
12: $Fiabilidad =$ (Indice del máximo de $contadores\_fiable$)$\times 0.1$.
13: **devolver** *Area* y $Fiabilidad$.

5.2.4. Submódulo de raytracer

El submódulo de raytracer es utilizado por el submódulo de *tracking* para averiguar la posición del objeto detectado en el entorno monitorizado. Este submódulo implementa un *raytracer*, el cual ha sido diseñado siguiendo como referencia la información presentada en el capítulo de Antecedentes (sección 3.4) donde se detalla su funcionamiento y las partes que lo forman (el mundo 3D, las cámaras virtuales y los rayos). A diferencia de los submódulos del módulo de procesamiento descritos hasta ahora, el submódulo de *raytracer* solo es utilizado por un único agente. Es decir, mientras que los submódulos de despliegue, aprendizaje, *tracking* y clasificación son ejecutados por N agentes, siendo N el número de fuentes conectadas al sistema, el submódulo de *raytracer* solo es ejecutado por un único agente. Luego los demás agentes para utilizarlo se lo tienen que comunicar al único agente encargado de procesar el submódulo de *raytracer*.

Mundo 3D

El mundo 3D sobre el que actúa el *raytracer* se ha diseñado para que esté formado solamente por planos y triángulos. Los motivos por los que solo se permiten planos y triángulos son:

- El plano y el triángulo son figuras sencillas donde la forma de calcular la intersección con el rayo no requiere un coste computacionalmente elevado. Esto es un factor muy importante debido a que la aplicación debe de trabajar en tiempo real.
- El triángulo permite aproximar la mayoría de las formas de los objetos. Es decir, las formas que presentan los objetos pueden ser aproximadas a base de triángulos.

Acelerador del raytracer

El diseño del *raytracer* debe ser eficiente puesto que se ejecuta constantemente y si éste no está bien diseñado la aplicación no podrá trabajar en tiempo real. Según el diseño especificado en la sección 3.4 se puede ver que el *raytracer* disminuye su rendimiento a medida que el número de objetos que forman el *mundo 3D* aumenta. Esto es obvio, puesto que teniendo un rayo que parte de un píxel se tienen que evaluar cada uno de los objetos que forman el mundo 3D para ver con cual objeto va a incidir el rayo.

Para solucionar esta caída de rendimiento cuando el número de objetos que definen el mundo 3D aumenta, se ha diseñado un algoritmo de aceleración basado en la utilización

de un *grid*. Este algoritmo reduce el número de objetos a evaluar para saber con qué objeto incidió el rayo. Según la sección 3.4 los rayos inciden siempre sobre los planos, debido a que éstos son infinitos. Luego este algoritmo de aceleración basado en un grid trabaja solamente con los objetos que son triángulos.

La idea consiste en crear una caja que contenga todos los objetos del mundo 3D. A dicha caja se le denomina *Grid* porque la caja esta subdividida en celdas. Supongamos que el mundo 3D tiene cuatro triángulos y se crea un grid que los incluya. La vista de pájaro del mundo 3D quedaría como en la figura 5.13.

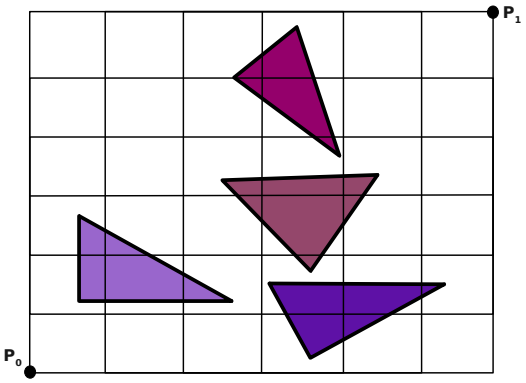

FIGURA 5.13: Representación de un grid.

Analizando la figura 5.13 se puede decir que las celdas que componen el *grid* van a almacenar la lista de objetos que pasan por dicha celda. Una vez construido el grid con su descomposición en celdas, se tiene que averiguar la lista de objetos que pertenece a cada celda que forma el *grid*. Para conseguir esto, los objetos son envueltos en cajas y se analiza las celdas que invade dicha caja. En la figura 5.14 se puede ver que el triángulo envuelto por la caja discontinua pertenece a las celdas: **c, d, i, j, ñ, o**.

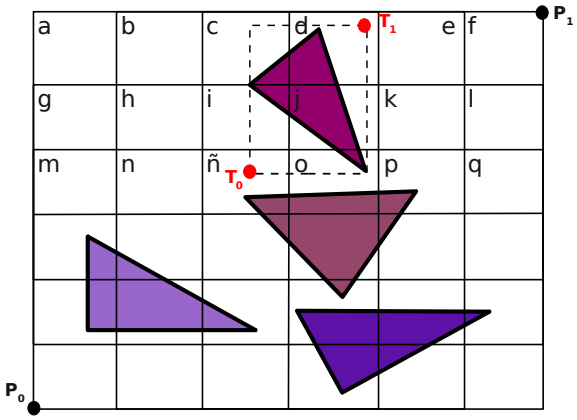

FIGURA 5.14: Asignación de los objetos a las celdas del grid.

Para construir este grid se tiene que calcular las coordenadas de $P0$ y $P1$ y el número de celdas que va a tener como se puede ver en la figura 5.13. El algoritmo 10 permite calcular las coordenadas de $P1$ quedándose siempre con la máxima, para calcular el $P0$ es el mismo algoritmo pero ajustado para que se quede siempre con la coordenada mínima.

Algoritmo 10 Algoritmo para calcular las coordenadas de $P1$

Entrada: $Objetos$: La lista de objetos que componen el mundo 3D.
Salida: $P1$: Las coordenadas del extremo superior del grid.

1: $x = -\infty$
2: $y = -\infty$
3: $z = -\infty$
4: **para todo** obj de $Objetos$ **hacer**
5: **si** $obj.caja.T_1.x > x$ **entonces**
6: $x = obj.caja.T_1.x$
7: **fin si**
8: **si** $obj.caja.T_1.y > y$ **entonces**
9: $y = obj.caja.T_1.y$
10: **fin si**
11: **si** $obj.caja.T_1.z > z$ **entonces**
12: $z = obj.caja.T_1.z$
13: **fin si**
14: **fin para**
15: Incrementar en 0.1 para que no se quede justo el grid: $P1 = (x+0.1, y+0.1, z+0.1)$.
16: **devolver** $P1$.

Ahora conociendo $P0$ y $P1$ se pueden calcular las dimensiones que tiene el *grid* con las siguientes ecuaciones:

$$\begin{aligned} ancho &= P1.x - P0.x \\ alto &= P1.y - P0.y \\ grosor &= P1.z - P0.z \end{aligned} \tag{5.1}$$

Y para calcular el número de celdas que hay en cada uno de los tres ejes (x,y,z) del grid, se necesita saber el número de objetos (n) que contiene el grid y sus dimensiones de tal forma que:

$$
\begin{aligned}
size\_grid &= \left[\frac{(ancho * alto * grosor)}{n}\right]^{1/3} \\
celdas_x &= \lceil \frac{2 \times ancho}{size\_grid} \rceil + 1 \\
celdas_y &= \lceil \frac{2 \times alto}{size\_grid} \rceil + 1 \\
celdas_z &= \lceil \frac{2 \times grosor}{size\_grid} \rceil + 1
\end{aligned}
\tag{5.2}
$$

Una vez construido el grid, se ha mencionado que se deben asignar los objetos a las celdas como se mostró en la figura 5.14. Para ello los índices de las celdas se encuentran dentro de los intervalos: $[0, celdas_x - 1] \times [0, celdas_y - 1] \times [0, celdas_z - 1]$.

Según muestra la figura 5.14 para conocer el rango de índices de celdas en las que cae el objeto, se calculan los índices de celdas para los puntos T_0 y T_1 en cada uno de los ejes (x, y, z). Para averiguar los índices de celda a los que pertenece un punto T se utilizan las siguientes ecuaciones:

$$
\begin{aligned}
indice_x &= \lfloor celdas_x \times \frac{T_x - P_{0x}}{p_{1x} - P_{0x}} \rfloor \\
indice_y &= \lfloor celdas_y \times \frac{T_y - P_{0y}}{p_{1y} - P_{0y}} \rfloor \\
indice_z &= \lfloor celdas_z \times \frac{T_z - P_{0z}}{p_{1z} - P_{0z}} \rfloor
\end{aligned}
\tag{5.3}
$$

De las ecuaciones 5.3 se obtienen los índices de las celdas dentro del intervalo $[0, celdas_x] \times [0, celdas_y] \times [0, celdas_z]$ luego si los índices obtenidos valen $celda_x$, $celda_y$, $celda_z$ entonces decrementarle una unidad para que se queden dentro del intervalo correcto.

Con los objetos ya clasificados en celdas, el algoritmo de aceleración basado en grid consiste en analizar solo los objetos de las celdas por las que pasa el rayo. A la hora de trazar un rayo sobre el grid se pueden dar tres casos:

- Que el rayo no incida en el grid. Luego directamente el rayo no golpea ningún objeto del mundo 3D.
- Que el rayo sea emitido dentro del grid. Luego se comenzará a examinar los objetos pertenecientes a la celda que contengan el origen del rayo.

- Que el rayo incide en el grid, pero es emitido fuera de éste. Luego se comenzará a examinar los objetos pertenecientes a la celda con la que incidió el rayo para entrar en el grid.

En la figura 5.15 se indica un ejemplo de los objetos que serían analizados por el algoritmo. Como el rayo incide en la celda *a* ésta es la primera en ser analizada. Pero como no tiene ningún objeto se analiza la siguiente celda que atraviesa que es la *b*. En la celda *b* se ve que el rayo no incide en el triángulo luego se pasa a analizar la celda *c* y vuelve a pasar lo mismo que tampoco incide con ese triángulo. Por último analiza la celda *d* y aquí el rayo incide con el triángulo luego finaliza el raytracer para ese rayo. Para este ejemplo se han analizado tres triángulos de cuatro, luego se ha ganado rendimiento.

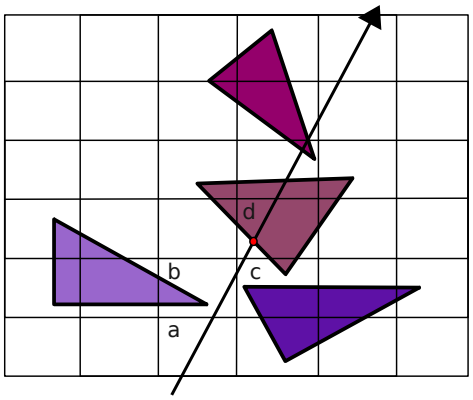

FIGURA 5.15: Ejemplo del funcionamiento del acelerador basado en grid.

5.2.5. Submódulo de clasificación

El submódulo de clasificación se encarga de realizar una catalogación a priori del movimiento detectado. Este submódulo se comunica con el submódulo de *tracking* para que le indique la información del rectángulo de movimiento que ha sido detectado. Se encarga de clasificar el movimiento en el rol *persona* o en el rol de *vehículo*. Se ha decidido realizar la distinción entre sólo estos dos tipos debido a que en la mayoría de los sistemas de vigilancia el comportamiento correcto para coches, camiones y motocicletas es el mismo y difiere del de las personas.

Por otro lado, la información sobre la clasificación que ofrece este submódulo no es definitiva y se contrasta con el submódulo de fusión y con el histórico de objetos. De esta forma se evitan errores puntuales en la clasificación como por ejemplo, se ha estado siguiendo una persona y resulta que en un determinado momento, bien por la fiabilidad de la zona o por oclusiones, la persona es clasificada como un vehículo. Esto no debería ocurrir

y la persona tras finalizar la ejecución del submódulo de fusión, debe estar clasificada como persona.

Para llevar a cabo esta clasificación el submódulo de *tracking* hace uso del submódulo de *raytracer* para estimar las dimensiones del objeto detectado. Obviamente estas dimensiones son estimaciones y no se pueden considerar como auténticas puesto que, dependiendo de la perspectiva del objeto y del punto de vista de la cámara, será más o menos fiable su información. Por esta razón se considera una clasificación a priori y no definitiva.

En el cálculo de la altura de un objeto detectado se calcula un plano perpendicular al suelo, paralelo al punto de vista de la cámara que contenga la posición donde el objeto fue detectado. Acto seguido se lanza un rayo por el punto medio superior del rectángulo que envuelve al movimiento de la imagen y se analiza dónde incide con este plano perpendicular. Una vez que se tiene dicho punto, se traza un vector entre este punto y la posición del objeto. Longitud de este vector representa a la altura del objeto. El algoritmo 11 se encarga de calcular la altura para un objeto detectado.

Algoritmo 11 Algoritmo para calcular la altura de un objeto detectado

Entrada: *Pixel* : El píxel que representa al punto medio superior del rectángulo del movimiento por el cual lanzar el rayo para calcular la altura.
Posicion : Posición 3D donde se encontró el objeto del cuál se va a calcular su altura.
Salida: *Altura* : La altura del objeto que se detectó en *Posicion*.

1: Crear el rayo teniendo en cuenta el *Pixel*, la distancia focal y la base ortonormal.
2: Obtener el vector normal (n) que define el plano del suelo. Este vector convertirlo a unitario.
3: Obtener el vector (w) de la base ortonormal de la fuente. Este vector convertirlo a unitario.
4: Calcular el producto vectorial: $proyeccion = n \times w$.
5: La normal del plano perpendicular y paralelo al plano vista es: $normal = proyeccion \times n$.
6: Crear el plano definido por *Posicion* y por *normal*.
7: Lanzar el rayo sobre el plano definido para obtener $Posicion_{altura}$.
8: Crear el vector entre *Posicion* y $Posicion_{altura}$ tal que: $Altura = longitud\ del\ vector$.
9: **devolver** *Altura*.

En cuanto al ancho es mucho más sencillo y consiste en lanzar un rayo por uno de los extremos inferiores del rectángulo para que incida con el suelo. Teniendo ese punto del suelo se crea un vector con la posición del objeto y la longitud del vector multiplicada por dos representa el ancho del objeto.

Con la altura y anchura estimada del objeto, el algoritmo de clasificación se basa en estos dos valores calculando la relación entre *altura/anchura* para distinguir entre vehículos y personas. Se debe conocer qué dimensión corresponde al ancho y alto porque dependiendo

de la perspectiva de la fuente, pueden estar intercambiadas. Para ello la aplicación compara las dimensiones con un modelo estándar de vehículo y de persona para conocer cuál de las dos dimensiones representa la altura y la anchura.

La relación *altura/anchura* para las personas tendrá un valor mayor que uno mientras que para los vehículos la relación será menor que uno. Para hacer la clasificación mucho más robusta. El algoritmo de clasificación utiliza otro parámetro que es el número de píxeles blancos que deja la mancha del movimiento. De esta forma para un grupo de personas se conoce que el número de píxeles blancos es mucho menor que para los vehículos. De esta forma se consigue hacer que la clasificación a priori sea mucho más robusta. El algoritmo 12 describe el comportamiento de la clasificación.

Algoritmo 12 Algoritmo para clasificar a priori entre persona y vehículo

Entrada: *Altura* : La altura del objeto que se pretende clasificar.
Anchura : El ancho del objeto que se pretende clasificar.
Numero_blancos : El número de blancos que definen el contorno del objeto detectado.
Salida: $Tipo$: El tipo del objeto que se le ha asignado.

1: Calcular $relacion = \frac{Altura}{Anchura}$.
2: **si** $relacion < 1$ Y $Numero\_blancos > Umbral\_Blancos$ **entonces**
3: $Tipo = vehiculo$.
4: **si no**
5: $Tipo = persona$.
6: **fin si**
7: **devolver** $Tipo$.

5.2.6. Submódulo de fusión

El submódulo de fusión es el encargado de componer la información que le van comunicando cada uno de los agentes que procesan el submódulo de *tracking*. Este submódulo, al igual que el de *raytracer*, también es procesado por un único agente. Los agentes encargados de los submódulos de *tracking* y clasificación se comunican con el agente encargado de la fusión para indicarle la información que ha sido detectada y clasificada. Este agente encargado del submódulo de fusión conoce en todo momento el número de agentes que le tienen que indicar información. Y cuando éste ha reunido toda la información que le tienen que comunicar, ejecuta el proceso de fusión que contiene este submódulo.

El submódulo de fusión debe poder fusionar información proveniente de *N* fuentes. A pesar de que en realidad se está trabajando con un máximo de tres fuentes, estos algoritmos deben estar diseñados y utilizar estructuras para almacenar la información de *N* fuentes. El

motivo por el que se ha decidido realizar una fusión de la información antes de procesar la información con el histórico de objetos es la eficiencia. De esta forma las heurísticas de identificación y el análisis del histórico de objetos se realiza una sola vez, mientras que si no se realizase la fusión serían aplicadas N veces. Los componentes que forman este submódulo son el histórico de objetos y las heurísticas de identificación.

Histórico de objetos

El histórico de objetos es la estructura donde se almacenan los objetos que han sido detectados durante la fase de aprendizaje. La estructura que represente el histórico debe ser eficiente en las operaciones de búsqueda de los objetos. Los objetos deben tener asociado un identificador único que permitan ser identificados a través de éste. Se ha decidido que este identificador único siga el formato:

$$tipo + contador\_tipo$$

El contador tipo consiste en indicar el número de objetos que han aparecido del mismo tipo, es decir, según se especificó en el submódulo de clasificación existen dos tipos (persona y vehículo), luego se tienen dos contadores (personas, vehículos). El contador de personas indica el número de personas que han aparecido hasta ese momento en el entorno a monitorizar y de forma análoga para el contador de vehículos. Algunos ejemplos de identificadores (persona1, persona2, persona3, vehículo1,...).

Se ha decidido utilizar como histórico de objetos un *diccionario* para hacer que las operaciones de búsqueda y eliminación de los objetos sea eficiente. Un *diccionario* es una estructura que está formada por pares <*clave, valor*>, de manera que la clave del histórico viene representada por el identificador del objeto y el valor representado por la información de ese objeto.

Con este diseño las operaciones de búsqueda y eliminación de objetos son tan simples como preguntar con el identificador como clave. Y son eficientes puesto que para encontrar un objeto no es preciso analizar todos los objetos que contiene el histórico.

Ahora bien el diseño del histórico presenta el inconveniente de que no permite almacenar la evolución [2] que el objeto tiene. Esto se debe a que las claves en el diccionario deben de ser únicas. Para solventar este problema en la información del objeto se debe almacenar dicha evolución. Luego cuando se pregunta al histórico con un identificador como clave, éste devuelve la evolución que ha tenido el objeto con dicho identificador.

[2] Las posiciones que el objeto ha ido tomando cuando se mueve por el entorno.

La información que debe recoger la evolución del objeto es:

- El listado de fechas en las que el objeto ha sido detectado.
- El listado de tiempos en los que el objeto ha sido detectado.
- El listado de los frames en los que el objeto fue detectado.
- El listado de las dimensiones que el objeto ha ido teniendo a lo largo de su detección.
- El listado de las posiciones que el objeto ha tenido durante su seguimiento.
- El listado de las fiabilidades que el objeto ha tenido.
- El listado del número de blancos que han definido el contorno del objeto durante su seguimiento.
- El tipo de zona de entrada/salida en el que el objeto se encuentra en este momento.

Heurísticas de Identificación

En este apartado se describen las heurísticas utilizadas para la fusión de la información de las *N* fuentes, para la detección de los objetos que han abandonado el entorno monitorizado y para identificar que la información resultante de la fusión representa parte de la evolución de un objeto que anteriormente fue detectado.

Antes de poder trabajar analizando el histórico se debe de fusionar la información de las *N* fuentes. Para compactar dicha información el algoritmo de fusión debe analizar todos los objetos con todos. Es decir, suponiendo que se tienen tres fuentes los objetos de la fuente 1 serán analizados con los objetos de la fuente 2, los de la 2 con la fuente 3 y los de la fuente 3 con los de la fuente 1. El algoritmo de identificación de objetos iguales detectados por otras fuentes se fundamenta en la distancia mínima entre dos objetos (*función euclidea*). Pero en entornos reales se da el caso en que la segmentación y el posicionamiento de las fuentes no tienen una precisión excelente. Por lo que para hacer más robusto el algoritmo de identificación de objetos iguales se busca el mejor candidato más próximo a él.

Por lo tanto, dada la lista de objetos de la fuente 1 y la lista de objetos de la fuente 2 se pretenden buscar los objetos iguales entre estas dos listas. Para ello se coge el primer objeto de la fuente 1 y se analiza la distancia con los objetos de la fuente 2. El objeto de la fuente 1 se considera igual con el objeto de la fuente 2 que presente menor distancia. Dicha distancia no debe superar un *umbral* y que el candidato de la fuente 2 con el que presenta dicha distancia no esté elegido por otro objeto de la fuente 1 que presente aún menor valor

de distancia. Si la menor distancia supera el *umbral* se debe a que el objeto de la fuente 1 se encuentra en una zona que es disjunta a la vista de la fuente 2, luego no será identificado con ningún objeto de la fuente 2.

En caso que el candidato de la fuente 2 elegido por el objeto de la fuente 1 haya sido elegido por otro objeto de la fuente 1 habría conflicto. Para solucionar este conflicto el objeto de la fuente 2 se quedaría ligado con aquel objeto de la fuente 1 que presentase menor distancia. Y para el otro objeto de la fuente 1 se tendría que volver a buscar su igualdad con el siguiente objeto de la fuente 2 más próximo a él y que no este elegido por otro mejor que él. El algoritmo 13 describe el funcionamiento de lo explicado anteriormente donde se obtienen las conexiones de los objetos que se consideran que son iguales.

Algoritmo 13 Algoritmo para identificar objetos iguales detectados por diferentes fuentes

Entrada: $Objetos\_detectados$: Es una lista de las listas de los objetos detectados por cada una de las fuentes.
Salida: $Objetos\_iguales$: Los objetos que se deben fusionar porque son iguales.

para todo $lista\_1$ y $lista\_2$ de $Objectos\_dectectados$ donde $lista\_2 \neq lista\_1$ **hacer**
    Inicializar *conectados* a ninguno.
    para todo obj_1 de $lista\_1$ **hacer**
        $obj\_candidato = NULL$.
        $min\_distancia = +\infty$.
        $distancias = Lista\_vacia$.
        para todo obj_2 de $lista\_2$ **hacer**
            Calcular *distancia* euclidea entre obj_1 y obj_2.
            si $Umbral \geq distancia$ Y $min\_distancia > distancia$ **entonces**
                $obj\_candidato = obj\_2$
                $min\_distancia = distancia$
            fin si
            Añadir *distancia* a la lista *distancias*.
        fin para
        si $obj\_candidato \neq NULL$ **entonces**
            si $obj\_candidato$ no está elegido en *conectados* **entonces**
                Indicar en *conectados* la conexión de obj_1 con $obj\_candidato$.
            si no
                Resolver conflicto buscando otro minimo en *distancias* y resolviendo los conflictos que puedan derivar.
            fin si
        fin si
    fin para
    En *conectados* se encuentran las conexiones entre los objetos de la $lista\_1$ y $lista\_2$ y deben ser almacenadas en $Objetos\_iguales$.
fin para
devolver $Objetos\_iguales$.

Pero el algoritmo anterior no mezcla la información. Para mezclar la información de los objetos se ha tenido en cuenta que la información que recoge cada fuente es distinta. Es decir, cada una ofrece dimensiones, posiciones, tipos de zona y objeto diferentes. Por ejemplo una fuente puede decir que es una persona, mientras que otras dicen que es un vehículo, o decir que se encuentra en zona de entrada y las otras en zona neutra. Para solucionar estas distinciones en la información se ha pensado en utilizar una media que esté ponderada en función de la fiabilidad que tiene la zona donde fue visto el objeto.

Por lo tanto la fusión de la información con los objetos conectados mediante el algoritmo 13 se realiza mediante la ecuación 5.4:

$$\frac{\sum_{i=1}^{N} caracteristica \times fiabilidad}{\sum_{i=1}^{N} fiabilidad} \tag{5.4}$$

Una vez compactada la información de las N fuentes por el algoritmo de fusión, a la información compactada se le aplica una heurística de identificación con los objetos que hay en el histórico. El objetivo de aplicar esta heurística es asignar identificadores a la información compactada. El funcionamiento de la heurística de identificación consiste en analizar cada uno de los objetos de la información compactada con cada uno de los objetos que se encuentran almacenados en el histórico.

En dicho análisis se contrasta la información del objeto nuevo con la información del objeto del histórico. Y se asigna una relación entre ellos marcada por un peso. Para el cálculo de este peso se tiene en cuenta la diferencia entre sus posiciones y se le aplica una penalización en caso de que sus tipos (persona o vehículo) sean diferentes. Después de analizar los objetos nuevos con los del histórico y conocer la lista de pesos que tiene el objeto nuevo con los del histórico si el objeto nuevo se encuentra en zona neutra se le asigna la asociación con menor peso, es decir, hereda el identificador del objeto del histórico con menor peso. Si por el contrario el objeto nuevo no se encuentra en zona neutra, éste hereda el identificador del objeto del histórico que tiene menor peso siendo menor que un umbral. Si el menor peso es mayor que el umbral entonces el objeto nuevo representa a un nuevo objeto que ha aparecido y se le asigna un nuevo identificador. El algoritmo 14 muestra el funcionamiento de la identificación de los nuevos objetos con los del histórico.

Tras asignar los identificadores se aplica un algoritmo para detectar cuando los objetos se han marchado, se han juntado o se han separado. Este algoritmo consiste en comparar los identificadores de los objetos nuevos con los identificadores de los objetos que fueron vistos la última vez.

Algoritmo 14 Algoritmo para asignar identificadores a los objetos detectados

Entrada: $Objetos\_detectados$: La lista de objetos resultantes de fusionar la información de las N fuentes.
$Historico$: El historico de objetos desde que empezó el *tracking*.

Salida: $Objetos\_identificados$: Se devuelven los objetos de entrada pero éstos ya tienen su identificador.

para todo obj de $Objetos\_detectados$ **hacer**
    para todo $antiguo$ de $Historico$ **hacer**
        Asignar $peso$ entre obj y $antiguo$.
    fin para
fin para
para todo obj de $Objetos\_detectados$ **hacer**
    Obtener $antiguo$ con menor $peso$.
    si $obj.zona = neutra$ **entonces**
        $obj.id = antiguo.id$.
    si no
        si $peso < Umbral$ **entonces**
            $obj.id = antiguo.id$.
        si no
            Asignar nuevo identificador a obj.
        fin si
    fin si
fin para
devolver $Objectos\_identificados$.

Antes de describir su funcionamiento, se debe mencionar que se ha decidido que cuando dos objetos se juntan o se separan deben adquirir nuevos identificadores. Las razones de esta decisión son porque cuando se unen habría que elegir uno de los dos identificadores para dárselo al que forma la unión de éstos y esto no es justo puesto que una aplicación que analizase los comportamientos pensaría que uno de los dos objetos desapareció; y en cuanto a la separación no se puede dar que dos objetos lleven asociado el mismo identificador del objeto que los agrupaba. Luego por estas razones se realiza una asignación de nuevos identificadores para la unión y separación de los objetos.

La forma que tiene el algoritmo para detectar que hay objetos que se han separado, es analizando los identificadores de los objetos nuevos y si tras este análisis se encuentran identificadores repetidos, es porque esos objetos representaban anteriormente a ese objeto. Luego se asignan nuevos identificadores a los objetos con identificadores iguales y el objeto almacenado en el histórico con el identificador que ocasionaba la igualdad es eliminado porque ha dado lugar a nuevos objetos. El algoritmo 15 describe la separación de objetos.

Algoritmo 15 Algoritmo para detectar la separación de objetos

Entrada: $Objetos\_nuevos$: Los objetos nuevos con los identificadores que han resultado de aplicar el algoritmo 14.
$Historico$: El historico que se tuvo hasta el momento actual.

```
1:  para todo obj de Objetos_nuevos hacer
2:      cambiar_id = falso .
3:      para todo obj1 de Objetos_nuevos Y obj1 ≠ obj hacer
4:          si obj.id = obj1.id entonces
5:              Asignar nuevo id a obj1 porque se han separado.
6:              cambiar_id = cierto .
7:          fin si
8:      fin para
9:      si cambiar_id = cierto  entonces
10:         Asignar nuevo id a obj porque se han separado.
11:         Remover del Historico el objeto con id obj.id.
12:     fin si
13: fin para
```

Para detectar que hay objetos que se marcharon del entorno, se analiza si existe algún identificador de los objetos vistos en el anterior frame que no se encuentre asignado a ninguno de los objetos nuevos. Si existe algún identificador antiguo que no aparece y el objeto con ese identificador se encontraba en zona de entrada/salida o salida y también se había predicho que iba a salir (predicción que se hace cuando el objeto pasa de estar en zona neutra a estar en zona de salida o entrada/salida), entonces el objeto con ese identificador es

eliminado del histórico debido a que se ha marchado. El algoritmo 16 describe la salida de objetos del entorno.

Algoritmo 16 Algoritmo para detectar la salida de objetos

Entrada: $Objetos\_nuevos$: Los objetos nuevos con los identificadores que han resultado de aplicar el algoritmo 14.
$Historico$: El historico que se tuvo hasta el momento actual.

```
1:  para todo obj de Historico hacer
2:      si obj.zona = salida O (obj.zona = entrada/salida Y obj.se_puede_marchar =
        cierto ) entonces
3:          se_ha_marchado = cierto .
4:          para todo obj1 de Objetos_nuevos Y se_ha_marchado = cierto  hacer
5:              si obj.id = obj1.id entonces
6:                  se_ha_marchado = falso .
7:              fin si
8:          fin para
9:          si se_ha_marchado = cierto  entonces
10:             Remover del Historico el objeto con id obj.id.
11:         fin si
12:     fin si
13: fin para
```

En cuanto para detectar la unión entre objetos se analiza si existen identificadores de objetos que fueron vistos anteriormente y no aparecen en los objetos nuevos. Si los objetos asociados a esos identificadores se encontraban próximos entre ellos y en zona neutra, entrada o en entrada/salida con la condición de que el objeto estaba entrando entonces se busca en la lista de objetos nuevos el más próximo a estos objetos antiguos. Aquel objeto nuevo más próximo representará al objeto unión de los objetos antiguos y su identificador representará al objeto antiguo del que heredó su identificador por ser el más próximo. Por lo tanto se eliminan del histórico los objetos antiguos que no aparecen en los nuevos más el objeto asociado al identificador del objeto nuevo más cercano; y a este objeto nuevo se le asigna un nuevo identificador. El algoritmo 17 describe la unión de los objetos.

Por último tras aplicar este algoritmo lo único que queda es actualizar el histórico según la información de los nuevos objetos, y un matiz a resaltar es que en la actualización de la información se da más importancia a la información antigua que a la nueva. Por lo tanto, si la información nueva difiere en gran medida a la antigua se mantiene la información antigua. Con esto se evita que si siempre un objeto ha sido detectado como persona y ahora es detectado como un coche que no sea clasificado como coche. Para que empiece a clasificarse como coche, se le tiene que estar indicando un número de veces superior al

Algoritmo 17 Algoritmo para detectar la unión de objetos

Entrada: $Objetos\_nuevos$: Los objetos nuevos con los identificadores que han resultado de aplicar el algoritmo 14.
$Historico$: El historico que se tuvo hasta el momento actual.

1: **para todo** obj de $Historico$ **hacer**
2: **si** $obj.zona = neutra$ O $obj.zona = entrada$ O $(obj.zona = entrada/salida$ Y $obj.ha\_entrado =$ **cierto** $)$ **entonces**
3: $se\_encuentra =$ **falso** .
4: **para todo** $obj1$ de $Objetos\_nuevos$ Y $se\_encuentra =$ **falso hacer**
5: **si** $obj.id = obj1.id$ **entonces**
6: $se\_encuentra =$ **cierto** .
7: **fin si**
8: **fin para**
9: **si** $se\_encuentra =$ **falso entonces**
10: Incluir $obj.id$ en la lista de objetos desaparecidos.
11: **fin si**
12: **fin si**
13: **fin para**
14: Agrupar de la lista de desaparecidos aquellos que estén próximos.
15: **para todo** $grupo$ de $grupos\_desparecidos$ **hacer**
16: **para todo** obj de $Objetos\_nuevos$ **hacer**
17: Registrar la distancia del obj con los objetos del $grupo$.
18: **fin para**
19: **fin para**
20: **para todo** $grupo$ de $grupos\_desparecidos$ **hacer**
21: Elegir el objeto obj con menor distancia hacia al $grupo$.
22: Remover del histórico los objetos que tengan los identificadores del $grupo$.
23: Remover del histórico el objeto que tenga el identificador $obj.id$.
24: Asignar un nuevo identificador al objeto obj, el cuál representa la unión.
25: **fin para**

que se le indicó cuando era persona. El algoritmo 18 representa el funcionamiento de la actualización del histórico.

Algoritmo 18 Algoritmo para actualizar el histórico

Entrada: $Objeto\_historico$: El objeto almacenado en el histórico al cuál se le va a actualizar la información.
$Objeto\_nuevo$: El objeto nuevo detectado cuya información va a actualizar al $objeto\_historico$.

Salida: $Objeto\_actualizado$: El objeto histórico actualizado con la información del objeto nuevo.

1: $Objeto\_historico.nueva\_fecha(Objeto\_nuevo.fecha)$.
2: $Objeto\_historico.nueva\_hora(Objeto\_nuevo.hora)$.
3: $Objeto\_historico.nuevo\_frame(Objeto\_nuevo.frame)$.
4: **si** $Diferencia(Objeto\_historico.ancho,\ Objeto\_nuevo.ancho) < Umbral$ **entonces**
5: $Objeto\_historico.nuevo\_ancho(Objeto\_nuevo.ancho)$.
6: **si no**
7: $Objeto\_historico.nuevo\_ancho(Objeto\_historico.ancho)$.
8: **fin si**
9: **si** $Diferencia(Objeto\_historico.alto,\ Objeto\_nuevo.alto) < Umbral$ **entonces**
10: $Objeto\_historico.nuevo\_alto(Objeto\_nuevo.alto)$.
11: **si no**
12: $Objeto\_historico.nuevo\_alto(Objeto\_historico.alto)$.
13: **fin si**
14: **si** $Diferencia(Objeto\_historico.grosor,\ Objeto\_nuevo.grosor) < Umbral$ **entonces**
15: $Objeto\_historico.nuevo\_grosor(Objeto\_nuevo.grosor)$.
16: **si no**
17: $Objeto\_historico.nuevo\_grosor(Objeto\_historico.grosor)$.
18: **fin si**
19: $Objeto\_historico.nueva\_posicion(Objeto\_nuevo.posicion)$.
20: Asignar el tipo dependiendo del número de veces clasificado como persona y vehículo.
21: Asignar el nuevo tipo de zona, analizando si pasa de entrada a neutra o de neutra a salida.
22: **devolver** $Objecto\_actualizado$.

5.2.7. Submódulo de depuración

El submódulo de depuración es el encargado de probar algunos submódulos del módulo de procesamiento. Las pruebas que se han realizado a los submódulos de despliegue, de raytracer y de *tracking* han sido de *caja negra*. Las pruebas de caja negra consisten en aplicar a la entidad a probar una serie de entradas y comprobar las salidas obtenidas con las que se esperaban.

El aplicar el submódulo de depuración durante el desarrollo de los submódulos de despliegue, de raytracer y de *tracking* permite que se tenga la certeza de que el funcionamiento de estos submódulos por separado es correcto, por lo tanto no debería haber problema cuando éstos se conecten entre ellos. Si cuando se conectan entre ellos, se produce un comportamiento incorrecto éste se debe a la conexión entre ellos. Pero si por el contrario no se hubiese aplicado este submódulo, el cerco para analizar de donde vienen los errores sería tan grande que los tiempos y los costes del desarrollo de la aplicación aumentarían.

Los componentes del submódulo de despliegue que se probaron con este submódulo de depuración fueron los de *Calibrado* y *Posicionamiento*. Debido a la naturaleza de los procesos de calibrado y posicionamiento se presentaba el problema de cómo conocer las salidas esperadas para que las pruebas de caja negra tuviesen éxito. Para solucionar este problema se pensó en utilizar *Blender* (sección 3.6.2), herramienta que permite el uso de cámaras virtuales y de estas cámaras virtuales se pueden consultar los parámetros de calibración y la matriz de transformación (sección 3.5.1) asociados a ellas.

Para depurar el proceso de calibrado se creaban mediante *blender* una serie de tomas del tablero de ajedrez, y dichas tomas eran procesadas por el proceso de calibrado y se obtenian los resultados de la calibración. Dichos resultados se comparaban con los resultados obtenidos de blender mediante el siguiente *script*:

```
1  import Blender
2  import math, sys, os
3  from Blender import NMesh, Object, Material, Lamp
4  from Blender.Mathutils import *
5  from math import *

7  def convFloat (vfloat):
8          return str('%f' %vfloat) + ' '

10 filename = os.getcwd() + "/calibracion_blender.txt"
11 file = open(filename, "w")

13 file.write("Distorsion Blender: 0, 0, 0, 0, 0\n")
14 file.write("Matriz de la camara Blender:\n")
15 for ob in Object.Get():
16         if ob.getType() == 'Camera':
17                 camara = ob.getData()
18                 ancho = Blender.Scene.GetCurrent().getRenderingContext().imageSizeX()
19                 alto =  Blender.Scene.GetCurrent().getRenderingContext().imageSizeY()
20                 distancia_focal = camara.getLens()/32.0
21                 file.write(convFloat(distancia_focal*ancho)+ "\t" + convFloat(0.0) + "\t"
                       + convFloat(ancho/2.0))
22                 file.write("\n")
23                 file.write(convFloat(0.0)+ "\t" + convFloat(distancia_focal*ancho) + "\t"
                       + convFloat(alto/2.0))
24                 file.write("\n")
25                 file.write(convFloat(0.0)+ "\t" + convFloat(0.0) + "\t" + convFloat(1.0))
```

LISTADO 5.5: Script para exportar de Blender los datos de calibración

En cuanto a la depuración del proceso de posicionamiento se reutilizaban las tomas creadas para el proceso de calibrado. De todas esas tomas se especificaba una de ellas para calcular la posición y orientación de la cámara cuando se realizo dicha toma. Los

resultados generados se comparaban con los resultados esperados que previamente se habían consultado en blender mediante el siguiente *script*:

```
1  import Blender
2  import math, sys, os
3  from Blender import NMesh, Object, Material, Lamp
4  from Blender.Mathutils import *
5  from math import *

7  def convFloat (vfloat):
8          return str('%f' %vfloat) + ' '

10 filename = os.getcwd() + "/position_blender.txt"
11 file = open(filename, "w")
12 file.write("Matriz de transformacion obtenida por blender:\n")
13 for ob in Object.Get():
14         if ob.getType() == 'Camera':
15                 matrix = ob.getMatrix() #Matriz 3x4 que es la buscada.
16                 file.write(convFloat(matrix[0][0])+ "\t" + convFloat(matrix[0][1]) + "\t"
                       + convFloat(matrix[0][2]))
17                 file.write("\n")
18                 file.write(convFloat(matrix[1][0])+ "\t" + convFloat(matrix[1][1]) + "\t"
                       + convFloat(matrix[1][2]))
19                 file.write("\n")
20                 file.write(convFloat(matrix[2][0])+ "\t" + convFloat(matrix[2][1]) + "\t"
                       + convFloat(matrix[2][2]))
21                 file.write("\n")
22                 file.write(convFloat(matrix[3][0])+ "\t" + convFloat(matrix[3][1]) + "\t"
                       + convFloat(matrix[3][2]))
23                 file.write("\n")
```

LISTADO 5.6: Script para exportar de Blender los datos de posicionamiento

Para desempeñar la depuración del submódulo del *raytracer* se creó una escena con una serie de objetos en *blender*. Dicha escena se exportó a un archivo de texto mediante el siguiente *script*:

```
1  import Blender
2  import math, sys, os
3  from Blender import NMesh, Object, Material, Lamp
4  from Blender.Mathutils import *
5  from math import *

7  def mulmatvec4x3(a, b):
8          # a is vector, b is matrix
9          r = [0, 0, 0]
10         r[0] = a[0]*b[0][0] + a[1]*b[1][0] + a[2]*b[2][0] + b[3][0]
11         r[1] = a[0]*b[0][1] + a[1]*b[1][1] + a[2]*b[2][1] + b[3][1]
12         r[2] = a[0]*b[0][2] + a[1]*b[1][2] + a[2]*b[2][2] + b[3][2]
13         return r

15 def convFloat (vfloat):
16         return str('%f' %vfloat) + ' '

18 filename = os.getcwd() + "/vertex_scene.txt"
19 file = open(filename, "w")

21 for ob in Object.Get():
22         if ob.getType() == 'Mesh':
23                 file.write ('# TRIANGLE   V0y      V0z     |    V1x      V1y      V1z   |
                        V2x      V2y      V2z\n')
24                 m = ob.getData()
25                 matrix = ob.getMatrix()
26                 xo,yo,zo = ob.getLocation()
27                 if len(m.verts) > 0:
28                         for face in m.faces:
29                                 file.write('t ')
30                                 face.v.reverse()
31                                 for vertex in face.v:
32                                         vect = vertex.co
```

```
33                                      vt = mulmatvec4x3(vect, matrix)
34                                      file.write (convFloat(vt[0]) + convFloat(vt[1]) +
                                            convFloat(vt[2]))
35                              file.write ('\n')
36          if ob.getType() == 'Camera':
37                  file.write ('# CAMERA: Position(x,y,z)  | Look  (Vector) |  Up  (Vector)
                        |  Lens  |  Name\n')
38                  file.write ('c ')
39                  x,y,z = ob.getLocation("worldspace")
40                  look = [0,0,-1]
41                  up = [0,1,0]
42                  matrix = ob.getMatrix()
43                  lookt = mulmatvec4x3(look, matrix)
44                  upt = mulmatvec4x3(up, matrix)
45                  file.write(convFloat(x) + convFloat(y) + convFloat(z))
46                  file.write(convFloat(lookt[0]) + convFloat(lookt[1]) + convFloat(lookt[2])
                        )
47                  file.write(convFloat(upt[0]) + convFloat(upt[1]) + convFloat(upt[2]))
48                  camara = ob.getData()
49                  file.write(convFloat(camara.getLens()))
50                  file.write(ob.getName())
51                  file.write("\n")
```

LISTADO 5.7: Script para exportar de Blender la información de los objetos de una escena

Ese archivo de texto fue procesado por el submódulo del *raytracer* para crear el mundo 3D de acuerdo a la escena realizada con *blender*. Una vez creado el mundo 3D se ejecutaba el proceso del *raytracer* y los resultados eran exportados a un archivo de texto. Después para comprobar que el funcionamiento del *raytracer* era el correcto mediante otro *script* se procesaron los resultados almacenados en el archivo de texto para proyectarlos sobre la escena y visualizar si eran correctos. Ese *script* de importación fue:

```
1   import Blender
2   import math, sys, os
3   import string
4   from Blender import NMesh

6   print "Escriba el nombre del archivo para cargar:"
7   name = raw_input()
8   filename = os.getcwd() + "/" + name
9   file = open(filename, "r")
10  scene = Blender.Scene.GetCurrent()
11  me = NMesh.New('TestMesh')

13  for line in file.readlines():
14          x = string.atof(string.split(line)[2])
15          y = string.atof(string.split(line)[3])
16          z = string.atof(string.split(line)[4])
17          if (x < 9999):
18                  me.verts.append(NMesh.Vert(x,y,z))


21  obj = Blender.Object.New('Mesh', 'myObj')
22  obj.link(me)
23  scene.link(obj)
24  Blender.Redraw()
```

LISTADO 5.8: Script para importar a Blender los puntos donde incidieron los rayos del *raytracer*

Y por último para depurar el submódulo de *tracking* y detectar que el proceso de segmentación funcionaba bien, se modelo un entorno que reproducía al entorno real de la calle *Paseo de la Universidad*. De este entorno se generó un vídeo, el cuál se aplicó al submódulo de *tracking* para analizar el comportamiento del proceso de segmentación. En

los resultados del proceso de segmentación se distinguían perfectamente los objetos móviles de los inmóviles.

Los resultados obtenidos de aplicar este submódulo de depuración a los mencionados se detallan en el capítulo 6.

5.3. Módulo de gestión de agentes

El módulo de gestión de agentes es el encargado de realizar las tareas de sincronización entre los agentes encargados en las funcionalidades de la aplicación. Para las funcionalidades de calibración, posicionamiento, definición de áreas fiables y de I/O no es necesaria la cooperación entre los agentes. Permitiendo que dichas funcionalidades puedan ser llevadas a cabo por agentes independientes sin necesidad de comunicarse entre ellos.

En cuanto a la fase de aprendizaje tampoco es necesario que éste sincronizada con las demás fases de aprendizaje realizadas por otros agentes y por lo tanto no requiere la comunicación con otros agentes. Pero una vez que el aprendizaje concluye los agentes que se encuentren en fase de *tracking* deben ser sincronizados.

Para que el módulo de gestión de agentes conozca en todo momento el estado y los agentes asociados a dicho estado en el que se encuentran las fuentes conectadas se ha diseñado el concepto del estado asociado a cada fuente conectada al sistema. Este estado puede tomar los siguientes valores:

- Calibrado. Indica que la fuente está en periodo de calibración.
- Posicionamiento. Indica que la fuente se encuentra en el proceso de posicionamiento.
- Definición de áreas I/O. Indica que la fuente está en periodo de definición de áreas I/O.
- Definición de áreas fiables. Indica que la fuente se encuentra en el proceso de definición de áreas fiables.
- Aprendizaje. Indica que la fuente se encuentra en el proceso de aprendizaje.
- *Tracking*. Indica que la fuente está en periodo de seguimiento.

El módulo de gestión de agentes consulta los valores de los estados de las fuentes para conocer en qué fase se encuentran y dependiendo de la fase en la que estén se proporcionará el soporte adecuado. Dicha consulta ha sido diseñada como una *máquina de estados*. En

la máquina de estados se contempla que el usuario pueda cancelar el proceso que se está aplicando sobre una fuente determinada para volverlo a realizar en otro momento determinado. La cancelación por parte del usuario de un proceso obligará al módulo de gestión de agentes comunicarse con el agente encargado de desempeñar ese proceso para que aborte su ejecución.

El proceso de sincronización y de comunicación de la información se debe dar cuando alguna de las fuentes conectadas pasa del estado de aprendizaje al estado de *tracking*. La sincronización consistirá en hacer que los agentes que se encuentren en la fase de *tracking* realicen la lectura de las fuentes *stream* al mismo tiempo. Esta sincronización es fundamental para que luego el proceso de fusión de la información tenga éxito, porque si la lectura va desincronizada algunas captarán el objeto antes que otras o en posiciones diferentes y se generará un comportamiento no deseado. Si las fuentes conectadas son de tipo *archivo de vídeo* el proceso de sincronización a parte de sincronizar los tiempos en los que se tienen que realizar la lectura de las fuentes *stream* también tiene que tener en cuenta que se esté leyendo el mismo frame y que los archivos de vídeo asignados a cada una de las fuentes *stream* tengan el mismo número de frames. El motivo por el que se tiene que controlar el número de frame es porque cada fuente *stream* puede acabar la fase de aprendizaje en un frame diferente, y para sincronizarlas es preciso que se ajusten todas al mismo frame. La forma de controlar esto es registrando cuando una fuente pasa de aprendizaje a *tracking*. Si es la primera fuente que ha pasado de aprendizaje a *tracking* entonces es anotado el frame en el que se encuentra y a partir de ahora las fuentes que vayan entrando al proceso de *tracking* llevarán la lectura a partir de este frame aunque éstas hayan acabado la fase de aprendizaje en una posición de frame anterior o posterior. El algoritmo 19 describe en pseudocódigo la forma de realizar este registro.

También en el proceso de sincronización se debía tener en cuenta el detalle de que debido a la complejidad que tienen los algoritmos de *tracking* realizar el análisis de los veinticinco frames por segundo era tarea imposible. Luego había dos formas de hacerlo:

1. Que la sincronización de la lectura de los frames se realizase en intervalos de tiempo irregulares. Se dice que son intervalos de tiempo irregulares porque depende del tiempo que tarden los algoritmos (segmentación, clasificación, *tracking*, fusión, etc) en procesar el frame.

2. Que la sincronización de la lectura de los frames se realizase en intervalos de tiempo regulares. Esto consistiría en realizar siempre la lectura de los frames de las fuentes en intervalos regulares, pero solamente se analizarían aquellos frames en los que los agentes encargados de realizar el *tracking* se encuentren en estado ocioso.

Algoritmo 19 Algoritmo de registro de las fuentes que pasan del aprendizaje al *tracking*

Entrada: Id : El identificador de la fuente que ha acabado el aprendizaje.
$Tipo$: El tipo de fuente que ha acabado el aprendizaje.
$Fuente$: El objeto fuente que ha acabado el aprendizaje.

1: Leer en exclusión mutua la variable que indica el número de fuentes que hay en *tracking*.
2: **si** $n\_fuentes\_tracking = 0$ **entonces**
3: **si** $Tipo = Archivo\_video$ **entonces**
4: Registrar el frame donde acabo el aprendizaje $(Fuente.Frame)$ y el final del video $(Fuente.Final\_frame)$.
5: **fin si**
6: Crear el agente encargado de la sincronización y coordinación cuyo funcionamiento viene descrito en el algoritmo 16.
7: **fin si**
8: Hacer un *wait* del semáforo que protege el acceso a las variables de estado de las fuentes.
9: Cambiar el estado de la fuente a *tracking*: $estados[Id] = tracking$.
10: Incrementar el número de fuentes en *tracking*.
11: Hacer un *signal* del semáforo que protege el acceso a las variables de estado de las fuentes.

En el diseño del proceso de sincronización se tuvo en cuenta la opción 2. De esta forma se estará analizando los frames que den tiempo dependiendo de la carga del sistema y la velocidad en la visualización de los frames de las fuentes se mantendrá a veinticinco frames por segundo. Esto último es lo que diferencia la opción 1 de la opción 2 y evita la sensación de lentitud en la reproducción de las fuentes.

Resumiendo el módulo de gestión de agentes registra cuando una fuente pasa del aprendizaje al *tracking* para asignarle un agente. Si es la primera en hacerlo y de tipo *archivo de vídeo* entonces también anota el frame desde el cuál se sincronizarán las demás fuentes. En intervalos regulares de 40 *ms* analiza si los agentes encargados de hacer el *tracking* están ociosos, de serlo así los despierta para que lean el frame actual de la fuente y comiencen a analizarlo. Si están ocupados entonces invoca a la visualización del frame para que no de la sensación de lentitud. Cuando los agentes están realizando el *tracking* y van terminando le van comunicando al agente que se encarga de la fusión la información. Cuando el agente encargado de realizar el *tracking* le comunica la información al agente de la fusión, éste se duerme esperando a que sea llamado de nuevo por el módulo de gestión de agentes para que procese un nuevo frame. Después de que el agente reciba toda la información necesaria (él sabe en todo momento cuantos agentes están realizando el *tracking* para esperarles a que le envíen su información) comienza a fusionarla. Y cuando finaliza el proceso de fusión, es este agente el encargado de indicar que los agentes de *tracking* se encuentran en estado

ocioso, para que el módulo de gestión de agentes despierte de nuevo a los agentes de *tracking*. La comunicación de la información entre los agentes se realiza mediante *memoria compartida* y la sincronización mediante el uso de *semáforos*. El algoritmo 20 describe en pseudocódigo la forma de llevar a cabo esta sincronización y coordinación con los agentes encargados del *tracking*.

Algoritmo 20 Algoritmo de sincronización y coordinación con los agentes del *tracking*

```
1: Copiar en exclusión mutua los valores de las variables de estado (estados_fuentes, n_fuentes_tracking).
2: Asignado = falso
3: Hacer un wait del semáforo que protege la consulta al estado del agente de fusión.
4: si fusion_terminada = cierto entonces
5:     Exportar y comunicar al agente de la vista 3D los objetos detectados.
6:     Indicar al agente de fusión que ya se procesó la información de los objetos detectados.
7:     para todo fuente que tenga la copia del estado en tracking hacer
8:         si fuente.tipo = Archivo_video entonces
9:             Coger el frame indicado por contador_frame.
10:        si no
11:            Coger frame actual de la fuente.
12:        fin si
13:        Despertar agente de tracking para que procese el frame cogido anteriormente.
14:    fin para
15:    Asignado = cierto
16: fin si
17: Hacer un signal del semáforo que protege la consulta al estado del agente de fusión.
18: para todo fuente que tenga la copia del estado en tracking hacer
19:    si Asignado = falso entonces
20:        si fuente.tipo = Archivo_video entonces
21:            Coger el frame indicado por contador_frame.
22:        si no
23:            Coger frame actual de la fuente.
24:        fin si
25:    fin si
26:    Visualizar el frame cogido anteriormente.
27: fin para
28: Incrementar contador_frame.
```

5.3.1. Submódulo de Agente

El submódulo de agente representa a la entidad de un agente en la aplicación. Todo agente de la aplicación está compuesto por un *hilo de ejecución* para que su procesamiento sea independiente del resto de agentes y de la propia aplicación. También se necesita que el agente utilice un *espacio de memoria independiente* que le permita almacenar la información que va generando.

El agente recogerá a partir de sus sensores la información que luego procesará mediante el uso de los submódulos y heurísticas definidos en el módulo de procesamiento. Los sensores de los agentes para percibir la información y los eventos indicados por el usuario harán uso del módulo de entrada.

Una vez que los agentes hayan percibido la información por los sensores comienzan a procesarla dependiendo del proceso que estén desempeñando. La información que vayan generando los agentes será comunicada hacia otros agentes mediante el módulo de gestión de agentes y comunicada hacia el exterior por medio de los módulos de exportación y visualización.

5.4. Módulo de visualización

El módulo de visualización es el encargado de la representación de la información de cada uno de los procesos que se llevan a cabo en los agentes. Éste está compuesto por dos submódulos. Para implementar los efectos que se decidieron en el diseño del módulo de visualización se ha utilizado la biblioteca *OpenGL* (ver sección 3.6.3). El módulo de visualización contiene componentes de tipo *gtkglext* donde se van a representar los frames que capturan las fuentes y aplicar los comandos de *OpenGL*.

5.4.1. Submódulo de reproducción del flujo de *stream*

Submódulo que se encarga de representar en el *gtkglext* el estado del proceso que se está llevando a cabo en la fuente. Cada fuente va a tener asociado su propio submódulo de reproducción del flujo de *stream*. El submódulo de reproducción de flujo de *stream* también ha sido diseñado como una máquina de estado debido a que tiene diferente comportamiento dependiendo del proceso que se esté realizando.

Para representar los frames que está captando la fuente sobre el *gtkglext* se crea una textura con la imagen del frame y un plano paralelo al plano de la cámara virtual del *gtkglext*

de manera que ocupe toda vista. Después se asocia la textura al plano definido. El algoritmo 21 describe los comandos de *OpenGL* que se deben realizar para reproducir los frames de la fuente en *gtkglext*.

Algoritmo 21 Algoritmo para la reproducción del flujo de *stream*

Entrada: *Frame* : La imagen de la fuente que se pretende visualizar.

1: Limpiar los buffers mediante: $glClear(GL\_COLOR\_BUFFER\_BIT)$.
2: Cargar la matriz de proyección: $glMatrixMode(GL\_PROJECTION)$.
3: Desactivar la profundidad: $glDisable(GL\_DEPTH\_TEST)$.
4: **si** $primera\_textura =$ **cierto entonces**
5: Generar un identificador para la textura y la inicializamos con los comandos: *glGenTextures*, *glBindTexture*, *glPixelStorei* y *glTexParameteri*.
6: Asignar la imagen a la textura con el comando $glTexImage2D(Frame)$.
7: $primera\_textura =$ **falso** .
8: **si no**
9: Buscar la textura que se creo con el comando *glBindTexture*.
10: Asignar la imagen a la textura encontrada mediante el comando *glTexSubImage2D*.
11: **fin si**
12: Activar el modo de textura en 2D mediante: $glEnable(GL\_TEXTURE\_2D)$.
13: Pintar el plano con la textura mediante los comandos: $glBegin(GL\_QUADS)$, $glTexCoord2f$ y $glVertex2d$.
14: Desactivar el modo de textura en 2D mediante: $glDisable(GL\_TEXTURE\_2D)$.
15: Activar la profundidad: $glEnable(GL\_DEPTH\_TEST)$.

En el proceso de calibrado

En el proceso de calibración se está reproduciendo en intervalos regulares los frames de la fuente de *stream* hasta que el usuario decide realizar una toma. En este punto la reproducción de la fuente es paralizada y se procede a analizar el frame que ha sido elegido como toma. Cuando éste es analizado se muestran los resultados obtenidos en la detección del patrón (tablero de ajedrez) y el usuario decide entre dar o no por buena la toma. Tras dar la respuesta el usuario puede continuar con la reproducción de los frames de la fuente de *stream* hasta que vuelva a elegir otra toma. Éste proceso se repite hasta que el calibrado finalice. También aparte de visualizar los frames de la fuente y los resultados de las tomas, se visualiza en todo momento un contador de las tomas que faltan por elegir para finalizar el proceso de calibrado. En la figura 5.16 se muestra el comportamiento del submódulo de reproducción del flujo de *stream* en el proceso de calibrado.

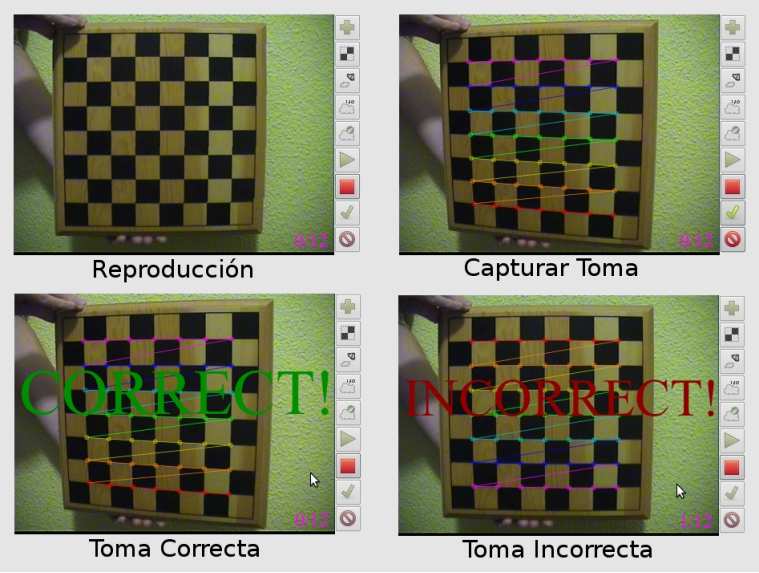

FIGURA 5.16: Submódulo de reproducción del flujo en el proceso de calibrado.

En el proceso de posicionamiento

En el proceso de posicionamiento se están reproduciendo constamente los frames que captura la fuente hasta que el usuario decide especificar las esquinas que representan la marca para posicionar la fuente con respecto a ésta. Tras señalar las cuatro esquinas, se realiza el proceso de posicionamiento y se muestran los resultados obtenidos para que el usuario detecte si hay algún error en los resultados. En la figura 5.17 se puede ver el comportamiento del submódulo para este proceso.

En el proceso de áreas de I/O

En el proceso de áreas de I/O se ha decidido representar el primer frame de lo que captura la fuente. Solamente se representa el primer frame, porque como la fuente ya ha sido posicionada la vista no va a cambiar. Sobre este primer frame, el usuario puede dibujar las zonas de entrada y salida. También puede seleccionarlas para poder eliminar la zona seleccionada debido a que no se ha dibujado correctamente el área o se ha equivocado. En el caso de que se decida cargar un archivo XML que contenga ya la definición de las áreas de I/O, después de realizar la carga se le visualiza al usuario el resultado. Los motivos por los que se visualiza los resultados de la carga del archivo XML es para que el usuario pueda comprobar que la definición de las áreas que incluye ese archivo XML son las correctas.

FIGURA 5.17: Submódulo de reproducción del flujo en el proceso de posicionamiento.

En el proceso de áreas fiables

En el proceso de áreas fiables el comportamiento del submódulo de reproducción es idéntico al de definición de áreas I/O. Un ejemplo del comportamiento del submódulo para este proceso se puede ver en la figura 5.18.

En el proceso de *tracking*

En el proceso de *tracking* se está reproduciendo constantemente los frames que va capturando la fuente. Cuando el proceso de segmentación detecta movimientos y son clasificados a priori por su aspecto en persona o vehículo se dibuja sobre el frame aquellos rectángulos que envuelvan al movimiento. Estos rectángulos se rellenan de un color *rojo* cuando su clasificación a priori es un *vehículo* y de color *verde* cuando es una *persona*. La clasificación que se muestra es particular a la propia fuente y solo tiene en cuenta la relación del aspecto. Luego esta clasificación será contrastada por las demás fuentes y por el histórico. Un ejemplo del comportamiento del submódulo en este proceso se puede ver en la figura 5.19.

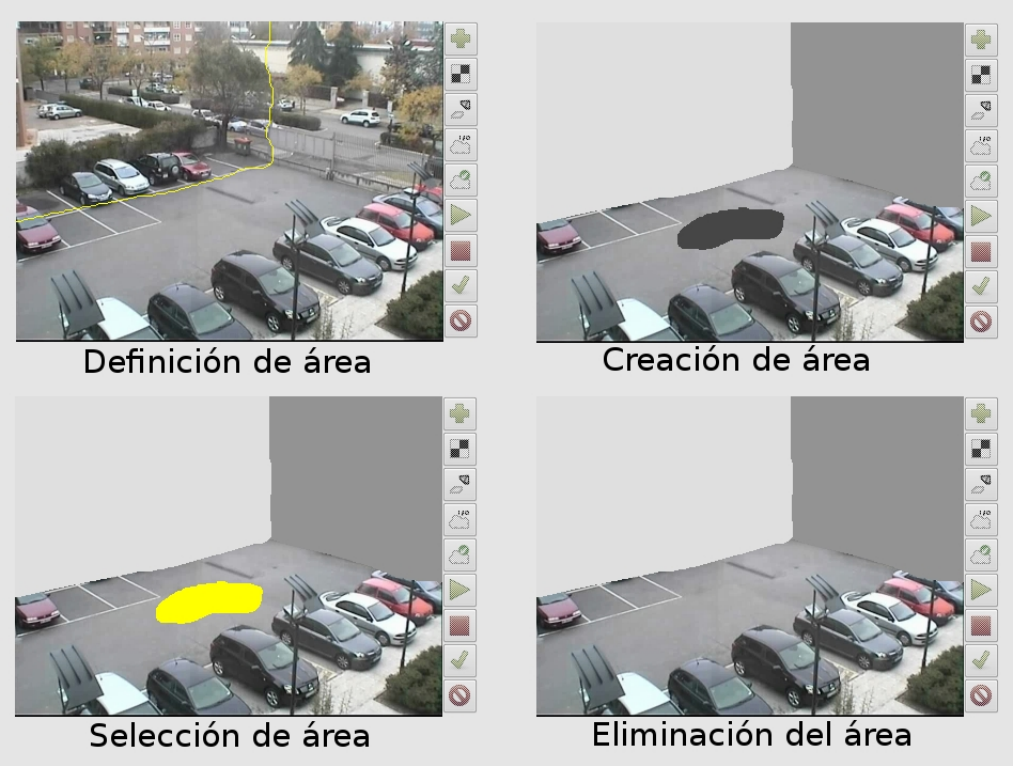

FIGURA 5.18: Submódulo de reproducción del flujo en el proceso de definición de áreas fiables.

5.4.2. Submódulo de entorno

El submódulo de entorno se encarga de la representación 3D del entorno que se está monitorizando. A parte de representar al entorno 3D permite que el usuario pueda interaccionar con la representación y cambiar la vista 3D realizando zoom, desplazamiento vertical u horizontal y rotaciones.

Para realizar las rotaciones de la vista 3D se pensó en activar dicha acción cuando sobre la vista 3D se pulse el botón izquierdo del ratón. Tras pulsar el botón se comienza a calcular los ángulos de rotación en función de la posición del ratón. La posición del ratón especifica dos dimensiones lo cual determina los dos ángulos de rotación en función de la diferencia entre la posición antigua del ratón y la nueva. Es decir, los ángulos de rotación son calculados siguiendo la siguiente ecuación:

$$angulo = angulo + (nueva\_posicion - antigua\_posicion) \tag{5.5}$$

Ahora bien si se atiende a los movimientos del ratón, cuando el usuario mueve el ratón de forma horizontal es porque quiere rotar el mundo sobre el eje Z, mientras que si el ratón

FIGURA 5.19: Submódulo de reproducción del flujo en el proceso de *tracking*.

se mueve de forma vertical es porque el usuario quiere rotar el mundo sobre el eje X como se puede ver en la figura 5.20.

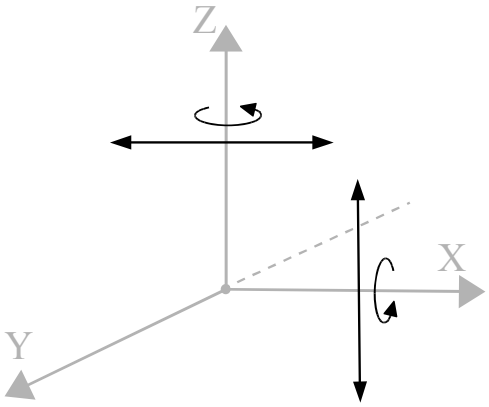

FIGURA 5.20: Movimientos del ratón para el cálculo de las rotaciones.

Por lo tanto mientras el usuario está moviendo el ratón manteniendo pulsado el botón izquierdo, la vista estará rotando sobre los ejes X y Z mediante las órdenes *glRotatef(ángulo_x, 1.0, 0.0, 0.0)* y *glRotatef(ángulo_z, 0.0, 0.0, 1.0)*. Inicialmente los ángulos de rotación sobre X e Z valen 0^o.

En cuanto al zoom y desplazamiento horizontal-vertical se pensó en realizarlo con la rueda del ratón. Si el usuario gira la rueda hacia adelante se está realizando un acercamiento y si lo realiza hacia atrás se produce un alejamiento. Apretando la rueda hacia dentro y sin soltarla el ratón es desplazado horizontalmente o verticalmente se produce un desplazamiento de la vista horizontal o vertical.

Para dar este efecto de zoom se parte de una posición inicial que es el zoom que tiene la vista inicialmente y por cada desplazamiento de la rueda hacia adelante o atrás

es incrementada o decrementada una unidad dicha distancia. Y para controlar que el zoom no vaya desde $(-\infty, \infty)$ se cheque que el resultado del incremento o decremento no supere ciertos límites, en el caso de que los supere dejarle el valor del límite. De esta forma el zoom que puede realizar el usuario está acotado en un intervalo $[min\_zoom,\ max\_zoom]$.

Y para dar el efecto de desplazamiento horizontal y vertical se hace de forma similar al efecto de rotación. Se utilizan dos variables para almacenar el desplazamiento en horizontal y en vertical que se tiene que aplicar sobre la vista. Dichas variables son actualizadas en función de la diferencia entre la última posición del ratón y la nueva de sus correspondientes coordenadas. Es decir, los desplazamientos son actualizados aplicando la siguiente ecuación:

$$\begin{aligned} desplа_x &= desplа_x + (nueva\_posicion_x - antigua\_posicion_x) * factor\_suavizado \\ desplа_y &= desplа_y + (nueva\_posicion_y - antigua\_posicion_y) * factor\_suavizado \end{aligned} \tag{5.6}$$

donde *factor_suavizado* es un número entre $[0,1]$ para hacer que el desplazamiento horizontal o vertical no sea muy brusco. Para un *factor_suavizado* de 0.01 el efecto de desplazamiento se consigue está bastante bien.

Una vez que se tiene calculado en las variables el zoom y el desplazamiento que el usuario quiere aplicar sobre la vista, se actualiza la vista mediante la orden *glTranslatef(*$desplа_x$*,* $desplа_y$*, distancia)*.

Suelo

Representa el suelo del entorno que se está monitorizando. Éste va a estar representado mediante un polígono de cuatro vértices cuyos lados van a tener una longitud de 60 metros.

Marcas

Representa las marcas que se utilizan en el proceso de posicionamiento de las fuentes. Dichas marcas son representadas del estilo que el suelo, mediante un polígono de cuatro vértices cuyas coordenadas serán las que el usuario especifique en el proceso de posicionamiento. Cada marca se dibuja sobre la vista 3D con el color asociado a dicha fuente, es decir, cuando finalice el proceso de posicionamiento la cámara virtual se dibujará del mismo color que el de la marca con la que se posiciono. Para la fuente 1 se le asigna un

color *rojo*, para la fuente 2 se le asigna un color *amarillo* y para la fuente 3 se le asigna un color *azul celeste*.

Cámara virtual

La cámara virtual representa la fuente que se ha desplegado en el entorno de monitorización. Para representar la cámara virtual se ha seguido un modelo similar al que utiliza *blender*. En la figura 5.21 se representan las dimensiones y forma de este modelo.

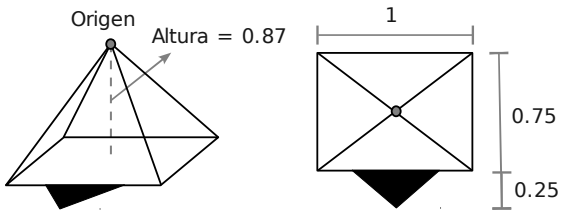

FIGURA 5.21: Especificación del modelo de una cámara virtual.

El módulo de gestión de agentes y el módulo de procesamiento se encarga de indicarle la matriz de transformación y el color asociados a la fuente que se acaba de posicionar. Por lo tanto los pasos del algoritmo 22 que se deben seguir para visualizar una cámara virtual son los siguientes:

1. Hacer una copia de la matriz de proyección actual de la vista con el fin de modificar esta copia y que no afecte a la vista 3D.
2. Multiplicamos la copia de la matriz por la matriz de transformación de la fuente. De esta forma se está posicionando y orientando el modelo de la cámara virtual cuando se dibuje en el siguiente paso.
3. Se asigna el color que tiene asociado dicha fuente.
4. Se dibuja el modelo de la cámara siguiendo las dimensiones especificadas en al figura 5.21.
5. Se elimina esa copia de la matriz de proyección porque ya se ha dibujado correctamente la cámara virtual.
6. Se elimina el color que se asignó dejando el color por defecto.

Algoritmo 22 Algoritmo para la visualización de una cámara virtual

Entrada: $Transformacion$: La matriz de transformación asociada a la cámara virtual.
$Color$: El color que va a tener la cámara virtual que se va a dibujar.

1: $glPushMatrix()$.
2: $glMultMatrixf(Transformacion)$.
3: $glColor4f(Color)$.
4: Dibujar el modelo de la cámara virtual.
5: $glPopMatrix()$.
6: $glColor4f(1.0, 1.0, 1.0, 1.0)$.

Objetos

Representa a cada uno de los objetos que están siendo detectados y seguidos por la aplicación. Los objetos son representados mediante una caja y un color. Los módulos de procesamiento y gestión de agentes le indican la posición, el ancho, el alto, el grosor y el color que tienen los objetos que se han detectado. Para el color de los objetos se ha pensado en utilizar un rango de *trece* colores con el fin de que éstos estén claramente diferenciados por sus tonalidades. Por lo tanto mientras en el entorno monitorizado no existan más de trece objetos diferentes éstos estarán claramente diferenciados. En el caso de que haya más de trece se comenzarán a repetir los colores. Los pasos del algoritmo 23 que se deben seguir para la visualización de un objeto son:

1. Se asigna el color que tiene asociado dicho objeto.
2. Se hace una copia de la matriz de proyección actual de la vista con el fin de modificar esta copia y que no afecte a la vista 3D.
3. Sobre la copia de la matriz de proyección se aplica un desplazamiento hacia la posición donde se encuentra el objeto detectado.
4. Sobre la matriz resultante del paso anterior se aplica un escalado proporcional a las dimensiones del objeto detectado.
5. Sobre la matriz resultante del paso anterior se dibuja un cubo estándar de dimensiones $(1 \times 1 \times 1)$.
6. Se elimina esa copia de la matriz de proyección porque ya se ha dibujado correctamente la caja.
7. Se elimina el color que se asignó dejando el color por defecto.

Algoritmo 23 Algoritmo para la visualización de los objetos detectados

Entrada: $Objeto$: El objeto que se va visualizar como una caja.
$Color$: El color que va a tener la caja que represente al objeto.

1: $glColor4f(Color)$.
2: $glPushMatrix()$.
3: $glTranslatef(Objeto.Posicion)$.
4: $glScalef(Objeto.Dimensiones)$.
5: Dibujar el modelo de la caja con dimensiones $1 \times 1 \times 1$.
6: $glPopMatrix()$.
7: $glColor4f(1.0, 1.0, 1.0, 1.0)$.

5.5. Módulo de exportación

El módulo de exportación se encarga de almacenar la información que se ha obtenido en los procesos de la aplicación. Los módulos de procesamiento y de gestión de agentes se encargan de comunicarse con el módulo de exportación para indicarle la información que debe exportar. Este módulo está formado por dos submódulos (*submódulo de exportación de archivos XML* y *submódulo de exportación del tracking*).

5.5.1. Submódulo de exportación de archivos XML

Este submódulo permite generar archivos XML donde se almacena la información referente a los procesos de calibrado, posicionamiento, áreas de I/O y áreas fiables. Para generar los archivos XML se ha utilizado la biblioteca *TinyXML*[3] y los formatos de los archivos expresados en la sección 5.1.2.

La forma de ir rellenando los archivos consiste en:

1. Crear un objeto de tipo *TiXmlDocument* el cuál va a representar la estructura del documento.

2. Para crear cada una de las etiquetas que componen la estructura del archivo, se crea un objeto de tipo *TiXmlElement*.

3. Cuando se tiene creada la etiqueta se le puede asociar el contenido que la etiqueta debe tener. Luego al objeto *TiXmlElement* que representa su etiqueta se le asocia mediante la operación *InsertEndChild* el contenido que se quiere que cuelgue de él. Según los formatos de los archivos algunas etiquetas solamente contienen un valor

[3] http://sourceforge.net/projects/tinyxml/

como es el caso de (*radial_distortions_K1*, *matrix_element_1_0*, *camera_center_x*, *X*, etc) él cuál es representado mediante un objeto *TiXmlText* que representa el valor como una cadena de texto para escribirlo. En el caso de que las etiquetas contengan como valor otras etiquetas, pues primero se deben crear aquellas etiquetas de las que no depende ninguna y teniendo esas creadas se van insertando en orden ascendente.

4. Después de tener la estructura del documento creada, ésta se debe asociar al documento. Para ello se aplica la operación *InsertEndChild* pero ahora sobre el objeto *TiXmlDocument*. De esta forma se añade al documento las estructuras que se van creando.

5. Cuando ya se tenga la estructura del archivo construida, el objeto *TiXmlDocument* es guardado en un archivo XML mediante la operación *SaveFile*.

Un ejemplo de aplicar el funcionamiento descrito anteriormente se puede ver en el algoritmo 24.

Algoritmo 24 Algoritmo de ejemplo de exportación en archivo XML

Entrada: *Valores* : Los valores de distorsión que se van a almacenar.
Ruta : La ruta donde el archivo se va a guardar en disco.

```
1: Crear: TiXmlDocument doc.
2: Crear etiqueta: TiXmlElement distorsion("distorsion").
3: para todo valor de Valores hacer
4:     Crear etiqueta: TiXmlElement elemento("Elemento").
5:     Crear el valor: TiXmlTextval(valor).
6:     Unir mediante: elemento.InsertEndChild(val).
7:     Añadir a distorsion: distorsion.InsertEndChild(elemento).
8: fin para
9: Añadir distorsion a doc mediante: doc.InsertEndChild(distorsion).
10: Guardar el archivo: doc.SaveFile(ruta).
```

5.5.2. Submódulo de exportación del *tracking*

Este submódulo es el encargado de exportar la información obtenida en el proceso de *tracking*. Dicha información se la proporcionan los módulos de gestión de agentes y de procesamiento. Por razones de eficiencia se ha considerado que la información se escriba en un archivo de texto, en lugar de un archivo XML. Luego la información de los objetos detectados es almacenada en el archivo de texto siguiendo el siguiente formato:

Idetificador#Fecha#Tiempo#Frame#Posición_x#Posición_y#Posición_-z#Alto#Ancho#Grosor#Tipo_clasificado#N_veces_persona#N_veces_vehículo

El algoritmo 25 describe el pseudocódigo del submódulo de exportación del *tracking* mediante el uso de las bibliotecas *iostream* y *fstream*.

Algoritmo 25 Algoritmo de exportación del *tracking*

Entrada: $Objetos$: La lista de objetos que han sido detectados en ese momento actual.
$Archivo$: El objeto del archivo ya abierto listo para poder escribir en él.

1: **para todo** obj de $Objetos$ **hacer**
2: Escribir: $Archivo.write(obj.Identificador + \# + obj.Fecha + \# + obj.Tiempo + \# + obj.Frame + \# + obj.Posicion\_x + \# + obj.Posicion\_y + \# + obj.Posicion\_z + \# + obj.Alto + \# + obj.Ancho + \# + obj.Grosor + \# + obj.Tipo\_clasificado + \# + obj.N\_veces\_persona + \# + obj.N\_veces\_vehiculo)$.
3: Escribir salto de linea.
4: **fin para**

5.6. Diagramas de clases de Diseño y Patrones

Una vez descrito los módulos que componen la aplicación en las secciones anteriores, aquí se indican los patrones que se han utilizado para el desarrollo de lo descrito anteriormente.

Como se especificó en la sección 4.1 la aplicación se ha desarrollado utilizando un *patrón arquitectónico multicapa*. Aplicando dicho patrón se consigue que los algoritmos que procesan la información queden separados de la parte encargada de visualizar e interaccionar con el usuario.

Para ayudarnos a la hora de asignar las responsabilidades que deben tener los objetos se ha seguido el *patrón experto*. El patrón experto indica que la asignación de responsabilidades debe recaer en el objeto que contiene la información, es decir, en el objeto experto. Luego la aplicación se ha diseñado asignando la responsabilidad de creación de objetos de otras clases a aquellas clases que conocen la información necesaria para crear dichos objetos.

Mientras se elaboraba el diseño de la aplicación, se presentó el problema de que algunas interfaces gráficas secundarias necesitaban la comunicación con la interfaz gráfica principal. Para evitar que se tuviesen diferentes puntos de acceso para comunicarse con la interfaz gráfica principal se decidió utilizar el *patrón singleton*. Con el uso de este patrón se proporciona un acceso global a dicha interfaz gráfica principal y se consigue que solamente haya una única instancia de ésta.

Según las estructuras y los algoritmos que se han descrito en las secciones anteriores se ha utilizado el *patrón iterator* para facilitar el acceso secuencial y abstraerse de la representación interna que tienen los diccionarios.

En el diseño del submódulo de *raytracer* se ha utilizado el *patrón polimorfismo*. Éste permite ser utilizado cuando los comportamientos de los objetos dependen de su tipo. Para evitar tener que estar preguntando por su tipo, se hace uso del *polimorfismo*.

Otro de los patrones utilizados es el *patrón adapter* para permitir el acceso a la información de una clase por medio de otra clase que adapta la forma de realizar la consulta con respecto a la clase que mantiene la información. Éste se ha utilizado para adaptar la consulta de los estados de los agentes cuando han pasado a la fase de *tracking* que es donde la gestión se hace más compleja.

Según la naturaleza de la aplicación, la interfaz gráfica principal aplica el *patrón facade* porque recoge la responsabilidad de atender las peticiones que va realizando el usuario

para delegarlas en las demás interfaces gráficas secundarias; y lo mismo ocurre con la clase *scene_camera* que es la clase principal de permitir la comunicación entre la capa de presentación y la capa de dominio delegando a su vez las responsabilidades en las clases de dominio dependiendo de la petición que le indique la capa de presentación.

El diagrama de clases de diseño de la capa de presentación se puede ver en la figura 5.22.

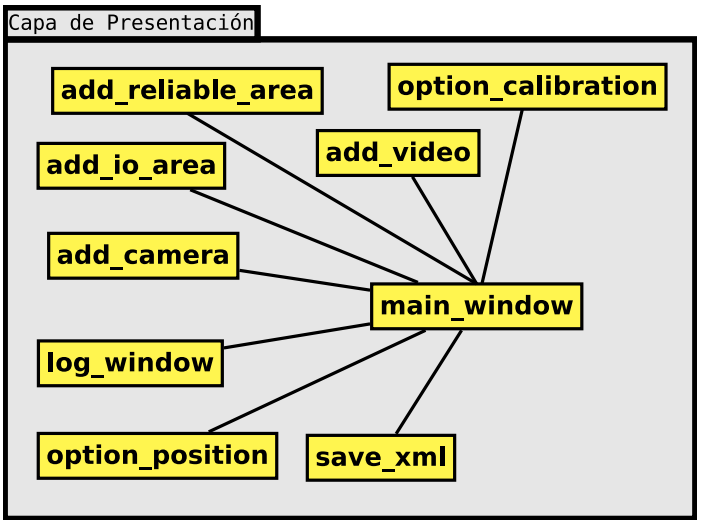

FIGURA 5.22: Diagrama de clases de diseño de la capa de presentación.

En la figura 5.23 se encuentra el diagrama de clases del paquete de *Raytracer*.

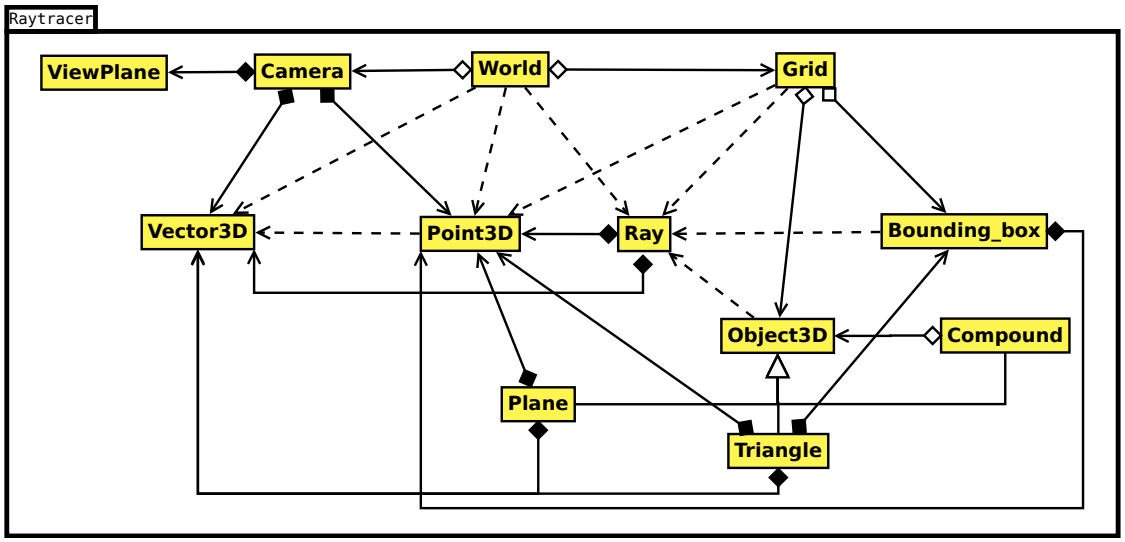

FIGURA 5.23: Diagrama de clases de diseño del paquete *raytracer*.

El diagrama de clases de diseño de la capa de dominio se encuentra en la figura 5.24.

Y en la figura 5.25 se encuentran las conexiones que existen entre la capa de dominio y la de presentación.

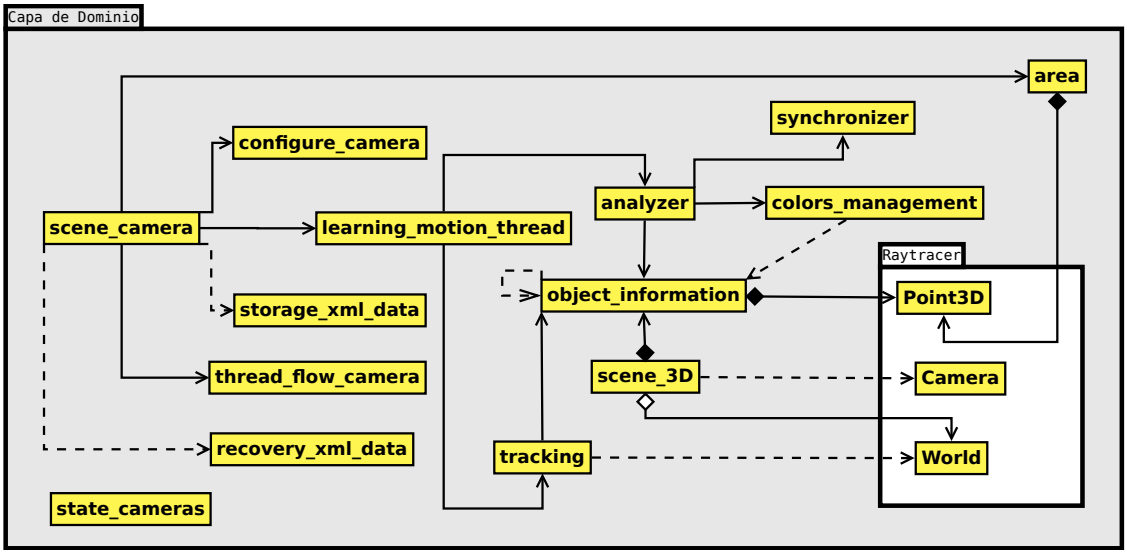

FIGURA 5.24: Diagrama de clases de diseño de la capa de dominio.

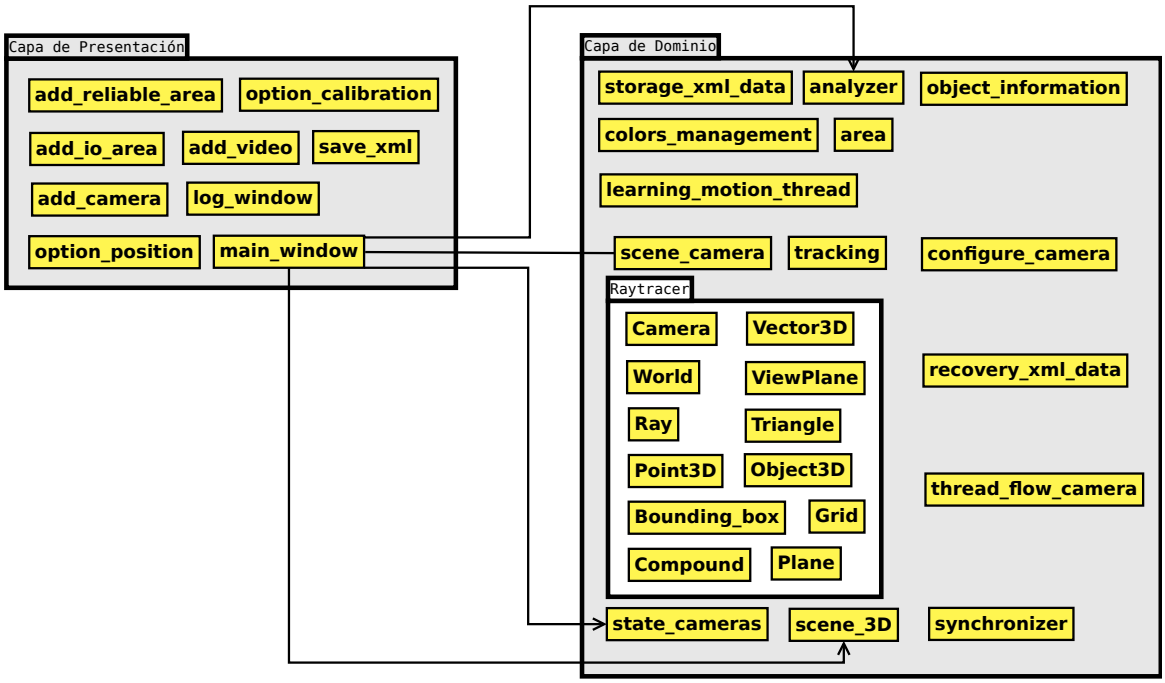

FIGURA 5.25: Diagrama de clases de diseño de las conexiones entre las capas.

6

Evolución y Resultados

En este capítulo se describen las iteraciones y la planificación que se ha seguido desde que se comenzó a desarrollar el proyecto. También se incluye una sección donde se evalúan los resultados obtenidos en cada uno de los procesos comparados con los referentes por el submódulo de depuración. En dicha sección de evaluación también se detallan los resultados obtenidos de probar el sistema sobre un entorno real.

6.1. Evolución

La idea de desarrollar *TraceMon* surgió el *20 de septiembre del 2010*. A partir de esa fecha se comenzó a perfilar en mayor detalle las funcionalidades que tenía que tener *TraceMon*. Dichos requisitos funcionales fueron expresados mediante diagramas de casos de uso (ver figura 4.2). Con estos diagramas de casos de uso se realizó un diseño preliminar (ver figura 4.3), a partir del cual se comenzó con las iteraciones del desarrollo de la aplicación.

6.1.1. Iteraciones

En este apartado se describen las iteraciones y los problemas que surgieron en cumplir con los hitos fijados en cada una de las iteraciones. El proceso de desarrollo de la aplicación en iteraciones comenzó el *4 de octubre del 2010*.

Iteración 1

El hito de la primera iteración consistió en realizar el diseño y modelado de un entorno 3D mediante *blender*. El motivo por el que modelar un entorno 3D era para poder depurar las funcionalidades (calibración, posicionamiento de las fuentes y *raytracer*) que iban a ser desarrolladas en las siguientes iteraciones. El entorno elegido a recrear con *blender* fue el *Paseo de la Universidad* incluida la rotonda y parte de la calle *Juan Ramón Jiménez*. Se comenzó a realizar el modelado urbanístico, aceras, suelo, farolas, señales, arbustos, rotonda, etc. Tras tener la infraestructura del entorno 3D se buscaron modelos de personas, vehículos y árboles con licencia *GPL* para poder utilizarlos. Los modelos de vehículos y árboles encontrados tenían demasiados detalles que hacían pesado el proceso de renderización. Por lo que se optó por crear un modelo de árbol con un cilindro como tronco y dos planos perpendiculares como la copa del árbol; y un modelo de vehículo a base de uniones de cajas. Una vez que se tenía modelado el entorno, se comenzó a aplicar las texturas a los objetos construidos. Toda esta primera iteración tuvo una duración de cinco semanas.

Iteración 2

El siguiente hito fue la construcción del submódulo del *raytracer*. La elaboración del *raytracer* representa la base en la que se apoyan los algoritmos de *tracking* y clasificación permitiendo conocer las posiciones y dimensiones de los objetos detectados. Para cumplir con este hito, se necesitaba conocer en detalle el funcionamiento de un *raytracer*. Después de conocer el funcionamiento del *raytracer* que se describió en la sección 3.4 se procedió a realizar un diagrama de diseño de éste y a implementar dicho diseño.

Tras tener una primera versión del *raytracer* al cabo de siete semanas se comenzó a realizar pruebas de rendimiento. El *raytracer* tenía que ser eficiente para dar soporte a que los algoritmos de *tracking* y clasificación puedan trabajar con flujo de vídeo en tiempo real. Después de comprobar que no era eficiente, se decidió desarrollar un algoritmo de aceleración basado en *grid* (ver sección 5.2.4). Para llevar a cabo su desarrollo se aplicaron modificaciones en el diagrama de diseño anterior y se procedió a implementarlo. Este proceso duró tres semanas.

Después de tener una versión con buen rendimiento del *raytracer* se procedió a crear con *blender* una escena simple y *scripts* de importación-exportación para probar que el funcionamiento del *raytracer* era correcto. Al cabo de una semana se obtuvieron los primeros resultados de aplicar el *raytracer* a la escena simple, y éstos no fueron correctos.

Por lo tanto se tuvo que realizar un proceso de depuración para detectar los errores que se habían cometido. Dicho proceso de depuración duró dos semanas hasta que por fin se detectaron los errores que se encontraban en la construcción de la base ortonormal y del plano vista.

Iteración 3

El hito asociado a la iteración 3 consistía en desarrollar el proceso de calibración de una cámara. El primer paso fue conocer el funcionamiento del proceso de calibración, el cual fue descrito en la sección 3.5.1 y teniendo el conocimiento de éste se procedió a implementarlo. Toda esta parte duró dos semanas. Cuando se tuvo la primera versión del proceso de calibración, se decidió probar el proceso mediante la creación de una escena de un tablero de ajedrez en *blender*. Y los primeros resultados obtenidos en el proceso de calibración no fueron correctos comparados con los que *blender* proporcionaba mediante la ejecución de un *script*. Por lo tanto comenzó un proceso de depuración para detectar el porqué los resultados obtenidos no eran correctos. Tras un periodo de dos semanas se descubrió que los errores provenían de no utilizar la función *cvFindCornerSubPix* para mejorar la precisión en la detección de las esquinas y que no se había tenido en cuenta el detalle de que *blender* y *openCv* mantienen diferentes sistemas de referencia.

Iteración 4

La iteración 4 consistió en el hito de desarrollar el proceso de posicionamiento de una cámara ya calibrada. Como en la iteración anterior se concluyó con el proceso de calibración funcionando correctamente, en esta iteración se iba a modificar dicho proceso de calibración para que se solapase con el proceso de posicionamiento. El comienzo de la iteración consistió en conocer los métodos que existen para averiguar la posición y orientación de la cámara. Tras conocer el funcionamiento se extendió la implementación del proceso de calibración con la implementación del proceso de posicionamiento. Aunque la última versión del proceso de posicionamiento se realiza con marcas de cuatro puntos, inicialmente este proceso de posicionamiento se realizó con tableros de ajedrez cuya información era reutilizada del proceso de calibración. Toda esta parte se realizó en un plazo de dos semanas.

Después de tener una primera versión del proceso de posicionamiento, se procedió a implementar un *script* para exportar la información de la cámara que tiene la escena realizada con *blender*. Y una vez más, los resultados obtenidos del proceso de posicionamiento con los esperados por *blender* no coincidian. Se realizó un proceso de

depuración de tres semanas para averiguar dónde podría estar el error. En las dos primeras semanas este proceso no encontró ningún error en la implementación teniendo en cuenta los detalles de que utilizan diferentes sistemas de referencia y la precisión. Pero en la tercera semana después de conocer que la implementación era correcta se descubrió que los resultados obtenidos en el proceso de posicionamiento eran expresados colocando el sistema de coordenadas en la propia cámara mientras que *blender* el sistema de coordenadas lo coloca en el propio tablero de ajedrez. Luego teniendo en cuenta este detalle en el plazo de una semana se implementaron los cambios del sistema de coordenadas y se concluyó con el proceso de posicionamiento funcionando correctamente.

Iteración 5

En esta iteración se pretendía cumplir con el hito del proceso de segmentación. Para ello el primer paso fue estudiar los diferentes métodos de segmentación que existen en la actualidad. El conocimiento de esos estudios puede verse descrito en la sección 3.5.2. Conociendo la variedad de los métodos de segmentación se decidió utilizar un algoritmo de *background* para la segmentación. Los motivos de la elección por este método se deben a que tiene un buen comportamiento en sistemas de vigilancia y *OpenCV* proporcionaba soporte para ello. Una vez que se tomó la decisión de utilizar el algoritmo de *codebook* para segmentar, se llevo a cabo la implementación del proceso de segmentación haciendo uso de dicho algoritmo. Tras tener una versión del proceso de segmentación se probó su comportamiento con vídeos generados del entorno 3D que se modelo en la iteración 1. Toda esta iteración duró aproximadamente cuatro semanas.

Iteración 6

Una vez que se realizaron las anteriores iteraciones, se necesitaba una interfaz gráfica para la aplicación. Luego para esta iteración surgió el hito de realizar las interfaces gráficas que iba a presentar la aplicación. Para llevar a cabo este hito, se comenzó realizando un diagrama de clases de diseño donde se recogieran los detalles de la comunicación entre las diferentes interfaces gráficas que iba a tener la aplicación. Y partiendo de este diagrama se realizaron los diseños de las interfaces gráficas mediante la herramienta *Glade*. Después de tener los diseños de las interfaces gráficas con *Glade* se procedió a implementar la lógica correspondiente al control de los eventos de botones, comunicación con otras interfaces, las operaciones de zoom, desplazamiento y rotación sobre la vista 3D, etc. Al cabo de cuatro semanas finalizó esta iteración.

Iteración 7

La iteración 7 tuvo como hito el desarrollar la estructura para la gestión de los agentes. En las anteriores iteraciones se habían desarrollado partes de la aplicación y antes de poder diseñar como llevar a cabo la gestión de los agentes se tenía que unificar y conectar el trabajo realizado en las anteriores iteraciones. Dicha conexión se realizó primero conectando los diagramas de diseño y posteriormente la implementación. Al conectar el proceso de calibración con las interfaces gráficas correspondientes éste se tuvo que adaptar de forma que el usuario pudiese interactuar con el proceso para indicarle los tableros que eran correctos. A la hora de conectar el proceso de posicionamiento, es en este punto donde el proceso se modifica para que el posicionamiento se realice mediante marcas en lugar de tableros de ajedrez y el usuario pueda especificar las coordenadas que definen la marca.

Tras conectar el trabajo de las iteraciones anteriores, se empieza a diseñar y a pensar en la gestión de los agentes. Se elaboró un diagrama de clases para especificar las clases que recogen sobre los estados de los agentes, la información compartida, etc. Una vez que se tuvo este diagrama se llevo a cabo la implementación. En la fase de implementación de esta iteración surgió el problema de que los agentes encargados de los procesos de calibración, posicionamiento, etc no podían actualizar la interfaz gráfica principal con los resultados que se iban obteniendo de la ejecución de esos procesos. El motivo era porque *GTK* solo permite que el proceso principal sobre el que se lanzó la interfaz gráfica principal pueda actualizarla. Después de encontrarse con esta limitación, se realizaron los cambios en el diseño e implementación, hasta que se tuvo una versión robusta sobre la gestión de los agentes en las fases de calibrado, posicionamiento y segmentación. La duración de esta iteración fue de aproximadamente seis semanas y supuso un punto de inflexión importante en el desarrollo de las demás funcionalidades.

Iteración 8

El hito asociado a la iteración 8 fue la definición de áreas fiables y de I/O. Esta iteración consistió en diseñar la parte del diagrama de clases asociado a esta funcionalidad. Y una vez actualizado el diagrama de clases se pasó a la implementación. En la implementación el primero paso consistió en como permitir al usuario dibujar de forma cómoda las áreas. Después de tener implementada dicha funcionalidad, se procedió a pensar e implementar la forma de construir las máscaras para luego aplicar en el proceso de *tracking*. Toda la iteración fue realizada en el plazo de una semana aproximadamente.

Iteración 9

Después de tener los procesos de calibrado, posicionamiento y definición de áreas funcionando correctamente, comenzó la iteración 9. El hito a cumplir en esta iteración consistía en realizar la funcionalidad de importación y exportación de la información de los procesos de calibrado, posicionamiento y definición de áreas de I/O y fiables. De esta forma en las siguientes iteraciones no se tendrían que estar repitiendo constantemente los procesos de calibrado, posicionamiento, etc, para llegar a probar el funcionamiento del *tracking*. Dicha iteración comenzó con el diseño de los diagramas para desempeñar la funcionalidad de importación y exportación, y tras esos diagramas se procedía a la implementación. Después de tener implementada la funcionalidad, se probó su funcionamiento analizando si la información que era importada y luego exportada coincidía. De esta forma se conseguía probar que el procesamiento de la importación y de la exportación era correcto. El desarrollo de esta iteración se realizó en el plazo de una semana.

Iteración 10

La iteración 10 consistió en realizar los hitos sobre los algoritmos de clasificación y de fusión de la información de las fuentes. Para cumplir con el hito del algoritmo de clasificación se realizó un estudio sobre las diferentes técnicas de clasificación utilizadas actualmente (ver sección 3.5.3). Después de conocer dichas técnicas se pensó el algoritmo empleado para clasificar y se llevó a cabo su implementación. El plazo de tiempo para cumplir este hito fue de una semana. Una vez resuelto el hito anterior, se inició el hito de fusión de la información. Para ello se comenzó diseñando el algoritmo de fusión de la información basándose en la *función euclidea*. Este algoritmo consistía en buscar los mejores candidatos para fusionarse por ser considerados los mismos objetos vistos por distintas fuentes. Una vez diseñado el algoritmo sobre papel se llevo a cabo la implementación. Este hito fue resuelto al cabo de una semana.

Iteración 11

Después de tener la parte de clasificación y de fusión de la información de las fuentes, en esta iteración se tuvo como hito el desarrollo de la parte de seguimiento e identificación de los objetos. Este hito es muy importante para que la aplicación permita identificar los objetos detectados con los que antiguamente fueron detectados. Para ello se realizó un estudio de las situaciones que se podrían dar y se diseñó los algoritmos y estructuras necesarias para llevar a cabo la identificación. Después de tener el algoritmo diseñado se

realizó la implementación de éste. El periodo de duración para cumplir con este hito de dos semanas.

Con el prototipo de la aplicación realizado se pensó en un último hito. Este hito consistía en el despliegue de la aplicación sobre un entorno real para analizar y estudiar su comportamiento. El entorno sobre el que se desplegó la aplicación fue los aparcamientos del edificio *Fermín Caballero*. Tras realizar el despliegue de la aplicación, se analizaron los resultados y se hicieron pequeños ajustes y retoques en los umbrales de los algoritmos implementados en las anteriores iteraciones. El plazo de tiempo asociado a este hito fue de dos semanas.

6.1.2. Recursos y costes

Haciendo un recuento del tiempo empleado en el desarrollo del proyecto se han necesitado 44 semanas. Esto supone la contratación de recursos humanos para un año de trabajo. Por lo tanto en la figura 6.1 se muestran los recursos utilizados en el desarrollo del proyecto junto al coste total para la elaboración de éste. Las estadísticas sobre el número de líneas de código, archivos fuentes, comentarios, etc se encuentran reflejados en la figura 6.2.

Recurso	Cantidad	Coste (euros/unidad)
Recursos humanos	1	21600
Cámara Logitech AF-Sphere	1	117,16
Cámara Axis 207W	1	259,32
Cámara Axis 213r	1	980,89
Ordenador portatil	1	609,13
		23566,5

FIGURA 6.1: Costes del proyecto.

Tipo de Archivo	Nº Archivos	Líneas en blanco	Líneas de Comentarios	Líneas de Código	Total líneas
C++	36	1372	339	**7631**	9266
Cabecera C++	37	1022	3936	**1177**	6331
Makefile	1	43	3	**89**	135
Scripts	4	18	1	**109**	128
Total	78	2455	4279	**9006**	15860

FIGURA 6.2: Estadísticas del código fuente.

6.2. Resultados

En esta sección se describen los resultados obtenidos de realizar pruebas con entornos virtuales y reales. También se realiza un estudio del rendimiento de la aplicación cuando a ésta se le conectan diferente número de fuentes.

6.2.1. Resultados con entorno virtual

Antes de describir los resultados de *tracking* obtenidos con un entorno virtual, se van a describir los resultados obtenidos de aplicar la depuración en los procesos de calibración, posicionamiento y *raytracer*.

Una comparación de los resultados obtenidos de la calibración de una cámara virtual con once tomas de tableros se puede ver en la figura 6.3. En dicha figura se pueden ver como los coeficientes de distorsión de una cámara virtual de *blender* son 0 y *opencv* presenta valores próximos a 0. Y como el error cometido en los parámetros de la distancia focal y punto central de la imagen de la cámara no superan un par de unidades.

En la figura 6.4 se puede ver como el error medio disminuye con el aumento del número de tomas de tableros de ajedrez.

Una comparativa de los resultados obtenidos en la depuración del proceso de posicionamiento de la cámara con respecto a un tablero de ajedrez se puede ver en la figura 6.5. Dicha cámara es definida por su matriz de rotación y posición cuyos valores se indican en la columna de *Blender*. El error cometido por el algoritmo del cálculo de estos parámetros es $\frac{0.064}{12} = 0.0053$.

	Blender	**OpenCV**
K1	0	-0.032143
K2	0	0.466096
K3	0	-1.747289
P1	0	0.000482
P2	0	0.000802
Fx	875.0	876.803223
Fy	875.0	873.463257
Cx	400.0	400.051880
Cy	300.0	299.016876

FIGURA 6.3: Comparativa del resultado de calibración para once tomas.

	Error
12 Tomas	2.18/9 = 0.24
16 Tomas	1.80/9 = 0.2
20 Tomas	1.67/9 = 0.18

FIGURA 6.4: Comparativa del error en calibración para diferente número de tomas.

	Blender	**OpenCV**
rotación1_1	0.999986	0.999978
rotación1_2	-0.001373	-0.001663
rotación1_3	0.005113	0.006387
rotación2_1	-0.000018	-0.000066
rotación2_2	0.964881	0.965152
rotación2_3	0.262688	0.261689
rotación3_1	-0.005293	-0.006600
rotación3_2	-0.262684	-0.261684
rotación3_3	0.964867	0.965131
posición_x	3.515390	3.501923
posición_y	-6.220792	-6.198534
posición_z	12.535859	12.558505

FIGURA 6.5: Comparativa del resultado de posicionamiento para tablero de ajedrez.

En cuanto a los resultados obtenidos en el proceso de depuración del *raytracer* se pueden ver en la figura 6.6. Estos resultados son correctos porque los rayos inciden perfectamente en los lugares exactos de las caras de los objetos, teniendo en cuenta la proyección y los límites de la resolución que tiene la cámara.

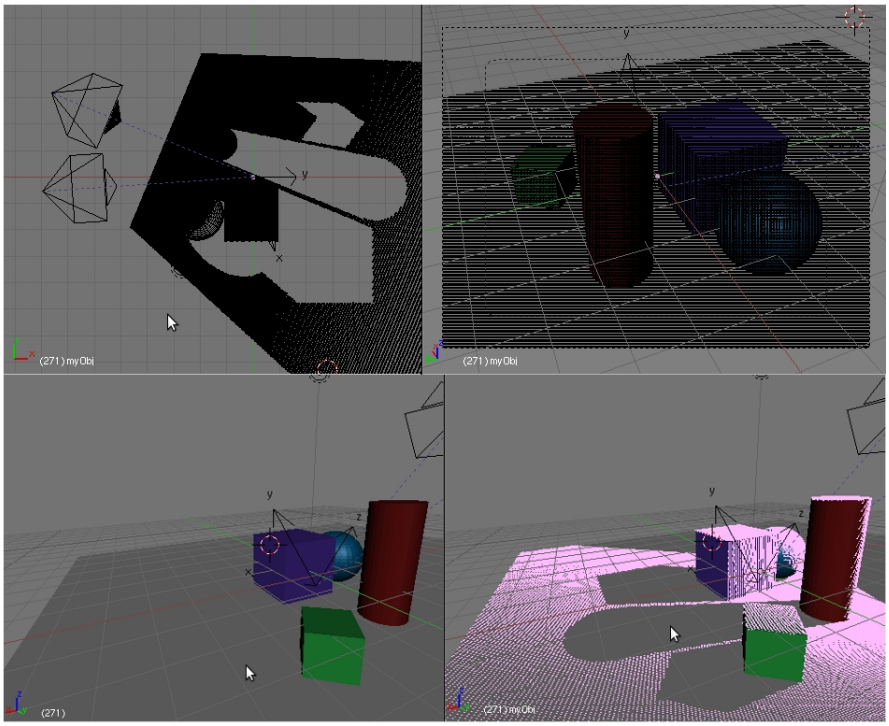

FIGURA 6.6: Comparativa del resultado de posicionamiento para tablero de ajedrez.

Una vez expresados los resultados obtenidos en los procesos de depuración del calibrado, posicionamiento y *raytracer* se detallan los resultados obtenidos de aplicar los vídeos sobre el entorno 3D del *Paseo de la Universidad*.

El error cometido en el proceso de calibración de las cámaras desplegadas en ese entorno virtual es $\frac{1.419}{9} = 0.157$. En cuanto los resultados obtenidos del posicionamiento con respecto a una marca se pueden ver comparados en la figura 6.7, donde el error cometido en la estimación de la posición y proyección de la cámara 1 es $\frac{2.697}{12} = 0.224$, para la cámara 2 es $\frac{2.665}{12} = 0.222$ y para la cámara 3 es $\frac{1.789}{12} = 0.149$. Aunque el error que se comete utilizando marcas con respecto a tableros de ajedrez ha aumentado, éste sigue considerándose bastante bueno.

El vídeo virtual recoge la situación de un primer peatón cruzando por el paso de cebra. Después un segundo peatón que cruza por el paso de cebra, mientras un coche que se dispone a hacer un cambio de sentido en la rotonda espera a que pase dicho peatón.

	Blender Cámara 1	OpenCV Cámara 1	Blender Cámara 2	OpenCV Cámara 2	Blender Cámara 3	OpenCV Cámara 3
rotación1_1	0,998821	0,999084	-0,094492	-0,085396	-0,234749	-0,221595
rotación1_2	0,048428	0,041482	-0,995520	-0,996328	0,971996	0,975137
rotación1_3	0,003365	0,010533	-0,003357	-0,006111	-0,010817	0,008950
rotación2_1	-0,024795	-0,018148	0,581524	0,574406	-0,435101	-0,432569
rotación2_2	0,449327	0,425966	-0,057933	-0,054243	-0,095119	-0,100091
rotación2_3	0,893023	0,904703	0,811464	0,816772	0,895343	0,896028
rotación3_1	0,041735	0,042016	-0,808023	-0,814104	0,869241	0,873945
rotación3_2	-0,892054	-0,903788	0,074725	0,066239	0,214887	0,207712
rotación3_3	0,449998	0,425913	0,584393	0,576929	0,445245	0,443994
posición_x	-9,982409	-9,045878	-150,167786	-149,943359	130,293213	131,109542
posición_y	-188,679016	-189,337134	45,242146	43,435883	20,631472	19,939838
posición_z	84,787736	83,776831	97,864579	97,280536	65,289048	65,530815

FIGURA 6.7: Comparativa del resultado de posicionamiento para las cámaras del entorno virtual.

Los resultados que TraceMon obtiene de analizar el vídeo del entorno virtual se pueden ver en las figuras (6.8, 6.9, 6.10). Éstos son orientativos y no tiene porque comportarse TraceMon ofreciendo esas cifras ante cualquier otra situación.

Para obtener estos resultados, se han ido ajustando la definición de las áreas fiables y los umbrales de distancia que utilizan los algoritmos para mejorar su probabilidad de acierto. En la figura 6.8 se muestra el comportamiento del algoritmo de segmentación donde se recogen los fallos y los aciertos. En cuanto a los fallos, este algoritmo puede cometer tres tipos de fallos:

- *Objeto ignorado por sistema*: Esta situación se da cuando existe un objeto en movimiento pero éste no es detectado por el algoritmo de segmentación.
- *Ruido*: Esta situación se da cuando el algoritmo de segmentación introduce ruido a la hora de detectar los movimientos. Por ejemplo detecta objetos estáticos que forman parte del entorno como árboles, o divide el movimiento de un objeto en dos, etc.
- *Objeto ignorado por zona*: Esta situación se da cuando el objeto ha sido ignorado por pertenecer a una zona de fiabilidad 0 o zona donde la visibilidad del objeto produciría ruido en el sistema. Esta situación se considera fallo a nivel de cámara. Y a nivel de multicámara se considerará fallo cuando todas las cámaras en el mismo momento produzcan un fallo de este tipo dando como resultado a la perdida de ese movimiento.

Se puede apreciar en la figura 6.8 como los problemas de segmentación que existen a nivel de cámara se van corrigiendo cuando se mezcla la información con la del resto de fuentes. Esto se debe a que se refuerza dicha segmentación reduciendo los objetos ignorados por zonas.

Vídeos	**Segmentación**				**Porcentaje**	
Entorno real	*Acierto*	*Fallo*			*Acierto*	*Fallo*
		Objeto Ignorado por Sistema	*Ruido*	*Objeto Ignorado por zona*		
Cámara 1	71	0	0	21	77,17%	22,83%
Cámara 2	45	0	0	47	48,91%	51,09%
Cámara 3	44	9	0	39	47,82%	52,18%
Multicámara	92	0	0	0	100,00%	0,00%

FIGURA 6.8: Análisis de los resultados obtenidos por el algoritmo de segmentación con el entorno virtual.

En la figura 6.9 se detalla el comportamiento del algoritmo de clasificación para el entorno virtual. En ella se representan los aciertos y errores cometidos en la clasificación tanto a nivel de cámara como multicámara. También se ha especificado una columna donde se recogen los errores cometidos en el algoritmo de segmentación para no tenerlos en cuenta en el análisis del comportamiento del algoritmo de clasificación.

Principalmente el origen de estos errores se debe a que el objeto es detectado parcialmente debido a que está entrando en la vista de la cámara. Dicho error se reduce cuando se pone en común la información de la clasificación con las demás cámaras y se corrige aún mucho más dejándolo a 0, cuando éste es contrastado con la información del historial de clasificación que ha ido teniendo el objeto identificado. Se puede apreciar como el porcentaje de aciertos en multicámara es menor que el de algunas de las cámaras. La razón por lo que se da esto, es porque el error cometido a nivel de cámara se puede propagar a nivel de multicámara cuando la mayoría o en caso de empate afirme dicho error. El error se puede propagar por los siguientes casos:

- Una fuente comete el error en la clasificación de un objeto, y el resto de fuentes no perciben o ignoran dicho movimiento.
- La mayoría de las fuentes confirma el error.

- En caso de empate en el resultado, porque el número de fuentes que han clasificado ese movimiento es par, siempre se decanta en clasificarlo como *persona*, luego se produciría error si no fuese de este tipo.

Vídeos	**Clasificación Persona**		**Clasificación Vehículo**		**Objetos Ignorados y Ruido**	**Porcentaje**	
Entorno real	*Acierto*	*Fallo*	*Acierto*	*Fallo*	*No clasificados*	*Acierto*	*Fallo*
Cámara 1	36	0	33	2	21	97,18%	2,82%
Cámara 2	26	0	19	0	47	100,00%	0,00%
Cámara 3	31	0	12	1	48	97,72%	2,28%
Multicámara	54	0	36	2	0	97,82%	2,18%
Análisis temporal	54	0	38	0	0	100,00%	0,00%

FIGURA 6.9: Análisis de los resultados obtenidos por el algoritmo de clasificación con el entorno virtual.

Y por último en la figura 6.10 se encuentra el comportamiento del algoritmo de *tracking*. Dicho algoritmo presenta una tasa de fallos 0, esto es porque la efectividad en asignar correctamente la identificación de un movimiento detectado depende de la precisión en que fue reconocido por el algoritmo de segmentación. Al estar trabajando con un entorno virtual, dicha precisión es muy buena y la calidad de imagen también, luego se comete muy poco error en el posicionamiento de los objetos detectados. Y esto hace que los resultados del seguimiento de los objetos para el entorno virtual tenga esos porcentajes.

Vídeos	**Tracking**		**Porcentaje**	
Entorno real	*Acierto*	*Fallo*	*Acierto*	*Fallo*
Multicámara	92	0	100,00%	0,00%

FIGURA 6.10: Análisis de los resultados obtenidos por el algoritmo de *tracking* con el entorno virtual.

6.2.2. Resultados con entorno real

Para completar las pruebas de la aplicación, el sistema se desplegó sobre los aparcamientos del edificio *Fermín Caballero* de la Escuela Superior de Informática de Ciudad Real. Se utilizó como marca las dimensiones de un aparcamiento que miden 2.6×4.2 $metros$ y una vez posicionadas y definidas las regiones de entrada y salida se procedió a realizar el seguimiento. El análisis del comportamiento de TraceMon frente a las situaciones que se dieron en el entorno real se detallan en las figuras (6.11, 6.12, 6.13). El significado de las etiquetas que presentan las tablas se recogen en el apartado de *resultados con entorno virtual*.

Si se analizan los resultados de segmentación de la figura 6.11 se pueden ver que los errores de *Objeto ignorado* se reducen cuando se combina la información de las cámaras, pero a su vez el fusionar la información de las cámaras introduce mayor error de *ruido*. Esto se debe a que a diferencia del entorno virtual, la precisión que se obtiene en el posicionamiento de las fuentes es peor. Y esto obliga a que cuando se pone en común la información de las fuentes para las situaciones donde los objetos estén muy próximos, ese error en el posicionamiento obliga a dar una segmentación multicámara incorrecta aunque a nivel de cámara fue correcta. También se puede ver como el número de objetos ignorados por zona es bastante elevado. Esto es debido a que las vistas del entorno monitorizado presentan muchas zonas en donde se producen oclusiones parciales, o baja visibilidad haciendo que si no son ignoradas introduzcan ruido a la aplicación. Para evitar esas zonas donde se dan oclusiones parciales o la información que se obtiene en ellas es incorrecta se definen como zonas de baja o 0 fiabilidad. Esto produce poco porcentaje de acierto a nivel de cámara pero mejora los porcentajes de acierto a nivel multicámara, y se consigue que se reduzcan los problemas derivados de las oclusiones parciales.

Vídeos	**Segmentación**				**Porcentaje**	
Entorno real	*Acierto*	*Fallo*			*Acierto*	*Fallo*
		Objeto Ignorado por Sistema	*Ruido*	*Objeto Ignorado por zona*		
Cámara 1	140	2	1	43	75,26%	24,74%
Cámara 2	104	16	1	65	55,91%	44,09%
Cámara 3	79	18	0	89	42,47%	57,53%
Multicámara	177	0	9	0	95,16%	4,84%

FIGURA 6.11: Análisis de los resultados obtenidos por el algoritmo de segmentación con el entorno real.

En cuanto a la figura 6.12 se encuentran los resultados obtenidos de la clasificación. Éstos presentan un comportamiento similar al del entorno virtual, donde el error se va reduciendo cuando se mezcla la información de las cámaras y desaparece cuando éste es consultado con el historial que mantiene la clasificación temporal que ha tenido la vida de ese objeto. Se puede ver que el porcentaje de acierto de la clasificación multicámara es peor que el de algunas cámaras individuales, esto se debe a que la cámara 1 clasifica objetos de forma incorrecta cuando las demás cámaras su algoritmo de segmentación lo ignoraron de esta forma se propaga dicho error al nivel multicámara. Las tres situaciones que se pueden dar para propagar el error al nivel multicámara se encuentran descritas en el anterior apartado.

Vídeos	**Clasificación Persona**		**Clasificación Vehículo**		**Objetos Ignorados y Ruido**	**Porcentaje**	
Entorno real	*Acierto*	*Fallo*	*Acierto*	*Fallo*	*No clasificados*	*Acierto*	*Fallo*
Cámara 1	96	1	27	16	46	87,85%	12,15%
Cámara 2	69	0	34	1	82	99,00%	1,00%
Cámara 3	52	0	26	1	107	98,73%	1,27%
Multicámara	130	1	42	4	9	97,17%	2,83%
Análisis temporal	131	0	46	0	9	100,00%	0,00%

FIGURA 6.12: Análisis de los resultados obtenidos por el algoritmo de clasificación con el entorno real.

Por último los resultados obtenidos en la asignación de los identificadores (*tracking*) se puede ver en la figura 6.12 como se tiene una tasa de error que no se tenía en el entorno virtual. Esto se debe a que el algoritmo basado en la asignación de identificadores depende de la precisión que tenga el algoritmo de segmentación y de la precisión en la posición de los objetos detectados. Dicha precisión se reduce cuando se está trabajando con entornos reales, haciendo que se produzcan errores en la asignación de identificadores de objetos antiguos a los nuevos detectados. La forma que tiene el sistema de resolver las situaciones de objetos que se juntan y se separan es mediante la reasignación de los identificadores. Todo esto se basa en analizar los objetos que había antiguamente y los nuevos detectados. Si resulta que en la detección de los objetos nuevos se ha introducido *ruido*, esto provocará que el sistema intuya que algún objeto se ha separado (si no se encontraban en zona de entrada/salida), o que ha entrado un objeto nuevo (si el objeto que introduce el rudio se encuentra en zona de entrada o entrada/salida). Si se introducen en la fase de segmentación errores de *objetos ignorados* el sistema puede interpretar que ha habido una unión, siempre que los

objetos estuviesen próximos o dar por perdida la referencia de dicho objeto y reasignar identificadores. En cuanto a la detección de salida de objetos para las pruebas realizadas se comporta correctamente, debido a que el algoritmo para detectar estas situaciones analiza cuando un objeto pasa de estar en una zona a neutra a zona de salida o entrada/salida.

Vídeos	**Tracking**		**Porcentaje**	
Entorno real	*Acierto*	*Fallo*	*Acierto*	*Fallo*
Multicámara	177	9	95,16%	4,84%

FIGURA 6.13: Análisis de los resultados obtenidos por el algoritmo de *tracking* con el entorno real.

6.2.3. Resultados de rendimiento

La técnica utilizada para medir el rendimiento de las partes de TraceMon es *profiling*. Dicha técnica consiste en medir el rendimiento de la aplicación para una determinada ejecución. La herramienta utilizada para llevar a cabo este tipo de análisis ha sido *gprof*. Su uso es muy simple consiste en compilar el código fuente añadiendo el flag *-pg*. Y una vez generado el ejecutable basta con ejecutarlo. Si la ejecución de la aplicación acabó correctamente se generará un archivo llamado *gmon.out*. Para generar el informe basta con ejecutar en el terminal el siguiente comando:

```
1  > gprof ejecutable
```

Tras realizar varios informes ante diferentes situaciones de la aplicación, en las figuras (6.14, 6.15 y 6.16) se puede ver como varía el porcentaje de uso de la CPU para los siguientes componentes:

- *Algoritmo de clasificación*. El consumo de CPU asociado a este componente incluye el coste computacional de invocar al raytracer para obtener la posición y dimensiones para cada uno de los objetos que se han detectado para posteriormente analizar la relación entre sus dimensiones.
- *Algoritmo de fusión*. Este componente representa el consumo de llevar a cabo la fusión o mezcla de la información proveniente de las cámaras. Como es de esperar, cuanto mayor sea el número de fuentes a mezclar la información mayor será su consumo.

- *Algoritmo de segmentación*. El consumo de CPU asociado a este componente incluye el coste empleado en separar los objetos del fondo (los elementos estáticos que existen en el entorno).
- *Otros*. En este componente se encuentra el consumo de CPU del resto de módulos (visualización, exportación, sincronización, gestión, etc) que forman la aplicación.

En cuanto al análisis de los resultados de los diagramas de sector, se puede ver como el uso de CPU aumenta con el número de fuentes. Esto es porque se requiere de más agentes que se encarguen de esas tareas, lo cual conlleva a invertir más tiempo de CPU.

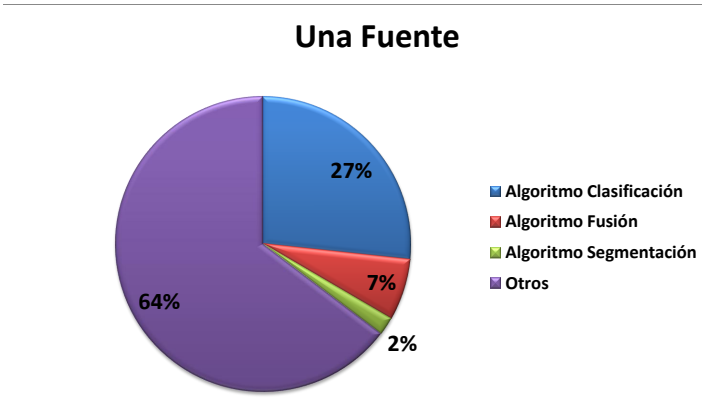

FIGURA 6.14: Resultados de rendimiento de los algoritmos de TraceMon con una fuente.

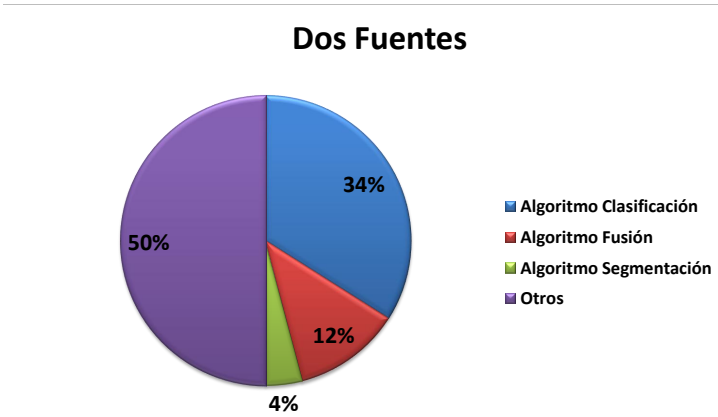

FIGURA 6.15: Resultados de rendimiento de los algoritmos de TraceMon con dos fuentes.

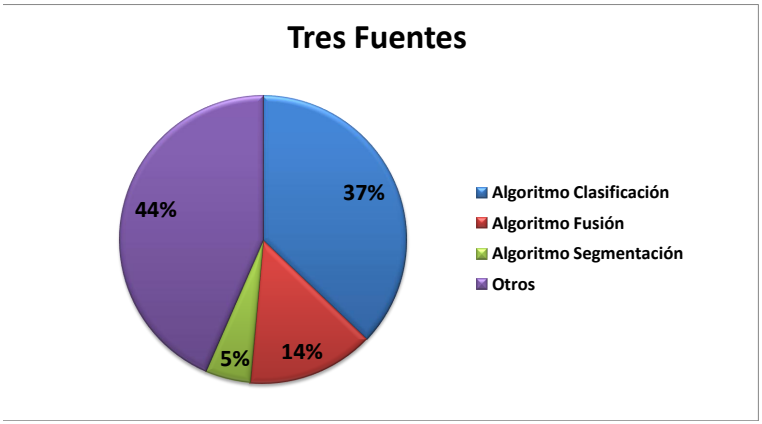

FIGURA 6.16: Resultados de rendimiento de los algoritmos de TraceMon con tres fuentes.

7

Conclusiones y propuestas

Los sistemas multicámara de vigilancia están diseñados para realizar la detección, clasificación y seguimiento de los objetos que se mueven en el entorno monitorizado. Para desarrollar las capacidades de detección, clasificación y seguimiento existen multitud de algoritmos que ofrecen mejor o peor comportamiento ante determinadas situaciones. Por ejemplo en el proceso de detección no siempre es fácil detectar los movimientos porque se suelen producir oclusiones, hay una mala distinción del objeto debido a que se fusiona con el entorno, efectos atmosféricos, etc.

También estos algoritmos de detección, clasificación y seguimiento están orientados a trabajar a nivel individual sin contrastar la información obtenida con la de otras cámaras. Y este aspecto de fusionar la información con la información de otras cámaras es muy útil para reducir los fallos cometidos a nivel individual y obtener un sistema con mejor comportamiento. Por esta razón, los sistemas multicámara están abriendo camino en el campo de la videovigilancia.

En este capítulo se describen los objetivos que se han alcanzado en el desarrollo de un sistema de *tracking* multicámara (TraceMon), junto con una serie de propuestas de trabajo futuro para extender y mejorar la funcionalidad de TraceMon.

7.1. Objetivos alcanzados

Los objetivos planteados en el capítulo 2 han sido satisfechos de la siguiente forma:

- **Soporte para la gestión de fuentes de vídeo heterogéneas**. Para cumplir este objetivo y poder abstraerse de las diferencias que existen en el acceso a los distintos tipos de fuentes (archivos de vídeo, cámaras locales y remotas) se ha utilizado las funciones que ofrece *OpenCV*. *OpenCV* da soporte para conectar a estos tres tipos de fuentes y generar un objeto *CvCapture*. En fuentes de tipo vídeo se ha diseñado *scene_camera* para que la reproducción se repita si ésta llega al final del vídeo y en fuentes remotas con el objetivo de reducir la latencia en la obtención del *frame* se ha diseñado un *buffer* para que vaya acumulando los frames. Gracias a la facilidad del objeto *CvCapture* y al diseño del *scene_camera* se consigue abstraer al usuario del tipo de fuente a utilizar.

- **Representación del entorno 3D**. Este objetivo se ha alcanzado gracias al uso de la biblioteca *OpenGL*. Por medio de operaciones y comandos se representan los objetos, el suelo y las fuentes que forman el entorno 3D. A parte de la representación, para permitir que el usuario pueda adaptar la vista 3D se ofrece la posibilidad de rotar, desplazar o hacer zoom sobre la vista 3D. Las ventajas de esta solución recaen en la usabilidad por parte del usuario para que con un simple golpe de vista pueda identificar los objetos, distinguir cada una de las fuentes y controlar la vista 3D; y en la eficiencia porque *OpenGL* ofrece un buen rendimiento para la visualización e interacción con entornos 3D. Además, *OpenGL* es la biblioteca de representación gráfica más portada en diversidad de dispositivos (multiplataforma en términos de Software y Hardware).

- **Cálculo del posicionamiento 3D**. Este objetivo se ha alcanzado utilizando la técnica *raytracer* para obtener el posicionamiento 3D de los objetos detectados y sus dimensiones. Dicha técnica requiere conocer la información sobre la posición y orientación de las cámaras. Para ello se realiza un proceso de posicionamiento utilizando *OpenCV* y obtener la información adecuada para utilizar el *raytracer*. Las ventajas de utilizar un *raytracer* para calcular el posicionamiento recaen en la eficiencia de éste (gracias a un algoritmo de aceleración basado en *grid*) y en la exactitud del cálculo de las posiciones y dimensiones del objeto.

- **Soporte para pruebas deterministas**. Este objetivo se ha alcanzado modelando y diseñando con *Blender* escenas de entornos (con tableros de ajedrez, con objetos para

el *raytracer* y con la calle *Paseo de la Universidad*) para probar el funcionamiento de los módulos de calibración, posicionamiento y *tracking*. La ventaja de diseñar y modelar los entornos con *Blender* ha sido un ahorro en el coste del proyecto y la posibilidad de controlar totalmente la exportación de modelos y el desarrollo de diversas pruebas para los componentes de la plataforma.

- **Soporte para el calibrado**. Este objetivo se ha alcanzado utilizando la biblioteca *OpenCV*. El sistema es capaz de realizar la calibración de la fuente mediante la detección de un patrón con forma de tablero de ajedrez. Gracias al uso del tablero de ajedrez como patrón se obtiene muy buena precisión en los resultados y es fácilmente replicable por otros usuarios del sistema.

- **Soporte para la definición de áreas I/O**. Este objetivo se ha alcanzado permitiendo que el usuario pueda dibujar sobre la vista de la fuente las áreas I/O. También se le permite al usuario el seleccionar áreas previamente creadas o su posterior eliminación. Para representar los diferentes tipos de áreas se han utilizado los colores (rojo, verde, azul) con la ventaja de hacer eficiente el proceso de creación de la máscara mediante la separación de los canales *RGB*. Las ventajas de las soluciones adoptadas para cumplir este objetivo recaen principalmente en la usabilidad del usuario ofreciendo una forma cómoda, precisa y eficiente en el uso de recursos para la definición de las áreas.

- **Soporte para la definición de zonas fiables**. Este objetivo también ha sido alcanzado basándose y reutilizando las soluciones aportadas para cumplir el objetivo anterior. En lo que se diferencia es en la forma de representar las áreas (con escala de grises) y en la forma de calcular la máscara. La solución adoptada permite que sea totalmente transparente al usuario.

- **Soporte para la gestión y comunicación de agentes**. Este objetivo se ha alcanzado diseñando el sistema para trabajar con agentes en los procesos de segmentación, *tracking*, clasificación e identificación. Con esta solución se ofrece un buen rendimiento en la gestión y paralelismo de los procesos que se están ejecutando y una total transparencia al usuario.

- **Despliegue de Agentes de aprendizaje**. Este objetivo se ha alcanzado empleando agentes que se encarguen de utilizar el algoritmo *codebook* para aprender las partes inmóviles de las móviles. *Codebook* ofrece buen rendimiento y mecanismos de limpieza de la información antigua para que se haga un consumo eficiente de la memoria utilizada.

- **Despliegue de Agentes de segmentación**. Este objetivo también se ha alcanzado y la forma de realizar la segmentación consiste en utilizar el modelo de aprendizaje *codebook* y obtener sus diferencias con la imagen actual. Las ventajas de esta solución son similares a la solución adoptada en el anterior objetivo.

- **Despliegue de Agentes de *tracking***. El uso de un histórico donde se almacena la información de los objetos antiguos permite su comparación con la información nueva y realizar su seguimiento y actualización. Este histórico ha sido diseñado de forma que las operaciones de acceso a él sean eficientes y el consumo de memoria utilizado por información antigua sea liberado cuando la información no sea necesaria.

- **Despliegue de Agentes de clasificación**. Este objetivo se ha alcanzado permitiendo que el sistema tenga agentes encargados de realizar la clasificación del objeto en *persona* o *vehículo* atendiendo a las dimensiones del objeto. Las ventajas de esta solución son la eficiencia del algoritmo de clasificación y una buena tasa de acierto debido a que existe una gran diferencia entre las dimensiones de un vehículo y una persona.

- **Despliegue de Agentes de fusión**. Se han desarrollado algoritmos para fusionar la información proveniente de los agentes encargados del *tracking* de cada una de las fuentes. Estos algoritmos han sido diseñados para que sean eficientes o fáciles de mantener o modificar.

- **Interfaz gráfica parcialmente desacoplada**. Este objetivo se ha alcanzado desarrollando el sistema siguiendo el patrón *multicapa* donde las interfaces gráficas quedan separadas de la lógica de los procesos de calibración, posicionamiento, definición de áreas y *tracking*. Las ventajas de la solución recaen en facilidad en las labores de mantenimiento y reutilización de ciertos módulos o lógica para otros proyectos.

- **Formatos de intercambio**. Este objetivo se ha alcanzado permitiendo que el sistema exporte la información del calibrado, posicionamiento, áreas y *tracking* en archivos XML y de texto. Se ha empleado la biblioteca *TinyXml* para el procesamiento de archivos XML. Gracias a estas soluciones es posible representar la información aportando estructura y significado a los datos almacenados.

- **Gestión de datos de configuración**. Este objetivo se ha alcanzado creando un módulo que permita procesar archivos que contengan información sobre los procesos de calibración, posicionamiento y definición de áreas. La ventaja principal es que la solución ha sido modularizada de tal forma que puede ser reutilizada por cualquier otra herramienta que requiera gestionar archivos con esas estructuras de información.

- **Escalabilidad**. El diseño del prototipo se ha realizado teniendo en cuenta objetivos de reutilización y escalabilidad. Aunque trabaja con un máximo de tres fuentes por razones de limitación de la resolución de pantalla y necesidades computacionales, realizando simples modificaciones se puede aumentar a un mayor número de fuentes.

- **Estándares libres**. Este objetivo se ha alcanzado desarrollando el sistema empleando herramientas y un sistema operativo libre.

7.2. Propuestas de trabajo futuro

A continuación se resumen algunas propuestas como trabajo futuro que han quedado fuera del alcance del desarrollo de este proyecto:

- **Representación 3D del entorno a monitorizar**. Esta propuesta consiste en dotar a TraceMon la funcionalidad de poder importar el modelo 3D del entorno que se está monitorizando. Para ello mediante alguna herramienta externa como *Blender* se crea el modelo 3D del entorno. Dicho modelo 3D se exporta de *Blender* en algún formato estándar como *Obj* y este archivo sería posteriormente importado en TraceMon.

  Para importar este archivo, se tendría que crear una clase que represente la interfaz gráfica por la cual el usuario pueda realizar la importación del archivo con extensión *Obj* y otra clase que se encargue de procesar la información del archivo con extensión *Obj*. La clase que representa la interfaz principal se encargará de comunicarle a la clase de procesamiento el archivo *Obj* a procesar y ésta se comunicará con las clases *World* del paquete del *raytracer* y *scene_3D*. La clase *scene_3D* tendría que modificarse para que permitiese representar en la vista 3D los objetos que le va comunicando la clase encargada de procesar el archivo con extensión *Obj*. En cambio la clase *World* y el paquete de *raytracer* no tendría que ser modificado porque ha sido diseñado e implementado para contener los objetos que componen el entorno. El diseño detallado y posterior implementación de esta propuesta podría realizarse en un plazo de dos semanas por un Ingeniero en Informática contratado a Tiempo Completo.

- **Completar información en la vista 3D**. Esta propuesta consistiría en añadir a la vista 3D la visualización de la información de los identificadores, dimensiones y fiabilidad sobre las cajas de los objetos que se van detectando. Para ello consistiría en crear con *OpenGL* el modelo de una etiqueta resumen donde representar dicha información. Tras tener modelada la etiqueta, sería necesario crear un método privado

en la clase *scene_3D* para que despliegue dicha etiqueta encima de la caja que representa al objeto. Si se realiza la propuesta anterior se tendría que modificar la forma de dibujar las cajas que envuelven a los objetos. Esto se debe a que si el entorno presenta pendientes y los objetos están subiendo esas pendientes, las cajas deben de representarse con la misma inclinación que la pendiente. El realizar esto último implicaría cambios en el paquete del *raytracer* para que proporcione información adicional sobre la inclinación que tiene el suelo con respecto a un plano horizontal. Dicha información tendría que ser almacenada en la clase *object_information* y los algoritmos de fusión y actualización del histórico deberían adaptarse para procesarla. Por último habría que realizar las modificaciones necesarias para que en el método de pintar cajas de la clase *scene_3D* se tenga en cuenta dicha inclinación a la hora de dibujar la caja. La estimación para el desarrollo de esta propuesta sería de tres semanas.

- **Reproducir comportamientos**. Esta propuesta consistiría en reproducir los comportamientos de los objetos señalados en una interfaz externa. La idea es que se cree una nueva interfaz gráfica para representar el comportamiento que realizó un objeto seleccionado por el usuario. Se habilitarán controles para avanzar, retroceder, o parar la reproducción del comportamiento. Esta propuesta sería totalmente novedosa para el diseño de TraceMon y consistiría en mantener un duplicado del histórico para que no se eliminen los objetos que van desapareciendo de la escena. Crear una clase que represente la nueva interfaz gráfica con las opciones de seleccionar un objeto, reproducir, pausar, avanzar o retroceder la visualización de su comportamiento. Dicha clase utilizará la funcionalidad implementada en las anteriores propuestas para representar el entorno 3D y las propiedades del objeto. La estimación de esta propuesta para un ingeniero informático sería de cinco semanas.

- **Optimización de algunos algoritmos**. Esta propuesta consistiría en mejorar los algoritmos elaborados en TraceMon con el fin de que mejoren su rendimiento. Al estar diseñado e implementado de forma modular no supondría mucho coste la optimización de los algoritmos. Sería necesario un estudio de los cuellos de botella concretos de cada algoritmo para proceder a su mejora concreta empleando técnicas de procesamiento distribuido en CPU o GPU. El coste de cada caso dependerá del grado de optimización del grado de optimización deseado y de la aproximación seguida.

7.3. Conclusiones personales

Durante el estudio de la ingeniería Informática se han aprendido técnicas y metodologías que ayudan en las fases de desarrollo de un proyecto (captura de requisitos, análisis, diseño, implementación, pruebas y mantenimiento). Por tanto, el desarrollo de un Proyecto Fin de Carrera es un punto muy importante para poner en práctica algunas de esas técnicas.

Pero hay que destacar que el mundo de las tecnologías de la información es muy amplio y está continuamente cambiando y por lo tanto la ingeniería Informática no puede tocar todas las ramas que abarca. Durante el estudio de la asignatura *Inteligencia Artificial* despertó en mí la curiosidad de conocer más este área y la relacionada con la visión por computador. Y de aquí nació la idea de desarrollar TraceMon.

Gracias al desarrollo de TraceMon he adquirido conocimientos que desconocía como el funcionamiento de un *raytracer*, el calibrado y posicionamiento de cámaras y los algoritmos de segmentación. Combinando estos nuevos conocimientos con los aprendidos en la titulación he conseguido adquirir cierta experiencia profesional para adentrarme en el mundo laboral.

ANEXOS

A

Manual de referencia

La estructura que presenta el CD que acompaña a este documento es:

- **TraceMon**. Directorio donde se encuentra el código fuente de la aplicación TraceMon.
- **Memory_latex_src**. Directorio donde se encuentra el código fuente para la generación de este documento.
- **Debug**. Directorio donde se encuentran las pruebas de depuración realizadas a los módulos de calibración, posicionamiento y *raytracer*.
- **Manual_doxygen**. Directorio donde se encuentran los archivos fuente para generar un documento donde se describen los archivos fuente que componen la aplicación. Para generar el documento dentro del directorio ejecutar:

  ```
  doxygen doxygen.cfg
  ```

  Dentro de *doc* se encuentran dos directorios *html* y *latex*. El directorio *html* contiene la documentación en formato *html* lista para ser explorada abriendo el archivo *index.html*. Y en el directorio *latex* se encuentran los archivos fuente para generar la documentación en formato *PDF*. Para ello ejecutar dentro del directorio *latex* la orden:

  ```
  make
  ```

Esto generará un documento llamado *refman.pdf* donde se encuentra la documentación en formato *PDF*.

- **Tests**. Directorio donde se encuentran los archivos de vídeo de las pruebas realizadas al sistema. Dentro de este directorio se encuentran dos directorios *Real* y *Virtual*. Dentro de cada uno de ellos se encuentran los directorios *Calibration*, *Position*, *Areas* y *Tracking*. En estos directorios se encuentran los archivos de vídeo utilizados y los archivos XML que almacenan la información de los resultados de utilizar estos vídeos en la aplicación.

A.1. Manual de usuario

En esta sección se describe la interacción que puede realizar el usuario en cada uno de los procesos que permite TraceMon. En la figura A.1 se muestra la interfaz gráfica principal donde el usuario puede seleccionar el número de fuentes que componen el sistema de monitorización con el menú *Options* (señalado por el recuadro verde).

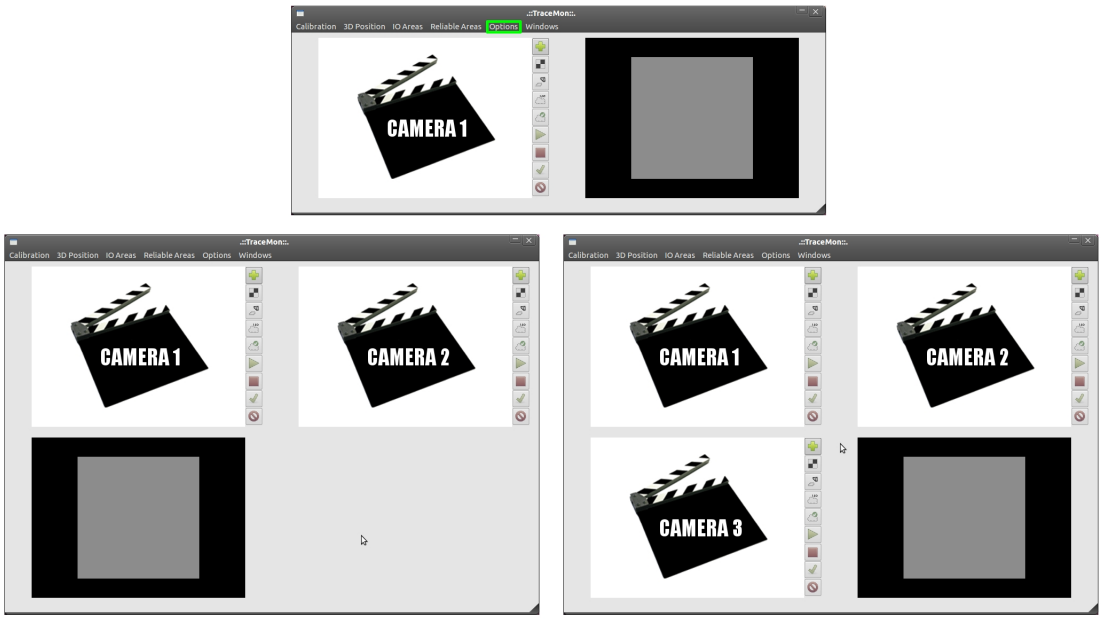

FIGURA A.1: Diferentes vistas la interfaz gráfica principal. “Ver en Anexo B imagen a color”

Para tener el control sobre la vista 3D (señalada en verde en la figura A.2) el usuario puede:

- Rotar la vista 3D pulsando sobre ella el botón izquierdo del ratón y, manteniendo pulsado el botón, mover el ratón sobre la vista hasta encontrar la rotación deseada.

- Desplazar la vista 3D pulsando hacia dentro la rueda del ratón y, manteniéndola pulsada, mover el ratón sobre la vista hasta encontrar el adecuado desplazamiento horizontal y vertical.
- Hacer zoom en la vista 3D moviendo la rueda del ratón hacia adelante o hacia atrás para acercar o alejar la escena.

FIGURA A.2: Eventos sobre la vista 3D. “Ver en Anexo B imagen a color”

A.1.1. Añadir las fuentes que forman el sistema

El usuario debe especificar el origen de la fuente para ello se utiliza el botón  . Pulsando dicho botón se abre la interfaz gráfica mostrada en la figura A.3.

En ésta es posible elegir el tipo de fuente asociada e indicar la información para conectar a ella. Tras añadir el origen de la fuente, se activará el botón  a través del cual el usuario puede iniciar el proceso de calibración.

A.1.2. Proceso de calibrado de las fuentes

El proceso de calibrado comienza cuando se pulsa el botón  y aparece la interfaz mostrada en la figura A.4.

En ella el usuario puede elegir si realizar el proceso de calibración o cargar un archivo XML que contenga los resultados del proceso de calibración realizado en otro momento. Si se decide realizar el proceso de calibración se debe especificar:

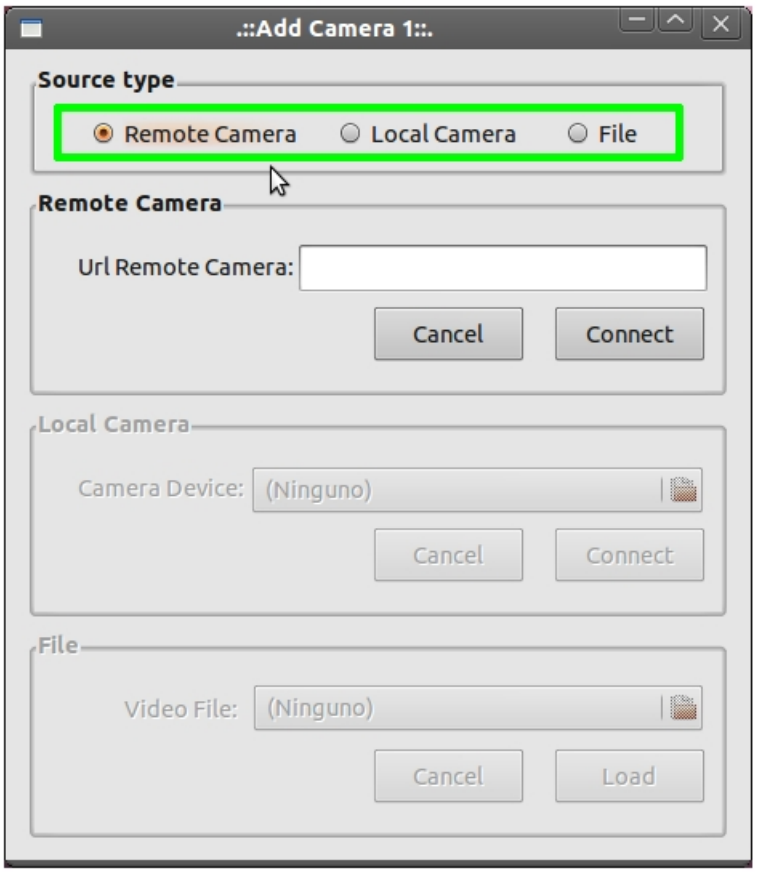

FIGURA A.3: Interfaz gráfica para añadir el origen de una fuente.

- El número de tomas de tableros de ajedrez correctos para concluir la calibración (entre 12-20)
- El tamaño del lado de los cuadrados que componen el tablero de ajedrez. Dicha dimensión debe de especificarse en *decímetros*.
- El número de esquinas internas que tiene el tablero de ajedrez en el eje horizontal (entre 3-13).
- El número de esquinas internas que tiene el tablero de ajedrez en el eje vertical (entre 3-13).

Cuando se han especificado estos parámetros y el usuario pulsa el botón *Start* comienza el proceso de calibración. El proceso de calibración es visualizado en el componente de la fuente, y cuando el usuario quiere realizar una toma sobre el frame actual que está visualizando debe pulsar el *botón izquierdo* del ratón sobre la vista de la fuente. De esta forma se paraliza la reproducción de la fuente y se muestra el resultado obtenido de analizar

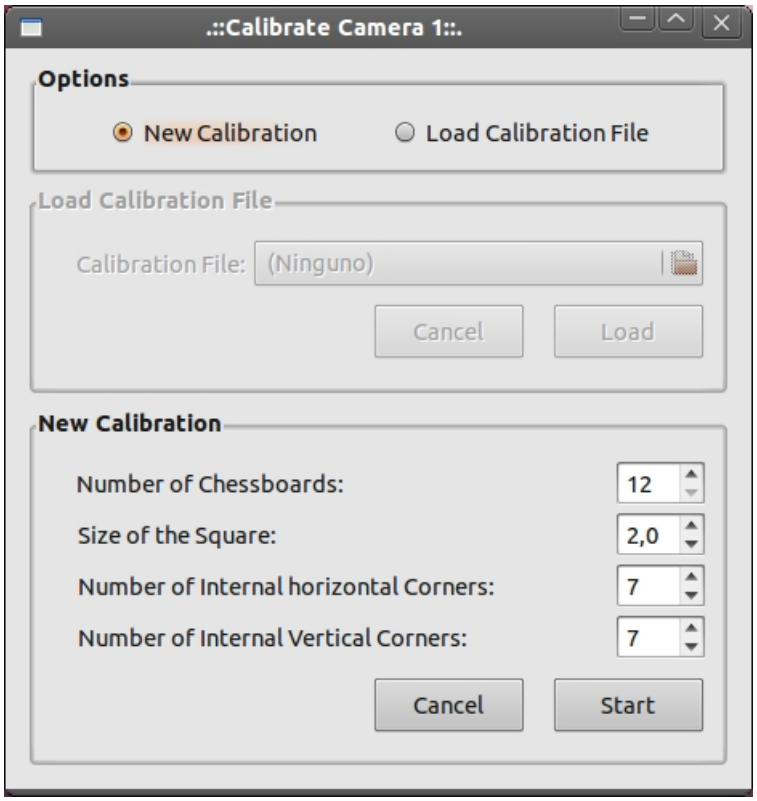

FIGURA A.4: Interfaz gráfica para iniciar el proceso de calibración.

ese frame. Si se ha detectado el tablero de ajedrez, se activarán dos nuevos botones  y  para aceptar o rechazar la toma del tablero. Si acepta el análisis entonces aparecerá la palabra *Correct!*. En el caso de que el análisis del frame elegido sea incorrecto o el usuario rechace los resultados obtenidos del análisis aparecerá la palabra *Incorrect!*. Después de que el usuario evalúe los resultados del análisis puede continuar con el flujo actual de vídeo pulsando el *botón derecho* y si fue correcta la toma, el contador de tableros será incrementado. En cualquier momento el usuario puede cancelar la ejecución del proceso que se esté realizando (calibración, posicionamiento, definición de áreas I/O, fiables y *tracking*) pulsando el botón  . Todo lo explicado anteriormente se visualiza en la figura A.5.

Después de finalizar el proceso de calibración se activará el botón  , el cual indica al usuario que puede realizar el proceso de posicionamiento. También en el menú de *Calibration* se habrá activado la opción de poder guardar la información del proceso de calibración de la cámara 1 ver figura A.6.

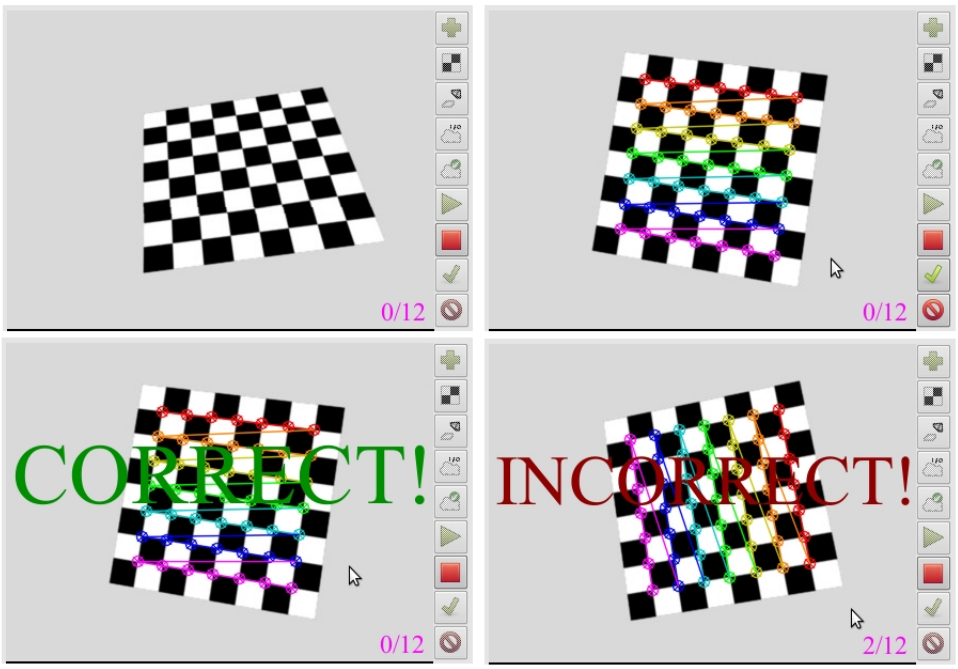

FIGURA A.5: Interfaces gráficas del proceso de calibración.

FIGURA A.6: Opción de guardar la información del proceso de calibración.

A.1.3. Proceso de posicionamiento de las fuentes

Cuando el usuario quiera iniciar el proceso de posicionamiento pulsará el botón  y se creará la interfaz que se muestra en la figura A.7.

En esta interfaz se da la opción de poder cargar los resultados obtenidos en un procesamiento anterior o de realizar el proceso de posicionamiento. Si el usuario elige realizar el proceso de posicionamiento debe especificar las cuatro coordenadas 3D que definen la marca dando un orden. Dichas dimensiones también deben de ser expresadas en *decímetros*. Y una vez que se han introducido y se pulsa el botón *Start* comienza el proceso de posicionamiento. En el componente de la fuente correspondiente se visualizará lo que

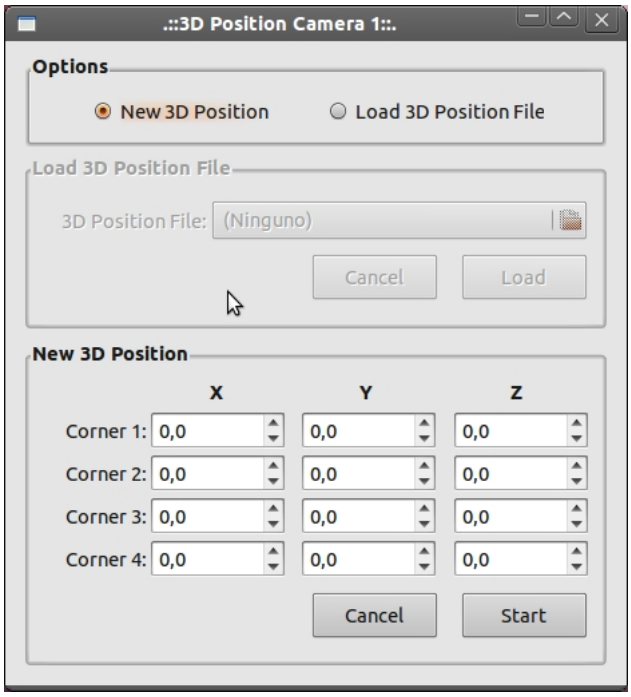

FIGURA A.7: Interfaz gráfica para iniciar el proceso de posicionamiento.

está captando la fuente y en la vista 3D se habrá dibujado la marca con las coordenadas que el usuario especificó.

Para especificar las coordenadas 2D en la vista primero se tiene que congelar la vista de la fuente, esto lo realiza el usuario pulsando el *botón izquierdo* sobre la vista de la cámara. Y una congelada la imagen de la fuente, el usuario especifica las coordenadas en el mismo orden en el que indicó las coordenadas 3D. Para indicarlas se posiciona encima de la coordenada 2D y pulsa el *botón derecho* del ratón y se dibujará un punto con el color asociado al orden de la coordenada *(1 = rojo, 2 = amarillo, 3 = azul, 4 = rosa)*.

Cuando se hayan definido las cuatro coordenadas y se vuelva a pulsar el *botón derecho* del ratón se llevará a cabo el proceso de posicionamiento y se mostrarán las coordenadas 2D en color *verde*. Estas coordenadas 2D de color verde representan las coordenadas que la aplicación ha obtenido de aplicar los resultados del proceso de posicionamiento a las coordenadas reales. De ahí que si los resultados obtenidos son buenos, estas coordenadas 2D de color verde deben coincidir con las señaladas. Para finalizar el proceso se vuelve a pulsar el *botón derecho* y en la vista 3D se puede ver la cámara virtual dibujada con la posición y orientación obtenida. En la figura B.5 se puede ver el proceso del posicionamiento.

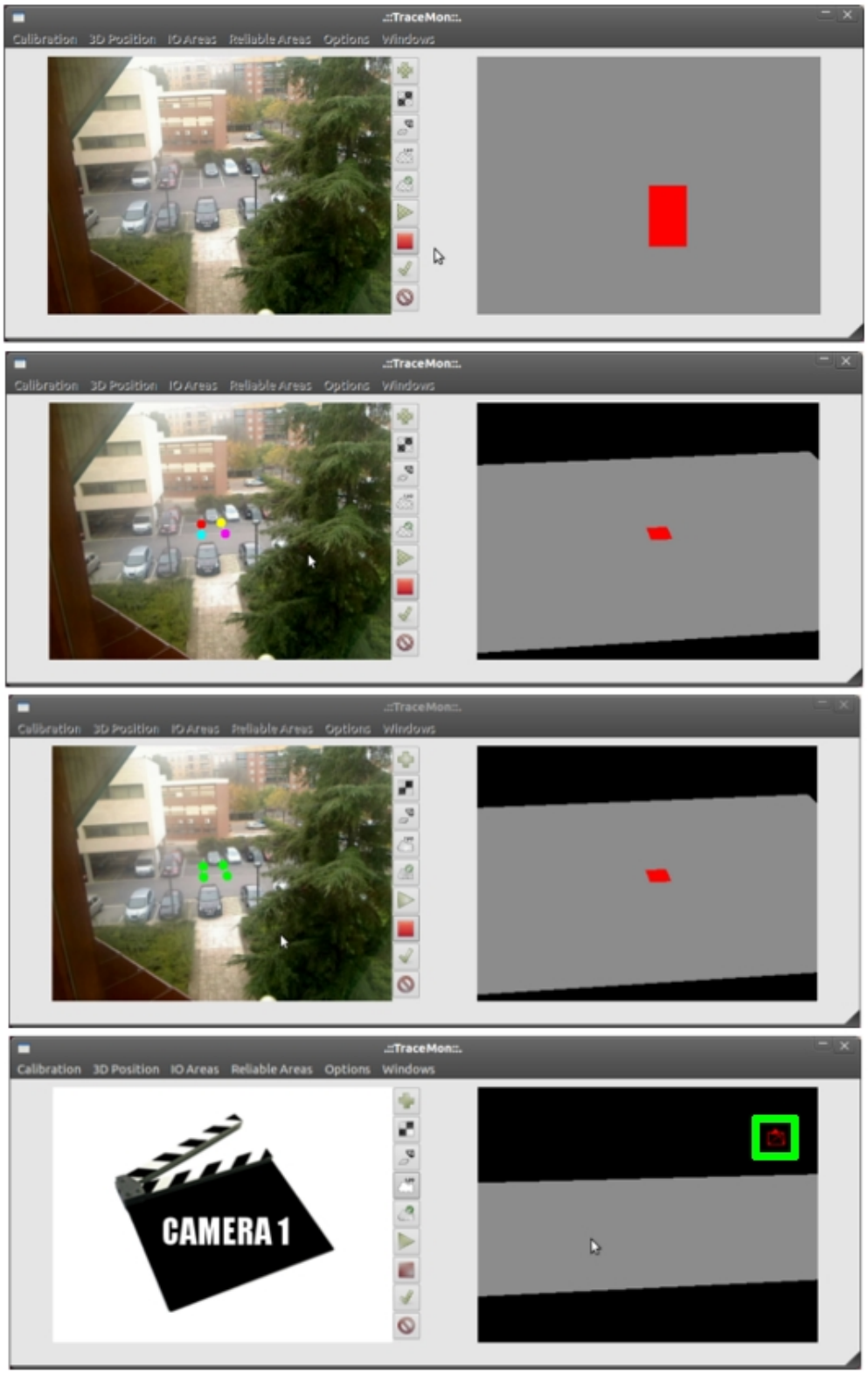

FIGURA A.8: Interfaces gráficas del proceso de posicionamiento. “Ver en Anexo B imagen a color”

Después de terminar el proceso de posicionamiento se activa en el menú *3D Position* el botón para poder guardar los resultados obtenidos del posicionamiento de la fuente. Y también se activa el botón  para permitir que el usuario realice el siguiente proceso que consiste en la definición de áreas de I/O.

A.1.4. Definición de áreas de I/O

Si se pulsa el botón  se crea una interfaz gráfica como la que se muetra en la figura A.9 para poder iniciar el proceso de definición de áreas.

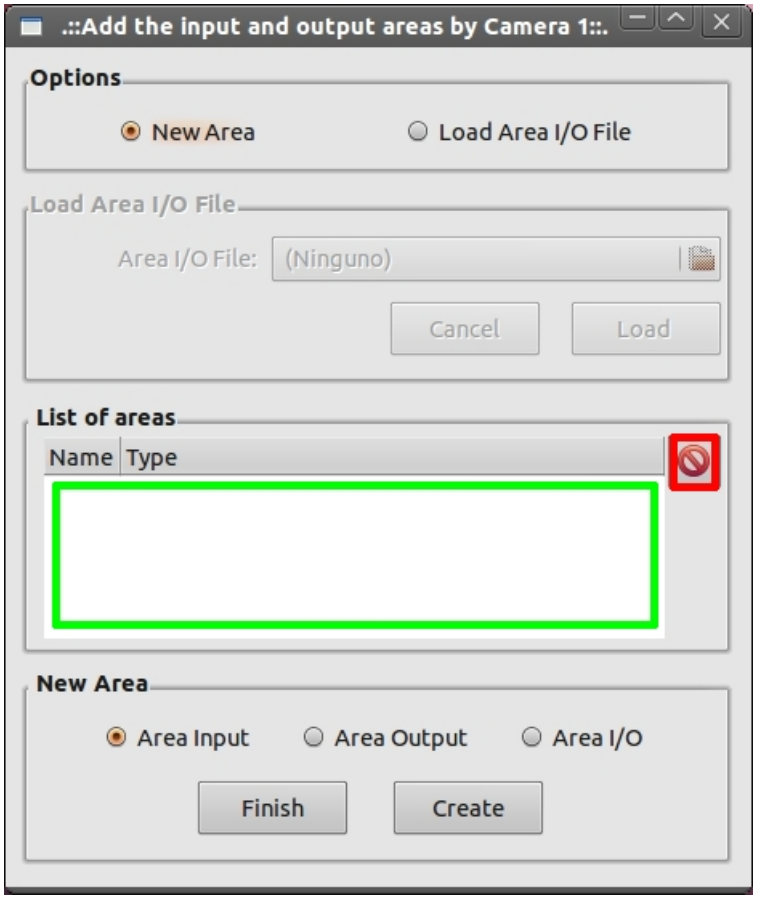

FIGURA A.9: Interfaz gráfica para iniciar el proceso de definición áreas I/O.

Como en las anteriores interfaces se ofrece la opción de poder cargar un archivo donde se encuentren las áreas I/O ya definidas. La zona señalada por el recuadro verde en la figura A.9 representa el listado de las áreas que hay creadas. Y es a través de éste por el que el usuario puede seleccionar un área de las que ha definido. Este área seleccionada es resaltada en la vista de la fuente por un color *amarillo*. Y para eliminarla basta con

tener seleccionada el área que se quiere eliminar y pulsar el botón señalado por el recuadro rojo en la figura A.9. La forma de definir las áreas es pulsando el *botón izquierdo* y sin soltarlo ir trazando el contorno del área. Después cuando se suelta el botón ya ha quedado definido el contorno y se puede crear el área seleccionado su tipo *(entrada = verde, salida = rojo o entrada/salida = azul)*. Cuando el usuario haya terminado de crear las áreas correspondientes finaliza el proceso pulsando el botón *Finish* y se crea la máscara asociada a dichas áreas. Todo este proceso se puede ver en la figura A.11.

Una vez finalizado la definición de áreas I/O se activa en el menú *IO Areas* la opción de poder guardar la máscara de I/O definida para dicha fuente. También se activa el botón  para que el usuario pueda comenzar la definición de áreas fiables.

A.1.5. Definición de áreas fiables

Si se pulsa el botón  aparece una interfaz gráfica como la mostrada en la figura A.10.

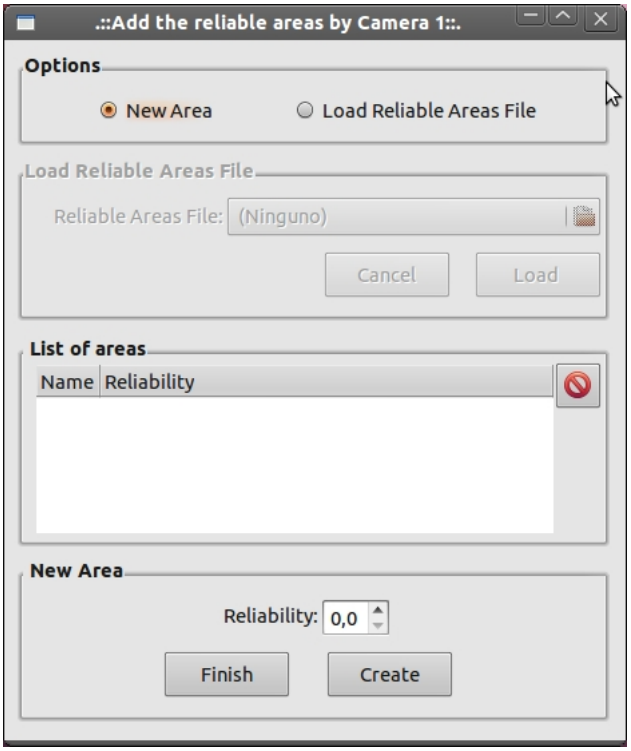

FIGURA A.10: Interfaz gráfica de la definición de áreas fiables.

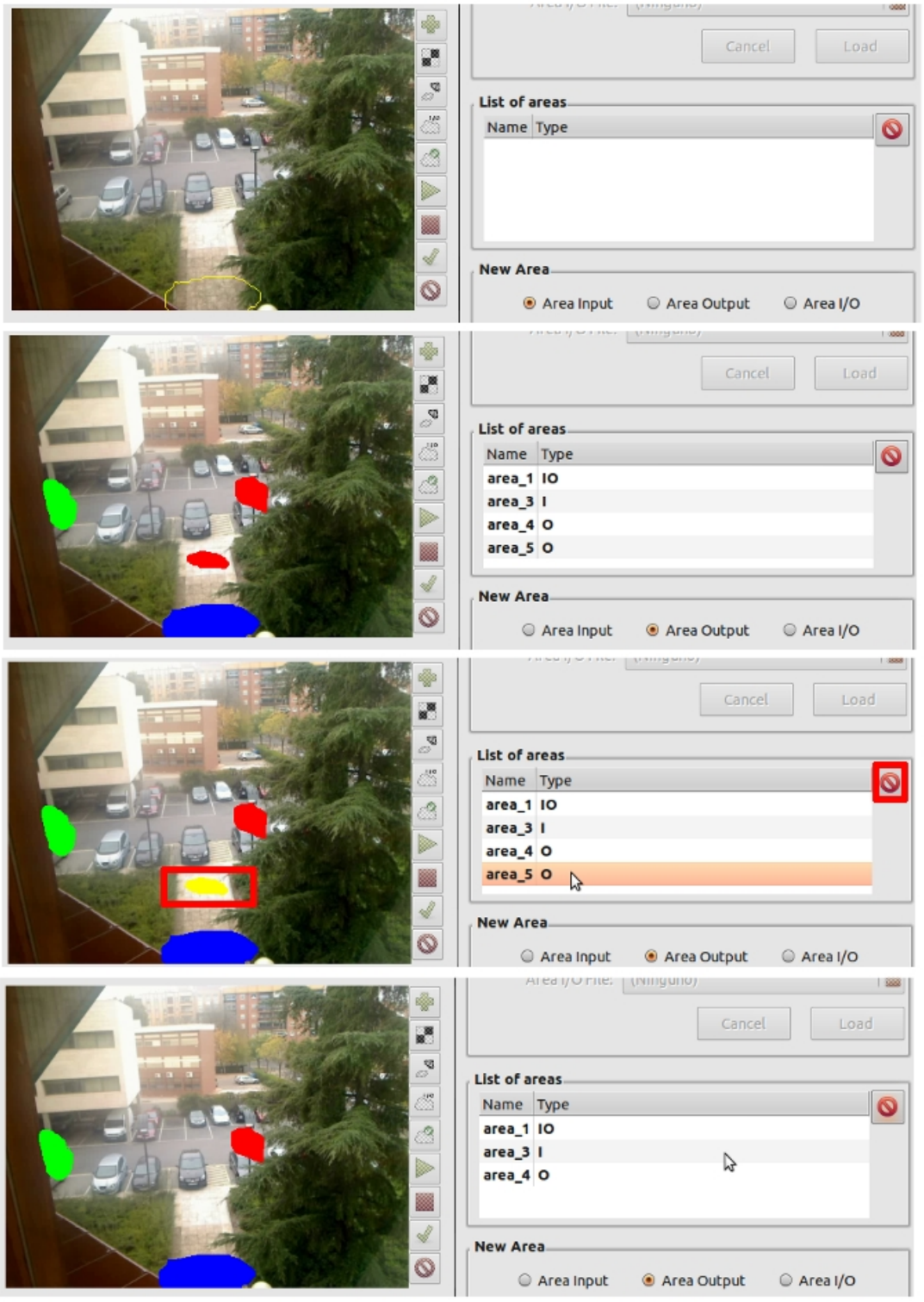

FIGURA A.11: Interfaces gráficas de la definición de áreas I/O. “Ver en Anexo B imagen a color”

Donde se puede apreciar que es muy similar a la interfaz gráfica para la definición de áreas I/O. El funcionamiento es exactamente el mismo, salvo que aquí se elige el grado de fiabilidad en lugar del tipo de área. También se ofrece la posibilidad de poder cargar un archivo donde se encuentren definidas las áreas.

El proceso de definición de áreas fiables finaliza cuando el usuario pulsa el botón *Finish* y termina la creación de la máscara de áreas fiables. Tras finalizar el proceso, se activa en el menú de *Reliable Areas* la opción de poder guardar las áreas fiables definidas en un archivo XML; y también se activa el botón  para que el usuario pueda realizar la fase de *tracking*.

A.1.6. Proceso de *tracking*

Si se pulsa el botón  comenzará la fase de aprendizaje del fondo y después el seguimiento.

Como se muestra en la figura A.12 el objeto detectado es señalado con un rectángulo verde (si la clasificación a priori es *persona*) o rojo (si la clasificación a priori es *vehículo*). Y luego su representación mediante una caja en la vista 3D.

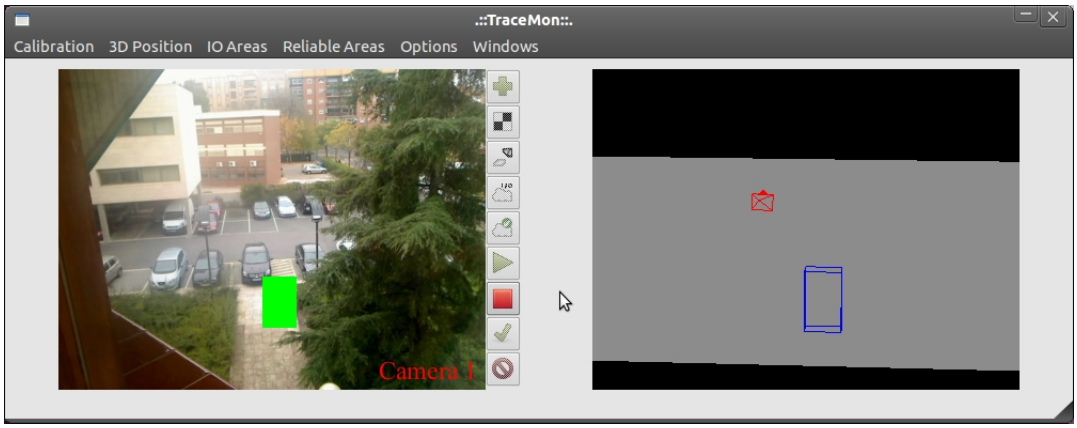

FIGURA A.12: Interfaz gráfica del proceso de *tracking*. “Ver en Anexo B imagen a color”

También si se quiere ver más información sobre los objetos detectados en la sección *Windows* del menú se puede acceder a la opción de *Log* el cual permite abrir una interfaz gráfica donde se puede ver la fecha, el identificador del objeto y su descripción (ver figura A.13).

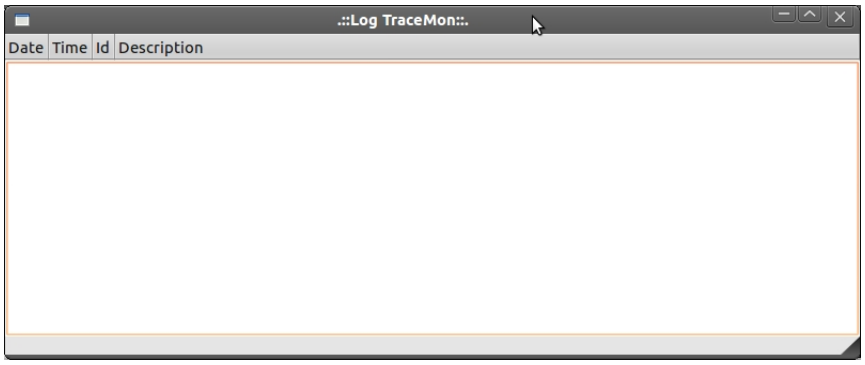

FIGURA A.13: Interfaz gráfica de información de los objetos detectados.

Por último mencionar que la aplicación genera dentro del directorio del ejecutable un archivo llamado *export_tracking.txt* donde se encuentra toda la información de los objetos que han sido detectados con mucho más detalle para que pueda ser procesada por otra herramienta externa.

B

Imágenes a color

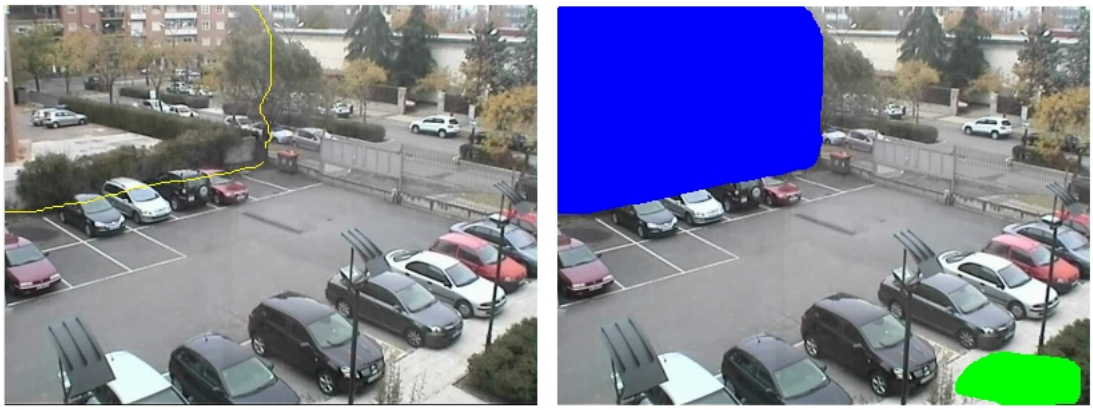

FIGURA B.1: Herramienta para la definición de áreas sobre la vista de la fuente.

FIGURA B.2: Eventos sobre la vista 3D.

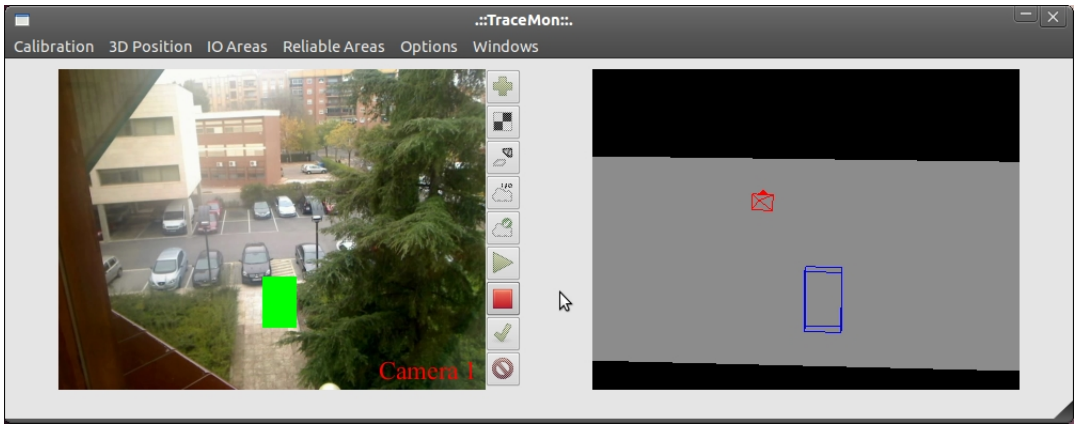

FIGURA B.3: Interfaz gráfica del proceso de *tracking*.

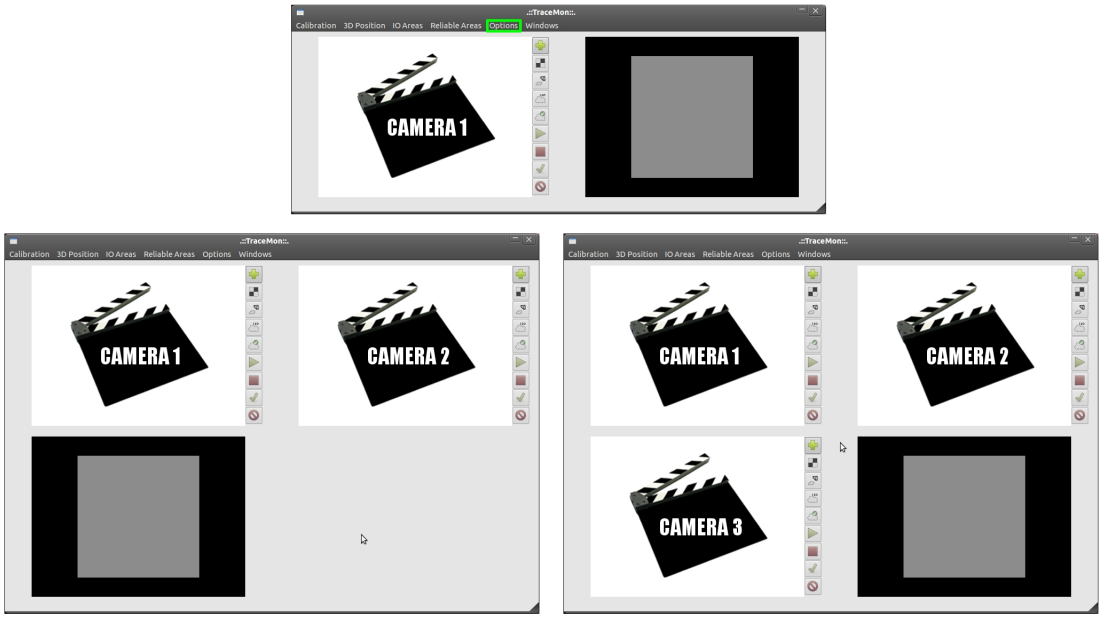

FIGURA B.4: Diferentes vistas la interfaz gráfica principal.

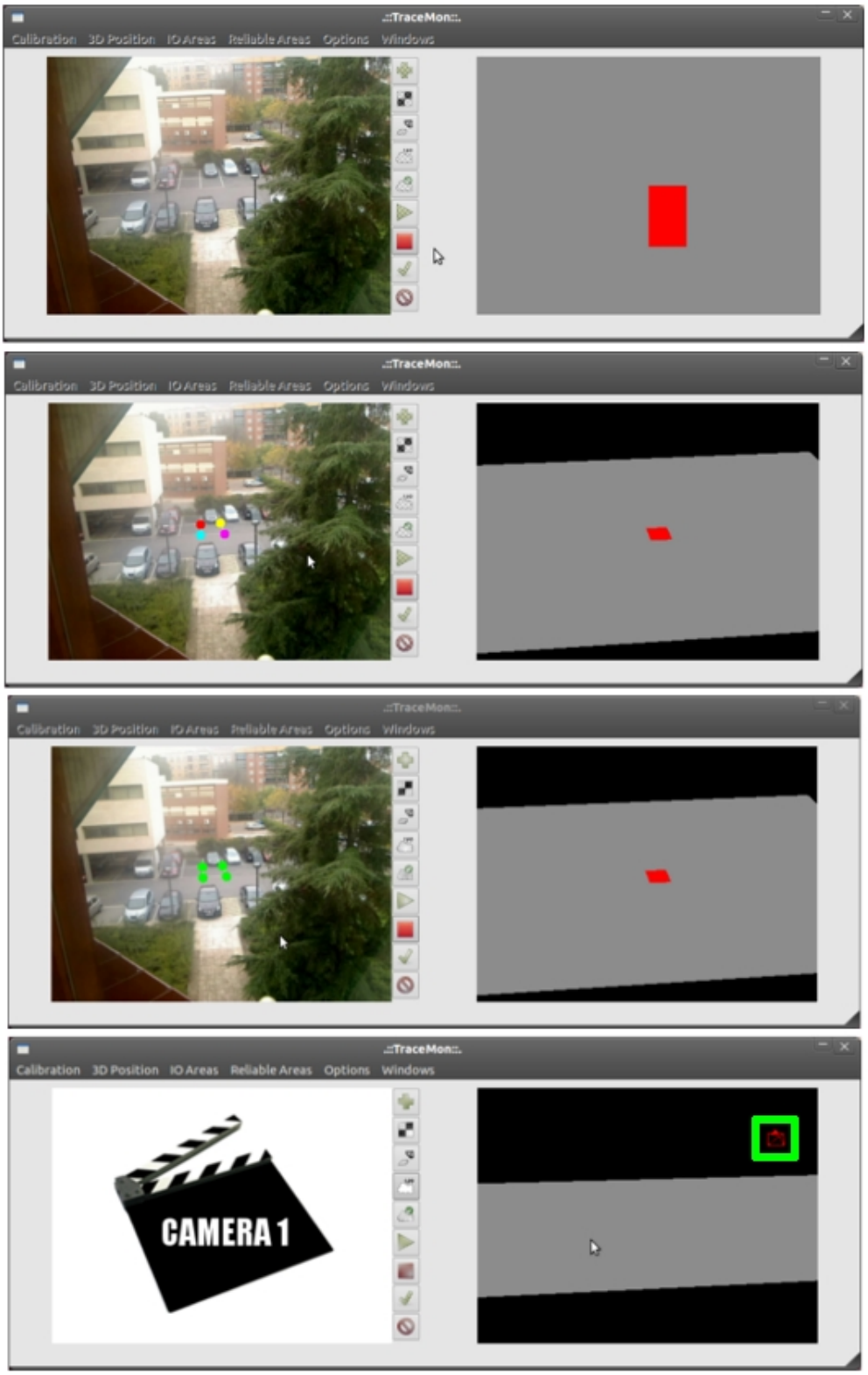

FIGURA B.5: Interfaces gráficas del proceso de posicionamiento.

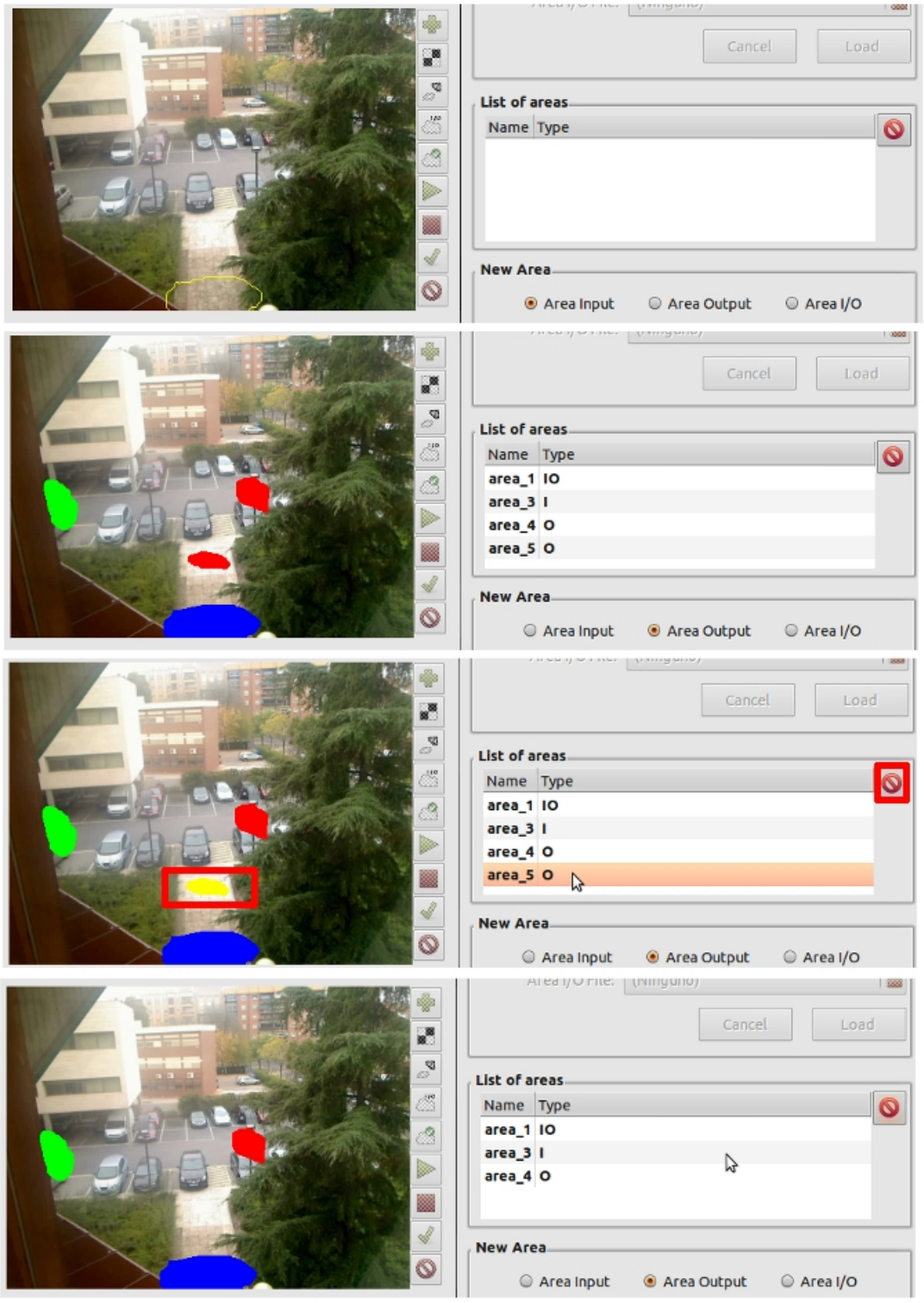

FIGURA B.6: Interfaces gráficas de la definición de áreas I/O.

Bibliografía

[AE10] Fathallah Nouboud Driss Mammass Jean Meunier Abderrahim Elmoataz, Olivier Lezoray. *Image and Signal Processing*. Springer, 2010.

[BE04] Chuck Allison Bruce Eckel. *Thinking in C++.: Practical programming*. Prentice Hall, 2004.

[BK08] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., 2008.

[Bro] Lisa M. Brown. View Independent Vehicle/Person Classification. *IBM T.J. Watson Research Center.*

[CG09] Néstor Romero y Julián Quiroga Sepúlveda. Carolina García. Detección y Seguimiento de Personas en un Cruce Peatonal. *XIV Simposio de tratamiento de señales, imágenes y visión artificial – STSIVA.*, 2009.

[Chr06] James Chronister. *Blender Basics: 2nd Edition*. 2006.

[Dam05] Consorcio Dammad. *Diseño y aplicación de modelos multiagente para la ayuda a la decisión*. Dykinson, S.L, Meléndez Valdés, 61-28015 Madrid, 2005.

[DD95] D. Dementhon and L. Davis. Model-based object pose in 25 lines of code. 15:123–141, June, 1995.

[DS10] The Khronos OpenGL ARB Working Group Dave Shreiner. *OpenGL programming guide: Seventh Edition*. Addison-Wesley, 2010.

[Fer09] David Vallejo Fernández. Tesis Doctoral: Arquitectura Multi-Agente basada en Servicios para un Sistema de Vigilancia Cognitiva. *Universidad de Castilla-La Mancha*, pages 1–76, 2009.

[Gar] José Luis Lerma García. *Fotogrametría moderna: analítica y digital*. Reproval, S.L.

[IB96] M. Isard and A. Blake. *Contour tracking by stochastic propagation of conditional density*. 1996.

[JBB94] DJ Fleet JL Barron and SS Beauchemin. Performance of optical flow techniques. *International Journal of Computer Vision*, 12(1):44–77, 1994.

[KB90] K. Karmann and A. Brandt. Moving object recognition using an adaptive background memory. In Time-Varying Image Processing and Moving Object Recognition. 2, 1990.

[KS06] Peter J. Hammond Knut Sydsaeter. *Matemáticas para el análisis económico*. Pearson Prentice Hall, 2006.

[Lil03] Björn Liljequist. Planes, Homographies and Augmented Reality. September, 2003.

[Mat96] Ofer Matan. Ensembles for supervised classification learning. 1996.

[MT] Hironobu Fujiyoshi. Masamitsu Tsuchiya. Evaluating Feature Importance for Object Classification in Visual Surveillance. *Dept. of Computer Science, Chubu University*.

[Pet00] N. Peterfreund. Robust tracking of position and velocity with Kalman snakes. IEEE Trans. Pattern Anal. Machine Intell. 22:564–569, June, 2000.

[PN94] R. Polana and R. Nelson. Low level recognition of human motion. In Proc. IEEE Workshop Motion of Non-Rigid and Articulated Objects. pages 77–82, 1994.

[RBR06] M. Dastani A.E.F. Seghrouchni J.J. Gomez-Sanz J. Leite G. O'Hare A. Pokahr R.H. Bordini, L. Braubach and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30(1):33–44, 2006.

[Som05] Lam Sommerville. *Ingeniería del Software. Septima edición*. Pearson Educación S.A, 2005.

[Suf07] Kevin Geoffrey Suffern. *Ray Tracing from the Group Up*. A K Peters, 2007.

[TH] Ying-Li Tian and Arun Hampapur. Robust Salient Motion Detection with Complex Background for Real-time Video Surveillance.

[Ver] David Vernon. *Lecture notes in computer science Vol 1843, Computer Vision- ECCV 2000*. Springer.

[VV05] M. Valera and S.A. Velastin. Intelligent distributed surveillance systems: a review. 152(2), April, 2005.

[yPN04] Stuart J. Russell y Peter Norvig. *Inteligencia Artificial: Un enfoque moderno. Segunda edición*. Pearson Educación, S.A, Ribera del Loira, 28 Madrid, 2004.