UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

INGENIERÍA ELECTRÓNICA

Trabajo Fin de Carrera

"Técnicas de Reconstrucción Volumétrica a partir de Múltiples Cámaras. Aplicación en Espacios Inteligentes"

Javier Rodríguez López
2008

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

INGENIERÍA ELECTRÓNICA

Trabajo Fin de Carrera

"Técnicas de Reconstrucción Volumétrica a partir de Múltiples Cámaras. Aplicación en Espacios Inteligentes"

Alumno: Javier Rodríguez López.

Director: D. Daniel Pizarro Pérez.

Tribunal:

Presidente: D. Manuel Mazo Quintas.

Vocal 1º: Da. María Soledad Escudero Hernanz.

Vocal 2º: D. Daniel Pizarro Pérez.

Calificación: .....

Fecha: .....

A mi familia y a Ana por apoyarme tanto durante estos años de estudio.

Agradecimientos

Deseo agradecer a mi tutor D. Daniel Pizarro Pérez la confianza y apoyo recibidos en la realización de este trabajo fin de carrera, así como a los compañeros de Espacio Inteligente por su ayuda y consejos que tanto me han ayudado en este proyecto.

Índice general

1. Resumen **13**
1.1. Resumen . . . 15

2. Memoria **17**
2.1. Introducción. . . . 19
2.1.1. Objetivos y definición del problema. . . . 19
2.1.2. Estructura de la memoria. . . . 21
2.2. Estado del arte. . . . 21
2.2.1. Espacios inteligentes. . . . 21
2.2.2. Técnicas de reconstrucción volumétrica a partir de múltiples cámaras. . . . 24
2.3. Geometría de formación de la imagen. . . . 25
2.3.1. Modelo de cámara Pin-Hole. . . . 25
2.3.2. Calibración de un conjunto de cámaras. . . . 28
2.3.2.1. Escenarios. . . . 29
2.3.3. Homografías. . . . 30
2.4. Técnicas de segmentación de imágenes en cámaras estáticas. . . . 33
2.4.1. Método sencillo. . . . 34
2.4.1.1. Umbral adaptativo según x. . . . 34
2.4.2. Segmentación mediante distancia estadística. . . . 35
2.4.3. Métodos avanzados de segmentación. . . . 36
2.5. Sistema de reconstrucción volumétrica a partir de múltiples cámaras. . . . 36
2.5.1. Obtención de una rejilla de ocupación 3D mediante homografías. . . . 37
2.5.1.1. Visual Hull. . . . 37
2.5.1.1.1. Definición I de V.H: Intersección de los conos visuales. . . . 37
2.5.1.1.2. Definición II de V.H: Volumen máximo. . . . 37
2.5.1.1.3. Construcción de Visual Hull. . . . 37
2.5.1.1.4. Ambigüedad de Visual Hull. . . . 38
2.5.1.2. Visual Hull a partir de homografías. . . . 39
2.5.1.2.1. Visual Hull en una rejilla de ocupación. . . . 39
2.5.2. Algoritmos de coloreado fotorealista. . . . 40
2.5.2.1. Coloreado por el método del cálculo del histograma. . . . 41
2.5.2.1.1. Ambigüedad en la texturización. . . . 42
2.5.2.1.2. Resultados obtenidos. . . . 42
2.5.2.2. Texturización basada en perspectiva inversa. . . . 42
2.5.2.2.1. Resultados obtenidos. . . . 48
2.5.2.3. Coloreado por el Algoritmo del Pintor. . . . 49
2.5.2.3.1. Resultados obtenidos. . . . 52
2.6. Reconstrucción volumétrica aplicada a espacios inteligentes. . . . 54
2.6.1. Detección de múltiples objetos a partir del grid de ocupación. . . . 54

2.6.2. Implementación hardware del sistema. 59
2.6.2.1. Sistema distribuido de adquisición y control de múltiples cámaras. 59
2.6.3. Implementación software del sistema. 61
2.6.3.1. Descripción de la herramienta OpenCV. 61
2.6.3.2. Descripción de la herramienta OpenSG. 61
2.6.3.2.1. Flujograma del programa. 62
2.6.3.2.2. Definición de la cámara OpenSG. 65
2.6.3.2.3. Creación de un escenario realista. 67
2.6.3.3. Arquitectura cliente-servidor para la adquisición y control. . . . 68
2.6.3.4. Medida de la temporización global del sistema. 68
2.6.3.4.1. Temporización para el cálculo de la rejilla 3D. 68
2.6.3.4.2. Temporización para la texturización inversa. 72
2.6.3.4.3. Temporización del Algoritmo del Pintor. 76
2.6.3.4.4. Tiempos totales. 79
2.6.3.5. Diseño distribuido para la obtención de la rejilla de ocupación. . 79
2.7. Conclusiones y trabajos futuros. 81
2.7.1. Líneas futuras de trabajo. 81

3. Manual de Usuario **85**
3.1. Manual 87

4. Pliego de condiciones **91**
4.1. Requisitos de Hardware 93
4.2. Requisitos de Software 93

5. Presupuesto **95**
5.1. Presupuesto 97
5.1.1. Coste de Equipos 97
5.1.2. Coste por tiempo empleado 97
5.1.3. Coste total del presupuesto de ejecución material 97
5.1.4. Gastos generales y beneficio industrial 98
5.2. Presupuesto de ejecución por contrata 98
5.2.1. Coste total 98

6. Bibliografía **99**

Índice de figuras

2.1. Proceso geométrico. 20
2.2. Muestra de contornos a diferentes alturas. 20
2.3. Escenario: Objeto O forma la silueta S_1^k en la cámara k en el instante t_1. 25
2.4. Modelo Pin Hole. 26
2.5. Sistema de coordenadas del plano imagen y el plano imagen normalizado. 27
2.6. Patrón en el proceso de calibración. 29
2.7. Calibración parámetros extrínsecos. 29
2.8. Mesa giratoria, imagen real. 30
2.9. Mesa giratoria, situación de captura. 30
2.10. Mesa giratoria, múltiples capturas. 31
2.11. Captura con un incremento de rotación de 36 grados. 31
2.12. Espacio Inteligente. 31
2.13. Escena cámara-plano Homografía. 32
2.14. Diagrama de escalado. 33
2.15. Diagrama del proceso de segmentación simple. 34
2.16. Diagrama del proceso de segmentación mediante distancia estática. 35
2.17. Variable aleatoria gaussiana F(x). 36
2.18. Ejemplo de resultado de segmentación utilizada. 36
2.19. Ambigüedad en Visual Hull 1. 38
2.20. Ambigüedad en Visual Hull 2. 38
2.21. Ambigüedad en Visual Hull 3. 39
2.22. Rejilla de ocupación. 40
2.23. Rejilla de ocupación. 40
2.24. Histograma típico de un vóxel en escala de grises. 41
2.25. Histograma típico de un vóxel en RGB. 41
2.26. Ambiguedad en la texturización. 43
2.27. Relación error/tamaño del vóxel. 43
2.28. Escena original a reconstruir. 44
2.29. Instantánea del volumen reconstruido y texturizado. 44
2.30. Rotación de la figura. 44
2.31. Algoritmo de coloreado 2. 45
2.32. Algoritmo de coloreado 2, explicación. 46
2.33. Imagen original e Imagen coloreada según la cámara texturizadora. 48
2.34. Distintas vistas de la escena reconstruida. 48
2.35. Escena original a reconstruir. 48
2.36. Escena reconstruida desde varias perspectivas. 49
2.37. Escena reconstruida sin imagen de fondo. 49
2.38. Orden de pintado de un escenario. 49
2.39. Problema en Algoritmo del Pintor. 50

2.40. Textura con mediana del color. 52
2.41. Textura con media del color. 52
2.42. Captura de la escena original. 53
2.43. Captura de los volúmenes reconstruidos y texturizados. 53
2.44. Distintas vistas, laterales y aéreas de la escena. 54
2.45. Tira de frames 1. 55
2.46. Tira de frames 2. 56
2.47. Tira de frames 3. 57
2.48. Tira de frames 4. 58
2.49. La misma escena reconstruida por distinto número de cámaras. 58
2.50. Geometría reconstruida vs geometría ideal. 59
2.51. Implementación hardware en el escenario de la mesa. 60
2.52. Implementación hardware en el escenario del espacio inteligente. 60
2.53. Flujograma del programa. 64
2.54. Comparación de ambos sistemas de coordenadas. 67
2.55. Detalles de la colocación del plano de fondo. 67
2.56. Situación del plano-volumen-cámara OpenSG. 67
2.57. Arquitectura hardware cliente-servidor. 69
2.58. Arquitectura software cliente-servidor. 69
2.59. Procesos del cálculo de la rejilla 3D. 71
2.60. Representación de tiempos consumidos. 71
2.61. Procesos del cálculo de la texturización basada en perspectiva inversa. 74
2.62. Representación de tiempos consumidos para la obtención de la corteza. 75
2.63. Representación de tiempos consumidos para el coloreado de la corteza. 75
2.64. Tiempo de obtención de corteza (Columna A) VS coloreado (Columna B). 76
2.65. Procesos principales en el Algoritmo del Pintor. 78
2.66. Consumo de tiempo de los procesos en el Algoritmo del Pintor. 78
2.67. Consumo de tiempos totales de la aplicación completa. 79
2.68. Arquitectura software cliente-servidor. 80
2.69. Ejemplo de refinado octree. 82
2.70. Mallado de una nube de puntos. 82
2.71. Refinado del mallado. 83

Capítulo 1

Resumen

1.1. Resumen

Este proyecto se encuentra entre los trabajos que van dirigidos a dotar a los espacios de inteligencia y más concretamente a la reconstrucción volumétrica de objetos con múltiples cámaras dentro de estos espacios. Para su realización se parte de una plataforma hardware y una estrategia software.

El sistema hardware, se compone de un conjunto de cámaras calibradas para la adquisición de datos y uno o varios computadores encargados del procesamiento y representación de los mismos. A lo largo del proyecto se detallan las posibles configuraciones y optimizaciones que se han realizado.

En el software, se trabaja bajo linux utilizando distintos tipos de librerías especializadas para cada tarea. Así se han desarrollado aplicaciones con OpenCV para la adquisición de imágenes y con OpenSG para generar los entornos virtuales, en los que se visualizarán las reconstrucciones de los volúmenes.

A lo largo de este trabajo, se afrontan algunos problemas tanto de programación como de hardware: la ubicación de los elementos sensoriales, la adquisición, acondicionamiento y procesamiento de datos, el conocimiento e integración de los periféricos, la gestión de herramientas y aplicaciones para optimizar los recursos del sistema, las formas mas adecuadas de reducir tiempos de computación etc ... son cuestiones tratadas y resueltas, exponiendo las posibles alternativas y la solución óptima final aplicada.

Palabras clave: visual hull, shape-from-silhouette, reconstrucción, 3D.

Capítulo 2

Memoria

2.1. Introducción.

El concepto de un espacio dotado de inteligencia ha sido propuesto por diversos autores y ha ido evolucionando separándose en distintas aplicaciones, incluyendo aquellas destinadas a la localización y navegación de robots, detección de usuarios y objetos. El objetivo fundamental de estos espacios es crear una interacción no intrusiva entre los usuarios y el entorno con la finalidad de ayudarle, prestándole un servicio en algún tipo de tarea, mediante una comunicación basada en un lenguaje natural.

Los elementos sensoriales en base a los cuales se establece la comunicación pueden ser muy diversos: micrófonos, ultrasonidos, sensores biométricos, infrarrojos, láser o como nos ocupa en este proyecto visión artificial.

Cuando una escena contiene objetos sólidos, es necesario considerar la naturaleza tridimensional de la misma, particularmente interesante en la visión artificial aplicada a la robótica, dada la información adicional que aporta el conocimiento de la totalidad de los volúmenes que ocupan el espacio, y su aplicación al cálculo de colisiones, trayectorias, etc... En las diferentes etapas de reconocimiento de la escena en tres dimensiones, se presentan más dificultades que en el caso de trabajar con imágenes bidimensionales. Estas dificultades son: la detección de las características de la escena en tres dimensiones, la segmentación, la influencia de la iluminación, el brillo, las sombras de la superficie que dependen de la orientación y por último la obtención de un volumen robusto y coloreado realista. Estas cuestiones serán abordadas y se intentarán resolver a lo largo del proyecto.

2.1.1. Objetivos y definición del problema.

En este proyecto se parte de un espacio inteligente acotado, dotado de un conjunto de cámaras calibradas, donde es necesario reconocer unívocamente las diferentes formas o volúmenes de manera precisa y rápida. El proceso de reconstrucción volumétrica de objetos en una escena implica la solución de los siguientes problemas:

- Extracción de información en cada una de las imágenes disponibles (segmentación).
- Correspondencia de la información entre diferentes cámaras y/o diferentes instantes de tiempo.
- Obtención de la rejilla de ocupación tridimensional de la geometría mediante homografías.
- Aplicación de un algoritmo de coloreado fotorealista.

La estrategia básica que se va a seguir es la explicada en la Figura 2.1. Se parte de varias vistas de diferentes cámaras de un mismo objeto, seguidamente se “segmentan”, con el fin de diferenciar el área del objeto a representar (en negro), del fondo (en blanco). El siguiente paso es aplicar la matriz de transformación H “Homografía”, que será explicada en el capítulo 2.3.3 y que permite obtener, tras fundir las informaciones de las distintas cámaras, el contorno del objeto a nivel del suelo.

Si mediante el proceso anterior se obtiene el contorno del volumen a nivel del suelo (cota z=0), incrementando esta cota con un valor conocido (z=Δh) y aplicando de nuevo el proceso, se obtiene un nuevo contorno a otra altura. Uniendo los contornos de las distintas alturas (Figura 2.2) obtenemos el volumen tridimensional.

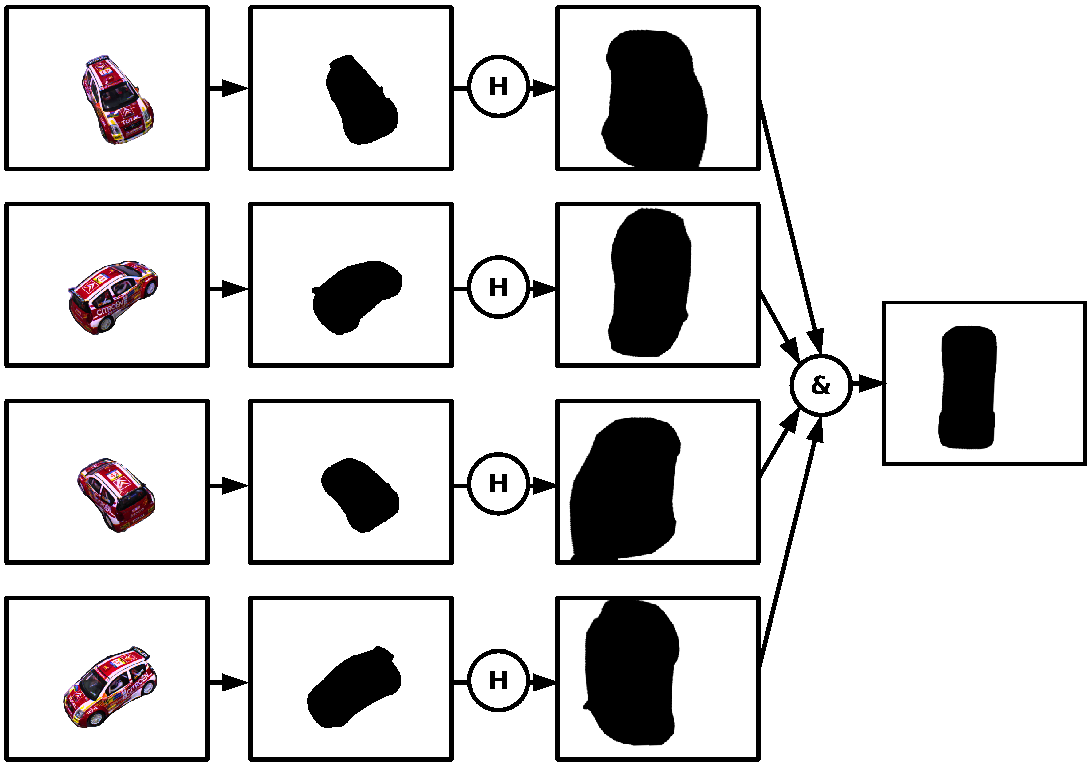

Figura 2.1: Proceso geométrico.

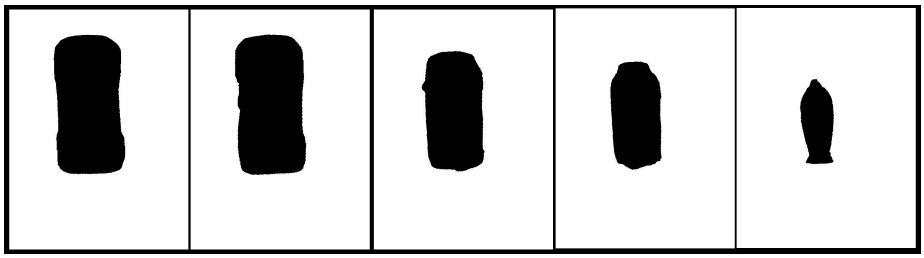

Figura 2.2: Muestra de contornos a diferentes alturas.

2.1.2. Estructura de la memoria.

Este documento se ha estructurado con la finalidad de dar una visión simple e intuitiva del problema que se aborda. Las distintas partes que lo componen son:

- **Estado del arte:** En este capítulo inicial, se hará un breve estudio de los trabajos previos de otros grupos de investigación en el campo de los espacios inteligentes y en el de la reconstrucción volumétrica de objetos. Se justificarán de esta forma la manera de abordar el problema y los métodos empleados.
- **Geometría de formación de la imagen:** Esta parte se dedica a las nociones teóricas y nomenclaturas adoptadas, con el fin de establecer todos los conceptos básicos y empezar a trabajar sobre ellos en capítulos posteriores.
- **Técnicas de segmentación de imágenes en cámaras estáticas:** El primer paso para poder reconstruir un objeto de una escena, es diferenciarlo del resto de elementos del fondo, se describirán las diversas técnicas que existen para segmentar imágenes así como la solución usada.
- **Sistema de reconstrucción volumétrica a partir de múltiples cámaras:** Detalla la manera de obtener la rejilla de ocupación tridimensional mediante homografías, y los algoritmos de coloreado realista empleados. Por último se propondrán distintos sistemas para refinar la geometría.
- **Reconstrucción volumétrica aplicada a espacios inteligentes:** En esta parte, se explica todo lo relativo a la aplicación desarrollada, encargada de realizar la reconstrucción volumétrica dentro del espacio inteligente, partiendo de cómo se ha distribuido el sistema hardware para la adquisición y control de múltiples cámaras y los aspectos software que incluyen lo referente a las herramientas utilizadas. Flujogramas de la aplicación, medidas de temporización de la misma y arquitecturas cliente servidor usadas.
- **Conclusiones y trabajos futuros:** Por último se resumen los objetivos conseguidos y sus respectivas conclusiones. Se proponen futuras líneas de investigación que se derivan del trabajo realizado.

2.2. Estado del arte.

En el campo de la reconstrucción de figuras en 3D con visión en entornos inteligentes, hay varios trabajos interesantes en los que se han utilizado diversas técnicas para optimizar, tanto la plataforma hardware en lo referente a la colocación de los elementos sensoriales, como la software incluyendo algoritmos o nuevas técnicas para optimizar los recursos.

2.2.1. Espacios inteligentes.

El inicio de los espacios inteligentes y la "Computación Ubicua" fueron conducidos por Weiser [1] [2]. Este autor realiza una clasificación en función de cuatro elementos básicos: ubicuidad, conocimiento, inteligencia e interacción natural.

- *Ubicuidad*: Múltiples sistemas embebidos con total interconexión entre ellos.

- *Conocimiento:* Habilidad del sistema para localizar y reconocer lo que acontece en el entorno y su comportamiento.
- *Inteligencia:* Capacidad de adaptarse al mundo que percibe.
- *Interacción Natural:* Capacidad de comunicación entre el entorno y los usuarios.

Uno de los primeros sistemas ubicuos implementados con esta filosofía se propuso en el proyecto Xerox PARC de Computación Ubicua (UbiComp) [3], desarrollado a finales de los 80. La red de sensores desplegada no estaba basada en información visual debido al coste prohibitivo para la época. Hoy en día varios grupos de investigación desarrollan y expanden el concepto de espacio inteligente y la computación ubicua. Algunos de los más notorios se indican a continuación:

- *Intelligent Room* [4]. Desarrollado por el grupo de investigación del *Artificial Intelligence Laboratory* en el MIT (Massachusetts Institute of Technology). Es uno de los proyectos más evolucionados en la actualidad. Se trata de una habitación que posee cámaras, micrófonos y otros sensores que realizan una actividad de interpretación con el objetivo de averiguar las intenciones de los usuarios. Se realiza interacción mediante voz, gestos y contexto.
- *Smart Room* [5]. Realizado en el *Media Lab* del MIT, utilizando cámaras, micrófonos y sensores se pretende analizar el comportamiento humano dentro del entorno. Se localiza al usuario, identificándolo mediante su voz y apariencia y se efectúa un reconocimiento de gestos. La línea de investigación actual es la capacidad del entorno para aprender a través del usuario que a modo de profesor le enseña nombres y apariencia de diferentes objetos.
- *Easy Living* [6]. Se trata de un proyecto de investigación de la empresa Microsoft con el objetivo de desarrollar "entornos activos" que ayuden a los usuarios en tareas cotidianas. Al igual que otros proyectos comentados se intenta establecer una comunicación entre el entorno y el usuario mediante lenguaje natural. Se hace uso de visión artificial para realizar identificación de personas e interpretación de comportamientos dentro del espacio inteligente.

Muchos de los proyectos mencionados no incluyen robots controlados por el entorno. El uso de agentes controlados por el espacio inteligente es estudiado por un grupo todavía reducido de laboratorios.

- T.Sogo [7] propone un sistema equipado con 16 cámaras fijas llamadas "Agentes de Visión" (VA), las cuales deberán aportar la información necesaria para localizar a un grupo de robots a través de un espacio lleno de obstáculos. Los Agentes de Visión consisten en cámaras omnidireccionales no calibradas. La localización se realiza en dos pasos. En primer lugar, un operador humano debe mostrar al espacio inteligente el camino que deberán seguir los robots. Una vez que el sistema ha aprendido en coordenadas de imagen el camino señalado, puede actuar sobre los robots, con el objetivo de minimizar el camino que recorren en la imagen con respecto al que realizan durante el período supervisado.

  La técnica es similar a la utilizada en aplicaciones de servo óptico, por lo que resulta muy difícil estimar la precisión que se obtiene realmente con el sistema. Dependerá en gran medida de la técnica de comparación en el plano imagen y la posición relativa de las cámaras y los robots.
- Un trabajo más evolucionado fue propuesto en la universidad de Tokyo por Lee y Hashimoto [8] [9]. En este caso se dispone de un Espacio Inteligente completo en el que

se mueven robots y usuarios. Cada dispositivo de visión es tratado como un dispositivo inteligente distribuido y conectado a una red (DIND). Tiene capacidad de procesamiento por sí solo, por lo que el espacio inteligente posee flexibilidad a la hora de incluir nuevos DIND. Puesto que se cuenta con dispositivos calibrados, la localización se realiza en espacio métrico. Los robots están dotados de balizamiento pasivo mediante un conjunto de bandas de colores fácilmente detectables por cada DIND. En este espacio de trabajo (Ispace), se realizan experimentos de interacción entre humanos y robots y navegación con detección de obstáculos.

- Otra propuesta es el proyecto MEPHISTO [10] del Instituto para el Diseño de Computadores y Tolerancia a Fallos en Alemania. En este proyecto el concepto de inteligencia distribuida se alcanza gracias a las denominadas Unidades de Procesamiento de Área Local (LAPU), similares a los DIND de Lee y Hashimoto. Se define una unidad específica para el control del robot (RCU) que deberá conectar y enviar instrucciones a los diferentes robots. La localización se realiza sin marcas artificiales, usando una descripción poligonal en coordenadas de imagen, la cual es convertida a una posición tridimensional con un enfoque multicámara.

- En el Departamento de Electrónica de la Universidad de Alcalá se desarrolla un proyecto de "espacio inteligente" [11] [12] [13] en el que mediante un conjunto de sensores se pretende realizar posicionamiento de robots móviles. Las alternativas incluyen visión artificial, ultrasonidos e infrarrojos. En dicho proyecto se enmarca la presente tesis que intenta seguir las líneas de investigación iniciadas, utilizando cámaras como sistema sensorial del espacio.

Los proyectos anteriormente comentados hacen hincapié en el diseño de sistemas distribuidos y flexibles, donde la inclusión de un nuevo sensor al entorno se realiza de forma dinámica, sin afectar al funcionamiento del sistema. Se han desarrollado trabajos, que si bien se diferencian en el planteamiento genérico, las técnicas usadas y el objetivo se relacionan con la tesis propuesta.

- Destaca entre otros el trabajo realizado por Hoover y Olsen [14], en el que utilizando un conjunto de cámaras con coberturas solapadas, son capaces de detectar obstáculos estáticos y dinámicos. Mediante un mapa de ocupación [15] es posible hacer navegar a un robot dentro del espacio de cobertura conjunta.

  La técnica del mapa de ocupación requiere una fase previa de calibración [16] en la cual el espacio se encuentra libre de obstáculos. Se obtienen imágenes de referencia que serán utilizadas para la localización de obstáculos en posteriores capturas.

  Mediante tablas de búsqueda se logra una correspondencia directa entre los puntos de pantalla y las coordenadas tridimensionales de puntos pertenecientes al plano que representa el suelo del espacio. Y así, si se realiza un fundido de la información de las distintas cámaras, se consigue un mapa de aquellos puntos del suelo que tienen alta probabilidad de estar ocupados.

  Uno de los principales inconvenientes de este enfoque, es la detección de objetos con bajo contraste con respecto al color del suelo en la imagen de referencia. El sistema desperdicia mucha información tridimensional mediante el mapa de ocupación, que podría ser utilizada para obtener información tridimensional del entorno y de los objetos que se mueven en él.

- Un proyecto similar al anterior es el presentado por Kruse y Whal [17] denominado MONAMOVE (Monitoring and Navigation for Mobile Vehicles), orientado a la navegación de un vehículo en un entorno industrial. En este trabajo se propone fundir la información

de localización obtenida del conjunto de cámaras distribuidas por el entorno, con sensores de ultrasonidos e infrarrojos equipados a bordo del robot. Al igual que el trabajo de Hoover y Olsen se utilizan tablas de búsqueda para crear un mapa de ocupación del entorno.

Los obstáculos móviles se detectan mediante análisis diferencial en sucesivas imágenes y los estáticos son comparados con dos imágenes de referencia, conseguidas en un proceso previo de calibración. Dichas imágenes de referencia se toman bajo dos condiciones de iluminación diferentes, y se realiza una comparación ponderada con la imagen actual, para obtener regiones correspondientes a objetos que existen en el entorno.

2.2.2. Técnicas de reconstrucción volumétrica a partir de múltiples cámaras.

El término "Shape-From-Silhouette"(SFS) es un método de reconstrucción de la silueta de un objeto en 3D a partir de los contornos de varias imágenes de este objeto. La idea de usar siluetas para una reconstrucción en 3D fue introducida por Baumgart en 1974 [18]. Baumgart estimó los contornos 3D de un muñeco y de un caballo de juguete a partir de 4 imágenes de su silueta. Partiendo de él, han sido propuestas diferentes variaciones del paradigma SFS.

Por ejemplo, Aggarwal [19] [20] usó descripciones volumétricas para representar la forma reconstruida. Potmesil [21], Noborio [22] y Ahuja [23] sugirieron usar una estructura de datos octree para acelerar el SFS. Shanmukh y Pujari consiguieron la posición y dirección óptimas de las cámaras para obtener siluetas de un objeto para una reconstrucción de la forma [24]. Szeliski construyó un digitalizador no invasivo usando una mesa giratoria y una sola cámara con el SFS como procedimiento de reconstrucción [25]. En resumen, el SFS se ha convertido en uno de los métodos más populares para la reconstrucción de objetos estáticos.

El término "Visual Hull" (V.H) es usado por los investigadores para denotar la forma estimada usando el principio SFS: El término fue acuñado en 1991 por Laurentini [26] quien también publicó una serie de papers estudiando los aspectos teóricos del Visual Hull aplicado a objetos poliédricos [27] [28] y curvos [29].

Calcular la forma utilizando SFS tiene muchas ventajas. La primera es, que las siluetas se consiguen con facilidad, en especial en entornos cerrados (espacios inteligentes) donde las cámaras son estáticas y el entorno está controlado (iluminación, brillos, sombras...). La aplicación de la mayoría de los métodos SFS, son también relativamente directos, en especial si los comparamos con otros métodos de valoración como el multi-baseline stereo [30] o el space carving [31]. Además la inherente propiedad conservativa de la forma estimada usando SFS es particularmente útil en aplicaciones como evitar obstáculos en la manipulación de robots y en el análisis de la visibilidad en navegación. Estas ventajas han provocado que un gran número de investigadores apliquen SFS para solucionar problemas gráficos y de visión artificial. Como por ejemplo aplicaciones relacionadas con la interacción humana: digitalización humana virtual [32], estimación de la forma humana [33], tracking/captura de movimiento [34] [35] y renderizado basado en imágenes [36].

Por otro lado, el SFS adolece de la limitación de que la forma estimada (el Visual Hull) puede ser muy burda cuando tenemos pocas siluetas de la forma, esto se nota de manera especial para objetos de cierta complejidad. Para hacer una estimación mejor de este tipo de escenas hay que incrementar el número de distintas siluetas del objeto. La forma más común de conseguirlo es haciendo una aproximación "a través del espacio", es decir, incrementando físicamente el número de cámaras utilizadas. Esta aproximación, muy simple, quizás no puede ser siempre factible debido a situaciones prácticas específicas o limitaciones físicas. En otra línea también hay trabajos [37] para hacer una aproximación "a través del tiempo", incrementando el número

de siluetas efectivas, haciendo distintas capturas en distintos instantes de tiempo (cuando el objeto se está moviendo).

2.3. Geometría de formación de la imagen.

En esta sección se hará un breve resumen sobre la notación, teoría y modelos que se van a utilizar, para una vez establecidos todos los conceptos clave poder trabajar sobre ellos.

- **Notación y Escenario.** Se supone que se tienen K cámaras posicionadas alrededor de un objeto O. Entonces $S_j^k; k = 1, 2...K$, son el conjunto de siluetas del objeto O, conseguidas a partir de K cámaras en el instante t_j. Un ejemplo de escenario es el descrito en la Figura 2.3 con un objeto en forma de cabeza rodeado por cuatro cámaras en el instante t_1.

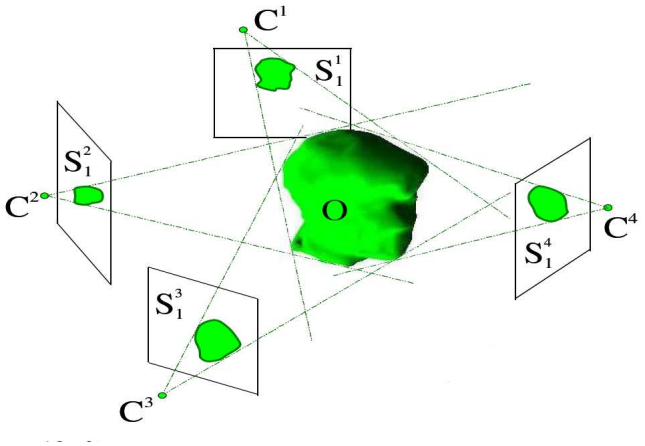

Figura 2.3: Escenario: Objeto O forma la silueta S_1^k en la cámara k en el instante t_1.

Se asume que las cámaras están calibradas con $\Pi^k() : R^3 \rightarrow R^2$ y C^k es la función perspectiva de la cámara k. En otras palabras $p = \Pi^k(P)$ son las coordenadas 2D de un punto 3D P en la k-ésima imagen. Como extensión de esta notación, $p = \Pi^k(A)$ representa la proyección del volumen A en el plano de la imagen de la cámara k. Se asume que se tiene un conjunto de K siluetas S_j^k y funciones de proyección $\Pi^k()$. Un volumen A de un objeto O se dice que está perfectamente definido si y sólo si su proyección en el k-ésimo plano coincide de manera exacta con la silueta de la imagen S_j^k para todo $k \in 1, ..., K$. Por ejemplo $\Pi^k(A) = S_j^k$. Si existe por lo menos un volumen no vacío que define las siluetas, se dice que el conjunto de siluetas de la imagen es consistente, de lo contrario es inconsistente.

2.3.1. Modelo de cámara Pin-Hole.

Para el conjunto de cámaras utilizadas en este proyecto, se ha utilizado un modelo de geometría proyectiva de cámara pin-hole que se muestra en la Figura 2.4. Este modelo, se usa para obtener las proyecciones de los puntos del espacio 3D sobre los planos de las imágenes 2D. En este modelo, el punto tridimensional *P(X,Y,Z)* es proyectado sobre el plano de la imagen pasando a través del centro óptico localizado en el plano focal. La recta que une el punto P y el centro óptico se llama línea de proyección e intersecta al plano imagen justo en el píxel *p(x,y)*, el cual es la proyección correspondiente de *P(X,Y,Z)*.

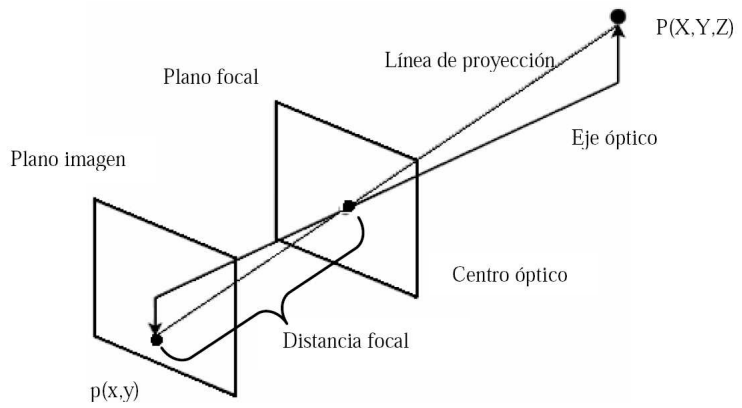

Figura 2.4: Modelo Pin Hole.

El centro óptico está situado en el centro del plano focal. Este modelo se completa con el eje óptico, el cual es una línea perpendicular al plano imagen, que atraviesa el centro óptico y el plano focal, que es el plano perpendicular al eje óptico. A partir de este modelo, y usando las reglas de la geometría, obtenemos las siguientes ecuaciones que relacionan el punto espacial P(X,Y,Z) con su correspondiente proyección el punto p(x,y) en la imagen:

$$y = f \cdot \frac{Y}{Z}$$
$$x = f \cdot \frac{X}{Z}$$

En la práctica, el sistema de coordenadas del mundo y de la cámara están relacionados por un conjunto de parámetros físicos, como la distancia focal de las lentes, el tamaño del píxel, la posición del origen de coordenadas, y la posición y orientación de las cámaras. Estos parámetros se pueden agrupar en dos tipos:

- **Parámetros intrínsecos:** Son aquellos que especifican unas características propias de la cámara: estos parámetros incluyen la distancia focal, que es, la distancia entre la lente de la cámara y el plano sensor, la localización en la imagen de su centro en pixels, el tamaño efectivo del píxel, y los coeficientes de distorsión radial de la lente.

  Es posible asociar con la cámara un *plano imagen normalizado* paralelo a su plano imagen pero localizado a una distancia unitaria del pihole. Se asocia a este plano su propio sistema de coordenadas con un origen localizado en $\hat{C}$ (Figura 2.5). La proyección perspectiva puede ser escrita en este sistema de coordenadas normalizado como:

  $$\begin{Bmatrix} \hat{u} = \frac{x}{z} \\ \hat{v} = \frac{y}{z} \end{Bmatrix} \Longleftrightarrow \hat{p} = \frac{1}{z}(I_d 0)\begin{pmatrix} P \\ 1 \end{pmatrix}, \tag{2.1}$$

  donde $\hat{p} = (\hat{u}, \hat{v}, 1)^T$ es el vector de coordenadas homogéneas de la proyección $\hat{p}$ del punto P en el plano imagen normalizado. El plano imagen de la cámara es en general diferente (Figura 2.5). Está localizada a distancia $f \neq 1$ del pinhole, y las coordenadas de la imagen (u,v) del punto de la imagen p son por lo general expresadas en unidades de pixels (en vez de unidades métricas). Además, los pixels son normalmente rectangulares (no cuadrados), así que la cámara tiene dos parámetros de escala adicionales: k y l:

  $$\begin{Bmatrix} u = kf\frac{x}{z}, \\ v = lf\frac{y}{z}. \end{Bmatrix} \tag{2.2}$$

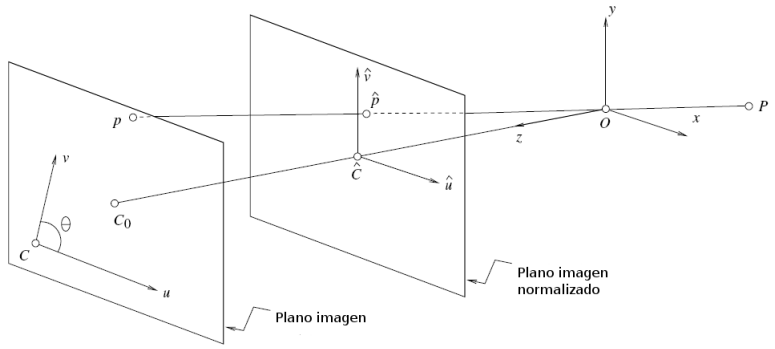

Figura 2.5: Sistema de coordenadas del plano imagen y el plano imagen normalizado.

Nota: f es una distancia expresada en metros, por ejemplo. Y un píxel tiene dimensiones $\frac{1}{k} \times \frac{1}{l}$, donde k, l, y f no son independientes, y pueden ser reemplazadas por las notaciones amplificadas $\alpha = kf$ y $\beta = lf$ expresadas en unidades de pixels.

En general, el origen del sistema de coordenadas de la cámara está situado en la esquina C del plano imagen, bien en la esquina inferior izquierda o también puede ir en algunos casos en la esquina superior izquierda, cuando las coordenadas de la imagen son los índices de la fila y columna de un píxel y no en el centro. El centro de la matriz del CCD normalmente no coincide con el punto principal C_0 hace que se añadan dos parámetros u_0 y v_0 que definen la posición (en unidades de píxel) de C_0 en el sistema de coordenadas del plano imagen. De esta manera la ecuación anterior es reemplazada por:

$$\begin{Bmatrix} u = \alpha\frac{x}{z} + u_0, \\ v = \beta\frac{y}{z} + v_0 \end{Bmatrix} \tag{2.3}$$

Por último, el sistema de coordenadas de la cámara se ve afectado por algunos errores de fabricación, el ángulo θ que se forma entre los dos ejes de la imagen no son iguales a 90 grados. En este caso la ecuación anterior se transforma en:

$$\begin{Bmatrix} u = \alpha\frac{x}{z} - \alpha cot\theta\frac{y}{z} + u_0, \\ v = \frac{\beta}{sin\theta}\frac{y}{z} + v_0. \end{Bmatrix} \tag{2.4}$$

Combinando las ecuaciones anteriores, podemos realizar un cambio de coordenadas entre la imagen física y la normalizada como una transformación afin planar: Combinándolas obtenemos:

$$p = \kappa\hat{p} \text{ , donde } p = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \text{ , y } \kappa = \begin{bmatrix} \alpha & -\alpha cot\theta & u_0 \\ 0 & \frac{\beta}{sin\theta} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

Poniéndolas todas juntas, se obtiene:

$$p = \frac{1}{z}MP \text{ , donde } M = \begin{bmatrix} \kappa & 0 \end{bmatrix} \tag{2.6}$$

y $P = (x, y, z, 1)^T$ es el vector de coordenadas homogéneas P en el sistema de coordenadas de la cámara. En otras palabras, las coordenadas homogéneas pueden ser usadas para representar la proyección perspectiva mapeada por la matriz M (3x4).

Nota: El tamaño físico y el desplazamiento de los pixels son siempre fijados por la cámara usada y el hardware del sistema, que en principio, pueden ser medidos durante el proceso de fabricación (aunque esta información no está disponible). Para lentes con zoom, la longitud focal puede variar a lo largo del tiempo de uso, junto con el centro de la imagen, cuando el eje óptico de las lentes no es exactamente perpendicular al plano de la imagen.

- **Parámetros extrínsecos:** Los parámetros extrínsecos de la cámara describen la relación espacial entre la cámara y el mundo, especificando la transformación entre el eje de coordenadas de la cámara y la referencia de imágenes del mundo. Estos parámetros son la matriz de rotación y el vector de traslación.

  Considerando que el marco de la cámara (C) es distinto del marco del mundo (W), y expresándolo como:

$$\begin{bmatrix} {}^{C}P \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{C}_{W}R & {}^{C}O_{W} \\ 0^{T} & 1 \end{bmatrix} \cdot \begin{bmatrix} {}^{W}P \\ 1 \end{bmatrix} \tag{2.7}$$

  y sustituyendo en la ecuación 2.6 obtenemos:

$$p = \tfrac{1}{z} MP \text{ , donde } M = \kappa \begin{bmatrix} R & t \end{bmatrix} \tag{2.8}$$

  $R = R^{c}_{w}$ es la matriz de rotación, $t =^{c} O_{w}$ es el vector de traslación, y $P = (w_x, w_y, w_z, 1)^T$ es el vector de coordenadas homogéneas del vector P en el marco del mundo (W). Esta es la manera más general de expresar la ecuación de proyección perspectiva. Se puede usar para determinar la posición del centro óptico de la cámara O en el sistema de coordenadas del mundo. Además, el vector de coordenadas homogéneas O, verifica MO=0. En particular si M = (A b), donde A es una matriz no singular 3x3 y b es un vector en R^3, el vector de coordenadas homogéneas del punto O es $-A_{-1}b$. Es importante comprender que la profundidad de z en la ecuación 2.8 no es independiente de M y P ya que, si m_1^T, m_2^T y m_3^T son las tres filas de M, sacado directamente de la ecuación 2.8 donde $z = m_3$ P. De hecho se puede reescribir la ecuación 2.8 en su forma equivalente:

$$\begin{Bmatrix} u = \frac{m_1 P}{m_3 P}, \\ \\ v = \frac{m_2 P}{m_3 P}. \end{Bmatrix} \tag{2.9}$$

  Una matriz de proyección puede ser descrita explícitamente como una función de sus cinco parámetros intrínsecos (α, β, u_0, v_0 and θ) y sus seis extrínsecos (los tres ángulos que definen R y las tres coordenadas de t):

$$M = \begin{bmatrix} \alpha r_1^T - \alpha cot\theta r_2^T + u_0 r_3^T & \alpha t_x - \alpha cot\theta t_y + u_0 t_z \\ \frac{\beta}{sin\theta} r_2^T + v_0 r_3^T & \frac{\beta}{sin\theta} t_y + v_0 t_z \\ r_3^T & t_z \end{bmatrix} \tag{2.10}$$

  Donde r_1^T, r_2^T y r_3^T son las tres filas de la matriz R y t_x, t_y y t_z son las tres coordenadas del vector t. Si R es escrita como el producto de las tres rotaciones elementales, los vectores r_i (i=1,2,3) pueden ser escritos de forma explícita en términos de los correspondientes tres ángulos.

2.3.2. Calibración de un conjunto de cámaras.

Es el proceso de estimar los parámetros intrínsecos y extrínsecos de las cámaras. Una de las técnicas más comunes para calibrar es el método lineal de Tsai, que se caracteriza por ser rápido

aunque quizás para según que tipo de aplicaciones puede ser poco preciso. Para automatizar este proceso se utiliza la toolbox de Matlab para calibración de cámaras, consiste en una serie de funciones que permiten calcular la totalidad de los ocho parámetros intrínsecos: factor de escala, longitud focal efectiva, coordenadas del punto principal, dos coeficientes de distorsión radial y dos coeficientes de distorsión tangencial. La corrección de la imagen es realizada usando un modelo inverso de distorsión especial.

Para usar la toolbox se necesita un patrón de calibración con un conjunto de puntos de control visibles de tamaño y posición conocidos previamente. Como datos de entrada en el proceso de calibración se utilizan las coordenadas 3D de los puntos de control y las coordenadas correspondientes de la imagen observada. Para lograr una calibración más precisa, se suele mostrar el patrón a la cámara desde distintas posiciones, variando la distancia y ángulo de este (Figura 2.6). En este proceso no es necesario que la totalidad de los puntos de control del patrón sean visibles en todo momento.

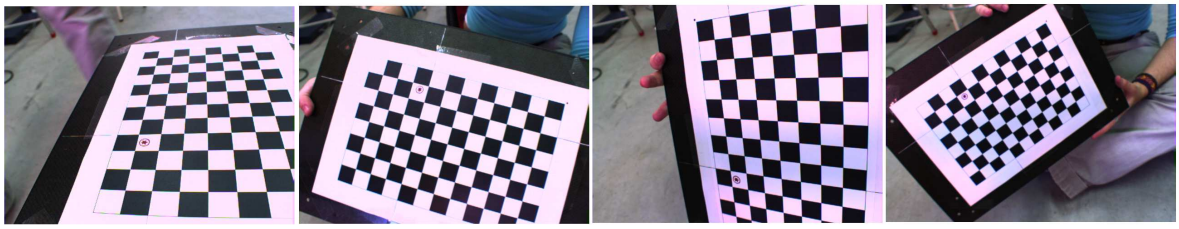

Figura 2.6: Patrón en el proceso de calibración.

Realizando el procedimiento anteriormente explicado se calibra cada cámara por separado, obteniendo los parámetros intrínsecos de cada cámara independiente. A continuación una vez ubicadas las cámaras en el espacio inteligente en su posición definitiva, se coloca un patrón en el suelo como se muestra en la Figura 2.7 y se calculan los parámetros extrínsecos de forma análoga a como se calcularon los intrínsecos con la toolbox de Matlab.

Figura 2.7: Calibración parámetros extrínsecos.

2.3.2.1. Escenarios.

En este proyecto se presentan dos posibles escenarios en donde aplicar la reconstrucción, el primero está formado por una única cámara situado alrededor de una mesa giratoria, el segundo es el propio “espacio inteligente”, formado por un conjunto de 4 cámaras colocadas alrededor de un área dentro del cual se sitúan las figuras a reconstruir. Ambos escenarios se explican con detalle a continuación:

La mesa giratoria (primer escenario), puede realizar giros en torno a un eje central en incrementos de un grado (Figura 2.9). Alrededor de la mesa se sitúa una única cámara como se muestra en la Figura 2.8, esta aplicación es muy útil para probar que los algoritmos y la aritmética matemática funcionan de forma correcta, para más tarde aplicarlos al espacio real, puesto que es un entorno muy controlado y robusto.

Figura 2.8: Mesa giratoria, imagen real.

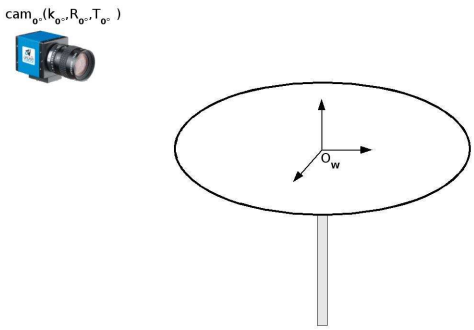

Figura 2.9: Mesa giratoria, situación de captura.

El principio de funcionamiento se basa como se observa en la Figura 2.10, en dejar fijo el objeto estático sobre la mesa y la cámara. Después se hace girar la mesa en torno a su eje en incrementos de grados constantes, capturando la escena en cada incremento. De esta manera se obtienen una serie de vistas como se ejemplifica en la Figura 2.11. En este sistema la relación entre el origen de coordenadas del mundo O_w y el de la cámara en cada incremento, obedece a una variación en la matriz de traslación, manteniéndose constante el conjunto de parámetros intrínsecos (la cámara es la misma) y la matriz de traslación.

El espacio inteligente (segundo escenario) que se ha utilizado como se indica en la Figura 2.12, consiste en un área acotada controlada (se pude actuar sobre los elementos de iluminación) rodeada por una serie de cámaras estáticas (en nuestro caso 4), enfocadas y calibradas de manera adecuada. En este sistema, cada cámara tiene su propio origen del mundo O_i donde i=0,1... hasta K cámaras, que se relacionan de manera individual con el origen del mundo real O_w con sus parámetros intrínsecos (K_i) y matrices de traslación (T_i) y rotación(R_i).

2.3.3. Homografías.

Como se muestra en la Figura 2.13, la correspondencia entre un punto p con coordenadas (u,v) en el plano imagen I, y el punto X con coordenadas (x,y) en el plano π que se caracteriza

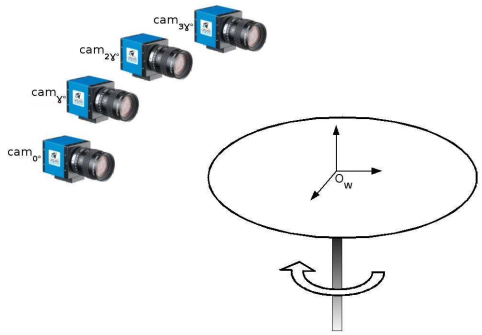

Figura 2.10: Mesa giratoria, múltiples capturas.

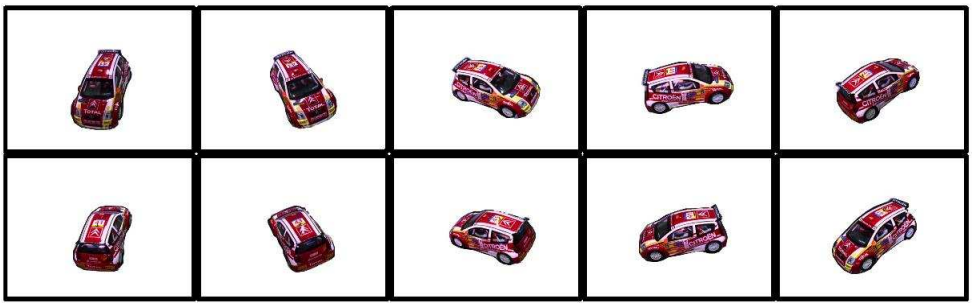

Figura 2.11: Captura con un incremento de rotación de 36 grados.

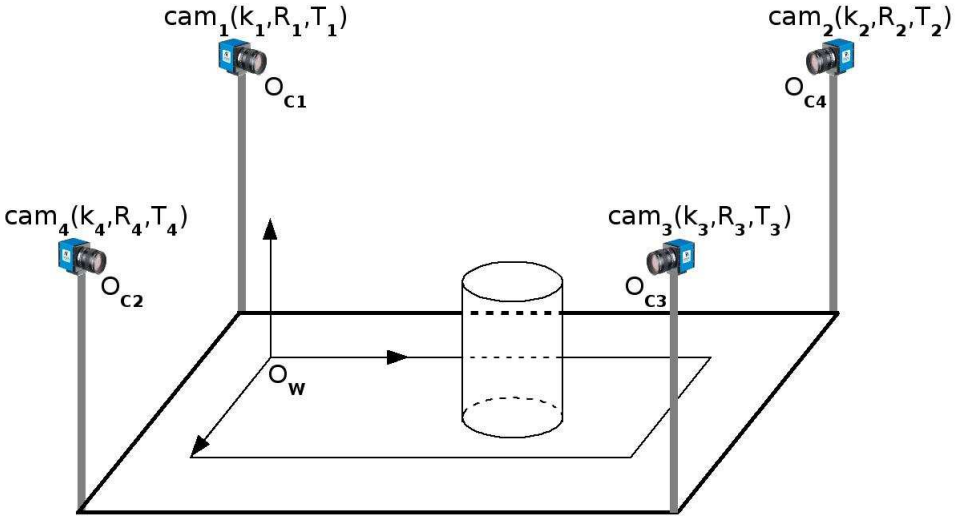

Figura 2.12: Espacio Inteligente.

porque todos sus puntos tienen cota cero (z=0), es la transformación H u "Homografía", que se puede escribir de la forma:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \lambda \cdot k(R_c X + T_c) \tag{2.11}$$

y expresado en función de H:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \lambda \cdot H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.12}$$

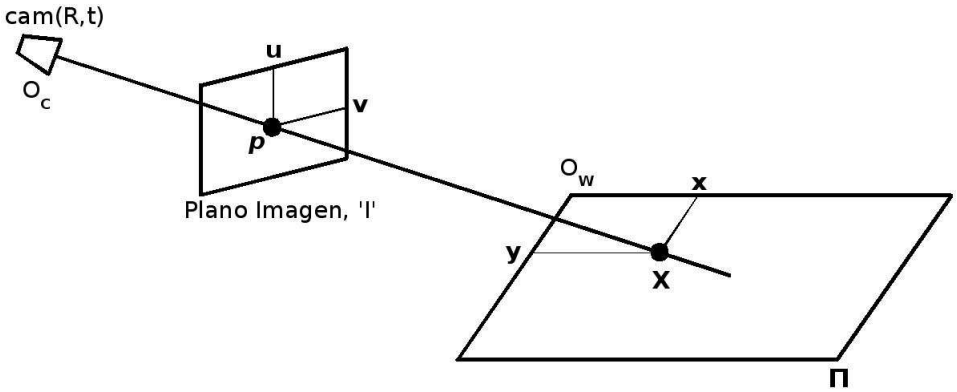

Figura 2.13: Escena cámara-plano Homografía.

La función H tiene inversa, por lo que la transformación también se puede escribir como:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \lambda^{-1} \cdot H^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{2.13}$$

Una vez establecidas las ecuaciones básicas, un punto cualquiera del plano π, dado que todos sus puntos tienen z=0, quedaría descrito:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \lambda \cdot k(R_c \begin{bmatrix} x \\ y \\ z \end{bmatrix} + T_c) \text{ para z=0 } \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \lambda \cdot k(R_c \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} + T_c) \tag{2.14}$$

Desarrollando la expresión tenemos:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \lambda \cdot ((k \cdot R_c) \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} + (k \cdot T_c)) \tag{2.15}$$

donde se definen $P_1 = (k \cdot R_c)$ y $T_1 = (k \cdot T_c)$:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \lambda \cdot (P_1 \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} + T_1) \tag{2.16}$$

Suponiendo que P_1 se puede escribir como $P_1 = (k \cdot R_c) = (c_1 c_2 c_3)$, tenemos la ecuación como:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \lambda(c_1 \cdot x + c_2 \cdot y + T_1) \tag{2.17}$$

o lo que es lo mismo:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \lambda \cdot (c_1 c_2 T_1) \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.18}$$

que igualándolo a la ecuación 2.12, nos da como resultado la matriz transformación $H = (c_1 c_2 T_1)$

Transformaciones de escala: La imagen origen con la que se va a trabajar tiene una escala de pixels, que tras aplicar la transformación "H" se va a escalar en milímetros, por lo tanto se necesita una segunda transformación "M" a la salida de la cual se obtiene de nuevo una escala en pixels, en la que se consigue una correspondencia conocida entre pixels y tamaño en milímetros (Figura 2.14).

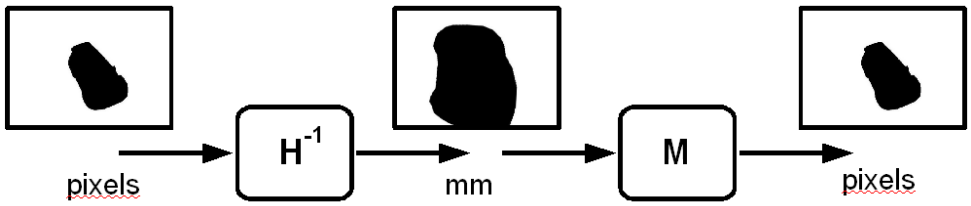

Figura 2.14: Diagrama de escalado.

Este escalado se realiza mediante la matriz M que se introduce de la siguiente forma:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \lambda \cdot H \cdot M \cdot \begin{bmatrix} u_g \\ v_g \\ 1 \end{bmatrix} \tag{2.19}$$

con:

$$M = \begin{bmatrix} Ah & 0 & T_1 \\ 0 & Ah & T_2 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.20}$$

2.4. Técnicas de segmentación de imágenes en cámaras estáticas.

El problema general de la segmentación de imágenes consiste en localizar los diferentes objetos que están presentes en una imagen. En un sistema de visión artificial genérico, la fase de segmentación se encuadra entre las fases de preprocesamiento, en la que las imágenes son sometidas a diferentes operaciones de filtrado que ayuden a mejorar la calidad de la imagen o a destacar los objetos que posteriormente queremos segmentar, y la de identificación o reconocimiento de los objetos, que es en la fase en la que se reconocen y reconstruyen volumétricamente los objetos.

Para esta tarea, como se puede observar en la Figura 2.15 partimos de una imagen estática tomada por una cámara del fondo de la imagen F(x), entendiendo como fondo todos los elementos relativamente estáticos que quedan dentro de la zona de trabajo, a este fondo le llamaremos "entrenamiento" y una "imagen de entrada" I(x) en la que además del fondo de la imagen se tiene el objeto a identificar. Tras aplicar el proceso de segmentación obtendremos una imagen de salida S(x) en la que se conseguirá únicamente el objeto a reconocer diferenciado (en negro) de una manera clara del resto de la escena (en blanco).

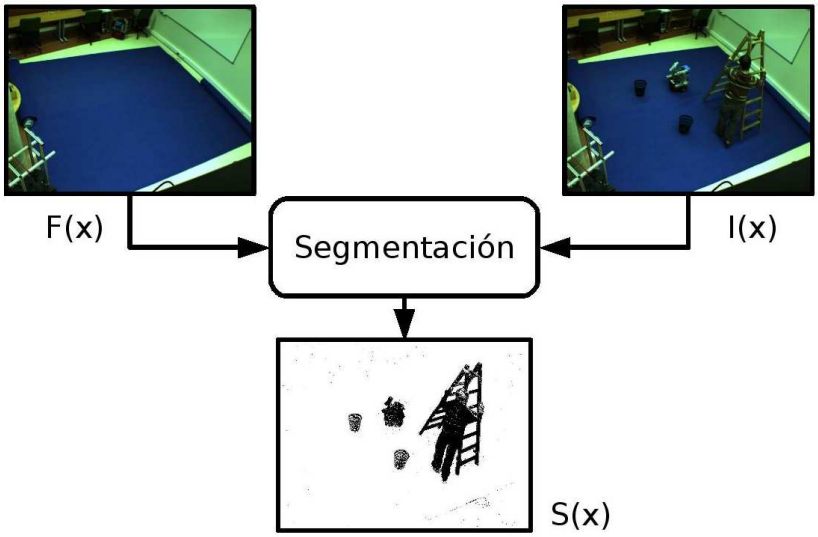

Figura 2.15: Diagrama del proceso de segmentación simple.

Existen diferentes métodos de segmentación. A continuación se explicarán en orden de complejidad los que se pueden utilizar:

2.4.1. Método sencillo.

Partiendo del diagrama propuesto en la Figura 2.15 se realiza un umbralizado basado en la diferencia entre la imagen de fondo y la imagen de entrada $\|F(x) - I(x)\|$, es decir:

$$\|F(x) - I(x)\| > \tau \longrightarrow S(x) = 0 \tag{2.21}$$

$$\|F(x) - I(x)\| \leq \tau \longrightarrow S(x) = 1 \tag{2.22}$$

El principal problema de este algoritmo es la aparición de ruido en la imagen de salida, para eliminarlo se suelen aplicar operadores morfológicos (clean, bwmorph...). Dada la escasa robustez del algoritmo, puesto que se ve muy afectado por los cambios de iluminación en la imagen de entrada y la dificultad para ajustar el umbral τ, pues es muy difícil encontrar el balance entre eliminar el ruido completamente excluyendo también el objeto a reconocer, y mantener la forma del objeto a segmentar conservando también el ruido de la escena, se opta por calcular el valor de τ en función del píxel 'x':

2.4.1.1. Umbral adaptativo según x.

A diferencia del método sencillo en que el valor de salida de la imagen se calcula con un valor estático de τ, en este método τ se modela como $\tau(x)$ obteniéndose el valor de la imagen

de salida de la siguiente manera:

$$\|F(x) - I(x)\| > \tau(x) \longrightarrow S(x) = 0 \tag{2.23}$$
$$\|F(x) - I(x)\| \leq \tau(x) \longrightarrow S(x) = 1 \tag{2.24}$$

El principal problema de usar un umbralizado adaptativo con $\tau(x)$, es que la función entre $\tau(x)$ y F(x) no se conoce, por lo que se recurren a otras técnicas más avanzadas como la segmentación mediante distancia estadística.

2.4.2. Segmentación mediante distancia estadística.

Para este método no partimos de una única imagen estática de fondo, sino que tenemos múltiples fondos, en lo que hay cambios en la iluminación, brillos, sombras, etc... que someterán al sistema a un entrenamiento con el objetivo de obtener la *media* y la *varianza* del fondo y mejorar el umbralizado, como se muestra en la Figura 2.16:

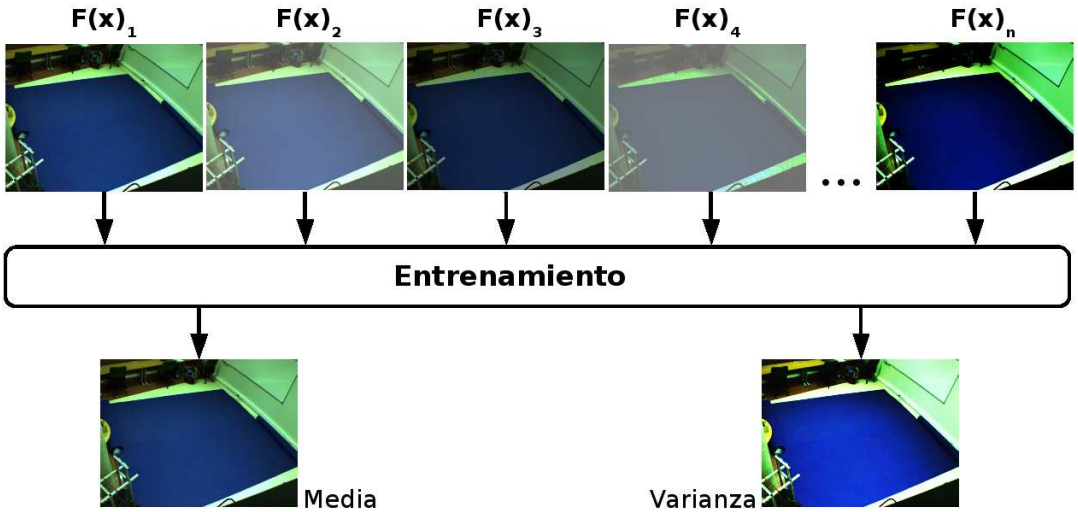

Figura 2.16: Diagrama del proceso de segmentación mediante distancia estática.

A partir de la *media* que se define como:

$$\hat{F}(x) = \frac{1}{N}\Sigma F(x) \tag{2.25}$$

y la *varianza* que se define como:

$$P_{F(x)} = \frac{1}{N}\Sigma(F(x) - \hat{F}(x)) \cdot (F(x) - \hat{F}(x))^t \tag{2.26}$$

donde para imágenes RGB es:

$$P_{F(x)} = \begin{bmatrix} \sigma_R^2 & & \\ & \sigma_G^2 & \\ & & \sigma_B^2 \end{bmatrix} \tag{2.27}$$

coincidiendo para las imágenes en escala de grises con el valor de la varianza al cuadrado: $P_{F(x)} = \sigma^2$.

"Distancia de Mahalanobis": Sea F(x) una variable aleatoria gaussiana (Figura 2.17) con Media $\hat{F}(x)$ y Varianza $P_{F(x)}$, y sea x una realización del proceso, entonces la distancia de Mahalanobis de x a la distribución F(x) es:

$$(I(x) - \hat{F}(x))^t \cdot \Sigma F(x)^{-1} \cdot (I(x) - \hat{F}(x)) = \chi^2(N)$$

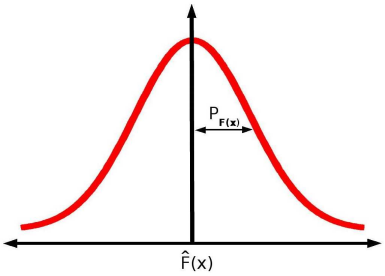

Figura 2.17: Variable aleatoria gaussiana F(x).

El resultado obtenido de un conjunto de frames utilizando este método es el mostrado en la Figura 2.18, donde se observan distintos momentos en el espacio inteligente con varios objetos en movimiento.

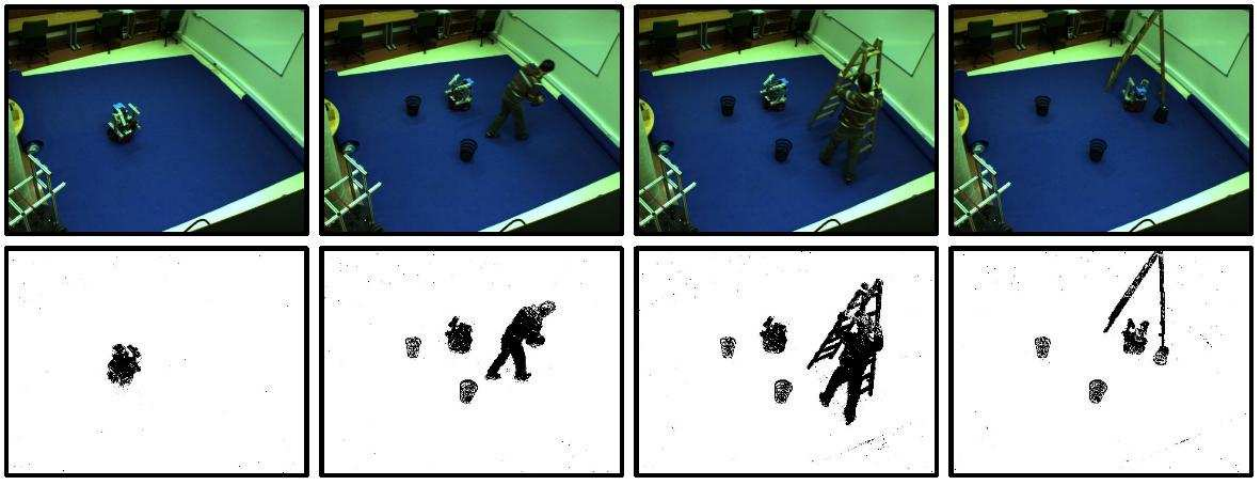

Figura 2.18: Ejemplo de resultado de segmentación utilizada.

2.4.3. Métodos avanzados de segmentación.

Referencias a trabajos más avanzados en el campo de la segmentación de imágenes con cámaras estáticas.

2.5. Sistema de reconstrucción volumétrica a partir de múltiples cámaras.

En este capítulo se detalla la manera de obtener la rejilla de ocupación tridimensional mediante homografías. Para lo cual es necesario previamente desarrollar las bases teóricas (Visual

Hull) y sus limitaciones. Seguidamente se explican los distintos algoritmos de coloreado realista estudiados y empleados. Se indican para todos ellos ejemplos de los resultados obtenidos así como los pros y contras de su uso.

2.5.1. Obtención de una rejilla de ocupación 3D mediante homografías.

Partimos de un escenario genérico, ya descrito en "Geometría de Formación de la Imagen" y más concretamente en la Figura 2.3, disponemos de K cámaras posicionadas alrededor de un objeto O. Siendo $S_j^k; k = 1, 2...K.$ son el conjunto de siluetas del objeto O obtenidas a partir de K cámaras en el instante t_j. A partir del Visual Hull obtenido a partir de la transformación Homográfica y que a continuación se describe detalladamente, se obtendrá la rejilla de ocupación.

2.5.1.1. Visual Hull.

Hay varias maneras de definir el Visual Hull, que es la reconstrucción del volumen estimado a partir del cálculo de las siluetas del objeto obtenidas mediante la Homografía. En este trabajo se van a exponer dos definiciones, que demuestran además las propiedades más importantes que se han aplicado. A pesar de que las dos definiciones son significativamente diferentes, son de hecho equivalentes (ver [38] para más detalles).

2.5.1.1.1. Definición I de V.H: Intersección de los conos visuales. El Visual Hull H_j con respecto a un conjunto consistente de siluetas de imágenes S_j^k, se define como la intersección de los K conos visuales, cada uno de ellos formado por la proyección de la silueta de la imagen S_j^k en el espacio 3D a través del centro de la cámara C^k. Esta primera definición, que es la más utilizada, define el Visual Hull como la intersección de los conos visuales formados por los centros ópticos de las cámaras y la silueta del objeto 3D. Esta definición nos aporta una manera directa para computar el V.H a partir de las siluetas aunque carece de información y una visión intuitiva del objeto (que forman las siluetas).

2.5.1.1.2. Definición II de V.H: Volumen máximo. El visual Hull H_j con respecto a un conjunto de siluetas S_j^k se define como el volumen máximo que encierra S_j^k para k=1,2,...K. De forma general, para un conjunto de siluetas S_j^k, hay un número infinito de volúmenes (entre los que se encuentra el objeto O) que encierran las siluetas. Con esta segunda definición se fija el V.H como el más grande de estos volúmenes, expresando una de sus cualidades más importantes, el Visual Hull es el volumen convexo máximo que engloba a todas las siluetas, asegurando que toda la geométrica del objeto a representar estará incluido dentro del V.H.

2.5.1.1.3. Construcción de Visual Hull.

- **Representación basada en 2D.** A partir de un conjunto consistente de imágenes, el V.H puede ser (acorde con la Definición I) construido directamente por la intersección de los conos visuales. Haciéndolo, el V.H se representa como parches en una superficie 2D obtenidos a partir de la intersección de las áreas de los conos. Aunque es simple y obvio en 2D, esta representación por intersecciones es difícil de utilizar para objetos 3D, ya que su V.H consiste en áreas curvadas irregulares que son difícilmente representables usando primitivas simples de geometría y son computacionalmente caras y numéricamente inestables para calcular.

- **Representación basada en 3D.** Ya que es difícil intersectar las superficies de los conos visuales generados por objetos en 3D, se utilizan técnicas para simplificar esta tarea, una de las más usadas consiste en una aproximación basada en voxels que se verá en el apartado posterior.

2.5.1.1.4. Ambigüedad de Visual Hull. El principal problema que presenta el V.H para una escena estática, son las regiones que pueden quedar ocluidas por la intersección de los conos y en las que no hay objeto. Esto queda de manifiesto en la Figura 2.19 donde intuitivamente se representa en dos dimensiones una escena típica de dos objetos, O_1 y O_2, y de dos cámaras, C^1 y C^2, donde los objetos quedan encerrados en las intersecciones de los conos; pero además como se muestra en la Figura 2.20 surje una tercera zona de intersección donde aparecería un objeto inexistente en la reconstrucción. Esta ambigüedad puede eliminarse de manera "hardware" colocando una cámara más que elimine la zona de incertidumbre, o mediante algoritmos software como consistencia del color.

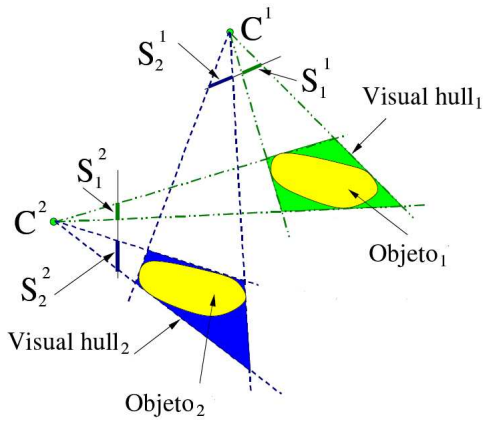

Figura 2.19: Ambigüedad en Visual Hull 1.

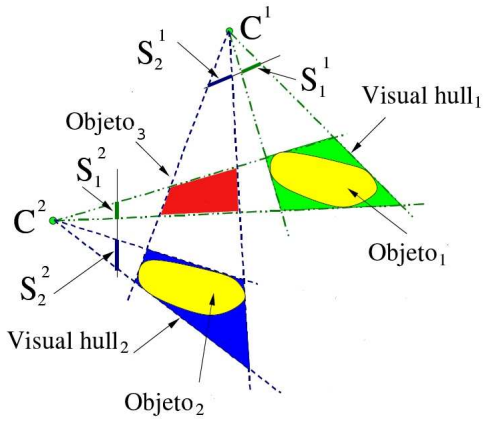

Figura 2.20: Ambigüedad en Visual Hull 2.

El la Figura 2.21 queda de manifiesto como a partir de las imágenes originales tomadas a partir de cuatro cámaras (a), se segmentan (b), y tras aplicar el Visual Hull y reconstruir la escena (c), se pueden ver en el volumen ciertas áreas de incertidumbre (d) marcadas en la

escena, que no pertenecen a los objetos originales pero pertenecen a los conos de intersección de las cuatro cámaras.

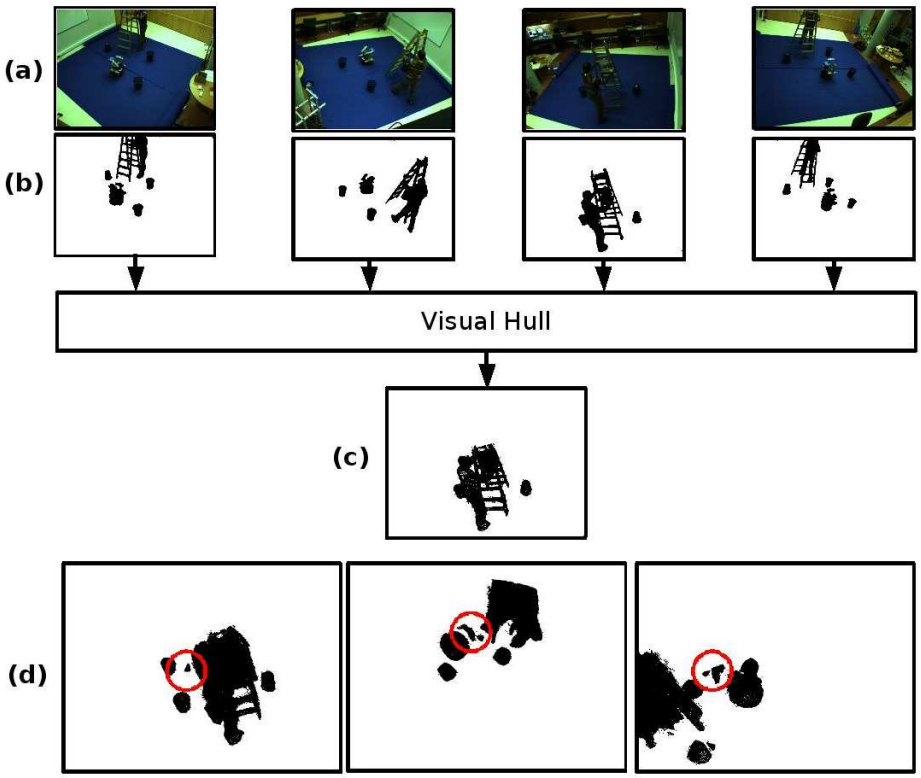

Figura 2.21: Ambigüedad en Visual Hull 3.

2.5.1.2. Visual Hull a partir de homografías.

Como se ha explicado en el capítulo de introducción, a partir de la transformación homográfica de varias cámaras se obtiene el contorno del objeto a cota z=0. De un modo gráfico en la Figura 2.22 tenemos que la relación entre la cámara 1 y el objeto cilíndrico a reconstruir aplicando la transformación homográfica, es la silueta B1. Análogamente las cámaras 2 y 3 producen con 'H' las siluetas B2 y B3. Finalmente fundiendo las siluetas B1, B2 y B3 obtenemos el contorno del cilindro a cota z=0, que forma parte del Visual Hull. A continuación se explicará como se ha conseguido la rejilla de ocupación completa.

2.5.1.2.1. Visual Hull en una rejilla de ocupación. La estrategia que se va a seguir para conseguir una rejilla de ocupación tridimensional completa partiendo de homografías, consiste en dividir el espacio inteligente en cubos de lado Δh. Por lo tanto tendremos la totalidad del espacio dividido en un gran cubo que a su vez está compuesto por pequeños voxels de lado Δh como indica la Figura 2.23. En la Figura 2.23 están resaltados en color naranja todos los voxels con una misma cota (z=0); al conjunto de voxels con una cota común se les denomina "slice". De este modo y como se ha explicado en el capítulo dedicado a la definición de la homografía, existe una relación entre los voxels de un mismo slice y la cámara. Aplicando esa relación 'H' desde cada una de las cámaras al objeto a reconstruir y fundiendo los resultados se obtiene el contorno del volumen a cota z=0. El siguiente paso es incrementar esa cota en Δh.(z=Δh.) y volver a aplicar la relación homográfica y fundir los resultados. Haciendo este proceso tantas

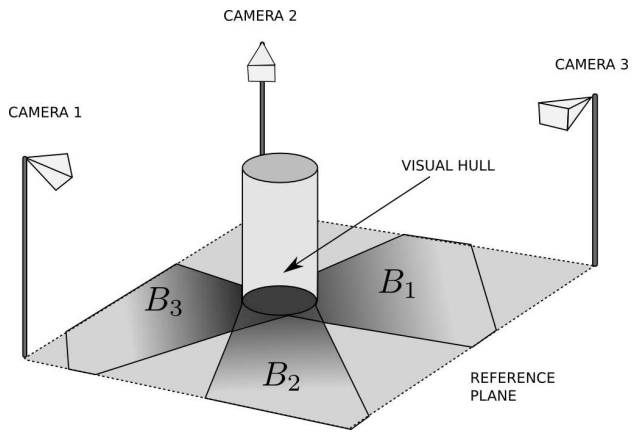

Figura 2.22: Rejilla de ocupación.

veces como slices tenga el cubo en el que se ha dividido el espacio, conseguiremos un grid de ocupación tridimensional.

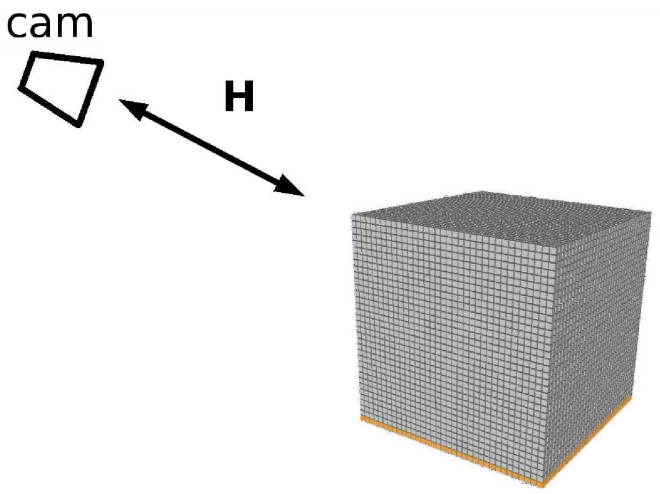

Figura 2.23: Rejilla de ocupación.

2.5.2. Algoritmos de coloreado fotorealista.

Una vez se ha obtenido la rejilla de ocupación tridimensional del objeto, llega el momento de aplicar a cada vóxel su textura o color correspondiente. Para llevar a cabo esta tarea nos encontraremos con varios problemas clave que hay que solucionar:

- Cuando se proyecta una escena de tres dimensiones en un plano de dos, es necesario determinar qué voxels son visibles y cuáles no. Es decir, tienen que estar definidos con claridad qué voxels quedan en primer plano y deben recibir color y cuáles quedan ocluidos por los primeros y no son coloreados.
- El algoritmo debe de ser lo suficientemente veloz para no tomar más tiempo del que se tarda en la tarea paralela de obtener los voxels.

Para llevar a cabo este proceso se han utilizado varios algoritmos que se describen a continuación.

2.5.2.1. Coloreado por el método del cálculo del histograma.

Este algoritmo se basa en calcular el histograma del color de cada punto del objeto, es decir, para cada uno de los voxels calcular qué color le da cada cámara y en función de ello asignarle un color. De manera simplificada, por cada punto-vóxel, se tiene un array de 0 a 255 valores (escala de grises), y cada posición de este array corresponderá al número de cámaras que "votan" por ese color determinado. De esta manera y como se observa en la Figura 2.24 podemos conocer cuantas cámaras están viendo el punto con un color determinado y asignarle ese color, o aplicar una media de los valores más votados. Extrapolando esta solución a una imagen RGB, tendremos

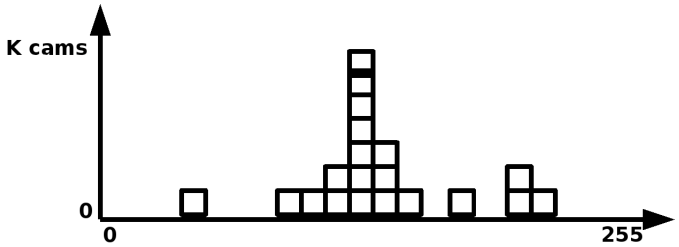

Figura 2.24: Histograma típico de un vóxel en escala de grises.

por cada vóxel tres histogramas (Figura 2.25), tomando el valor máximo en cada canal se obtiene el color adecuado.

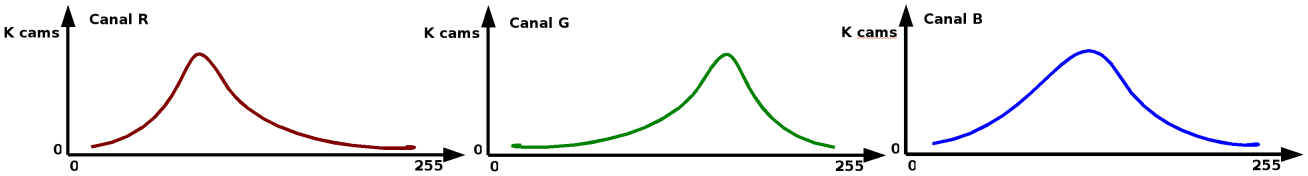

Figura 2.25: Histograma típico de un vóxel en RGB.

Pseudocódigo:

```
char histograma[256]; //variable del histograma de cada vóxel.

for(número total de voxels)
{
    bzero(histograma) //inicialización a cero del histograma.

    for(número de cámaras)
    {
        (U,V)=proyeccion 2D del vóxel 3D actual;
        U=round(U); //redondeo a un número entero (píxel X).
        V=round(V); //redondeo a un número entero (píxel Y).
        load (imagen_actual); // carga imagen de la cámara actual.
        color = Textura-Imagen-Actual(U,V) //color del pixel (U,V).
```

```
        histograma[color]++; //incremento del color medido.
    }

    color = máximo (histograma); //color del vóxel calculado.
}
```

El punto fuerte de este algoritmo es, que es el más sencillo de utilizar, y que por cada color tendremos el número exacto de cámaras que lo están viendo, consiguiendo coloreados muy cercanos a la realidad. Como punto más débil para esta solución, nos encontramos con que el tiempo de cómputo es exponencial respecto al número de puntos y de cámaras, lo que lo hace ineficiente para sistemas complejos o de tiempo real, en los que las restricciones temporales son muy altas.

2.5.2.1.1. Ambigüedad en la texturización. En la práctica, el principal inconveniente durante el coloreado, es que las cámaras son capaces de ver los puntos de las caras en un "segundo plano", que en teoría deberían estar ocluidas por las caras en primer plano. Este problema es debido a que los voxels no forman una superficie continua (malla) sino que son un conjunto discreto de puntos en el espacio, de esta manera las cámaras tienen en cuenta y aportan también color a todos los puntos en la escena (indistintamente si están en primer o segundo plano) creando coloreados erróneos en las caras en segundo plano. Para ilustrar de una manera más gráfica esta ambigüedad, se ilustra en la Figura 2.26 una escena típica en dos dimensiones donde una cámara reconstruye un volumen en forma de óvalo formado por voxels (cuadrados). Los voxels de la cara a texturizar, es decir en primer plano, se representan en color azul, mientras que los voxels en segundo plano que no deberían de ser tomados en cuenta se representan en rojo. Se observa de manera clara como en el plano imagen aparecen los voxels en segundo plano, apareciendo la citada ambigüedad.

Hay que tener en cuenta, que la importancia de este error es directamente proporcional al tamaño del vóxel, así como se muestra en la Figura 2.27. Con un tamaño "pequeño" del vóxel (a), la cantidad de puntos erróneos en el plano imagen es relativamente alto, frente al resultado obtenido generando un vóxel más grande(b).

Una de las posibles soluciones para este problema es aplicar un mallado al conjunto de puntos de la escena, con lo que se consigue una superficie robusta, o como se ha hecho en este proyecto, utilizar métodos de coloreado más complejos como se detalla en los sucesivos capítulos.

2.5.2.1.2. Resultados obtenidos. Realizando una reconstrucción del volumen del coche de juguete (Figura 2.28) utilizando la mesa giratoria y aplicando un coloreado en escala de grises (un solo canal) basado en calcular el histograma de cada vóxel se obtiene el volumen de la Figura 2.29, en él se aprecia que si se rota la escena, (Figura 2.30), el coloreado de los voxels que quedan en segundo plano no es correcto debido al problema anteriormente descrito.

2.5.2.2. Texturización basada en perspectiva inversa.

Este segundo método, que asigna a cada vóxel su color correspondiente tiene dos partes diferenciadas.

- La primera consiste en diferenciar del conjunto total de voxels, cuáles de ellos forman parte de la corteza del objeto, así tendremos una distinción entre puntos internos de la figura que no reciben color y puntos de la corteza que deben ser coloreados.

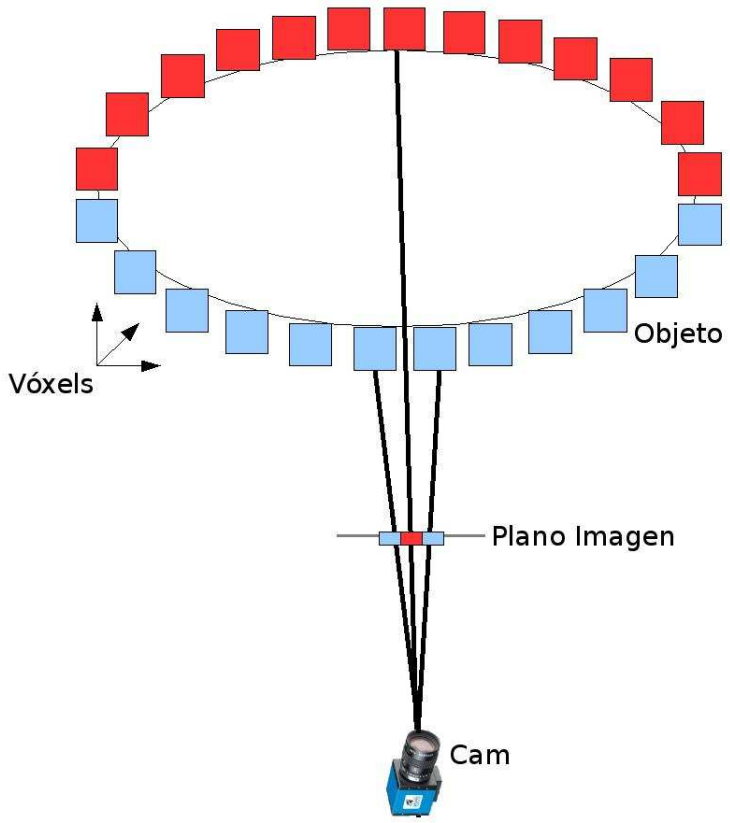

Figura 2.26: Ambiguedad en la texturización.

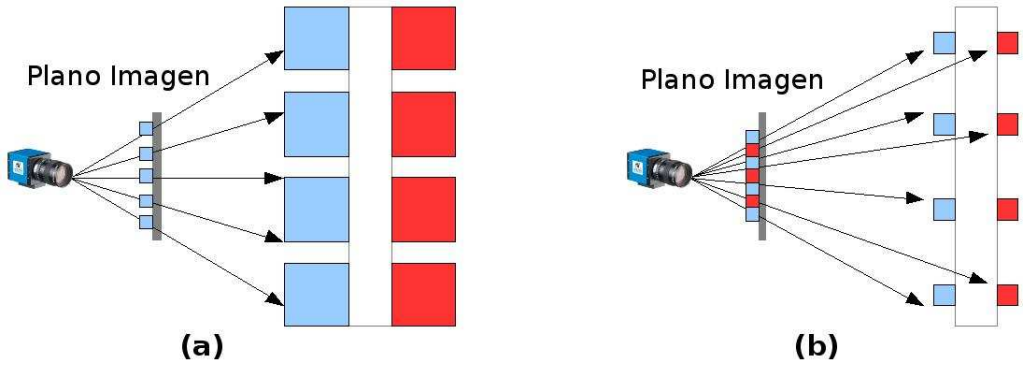

Figura 2.27: Relación error/tamaño del vóxel.

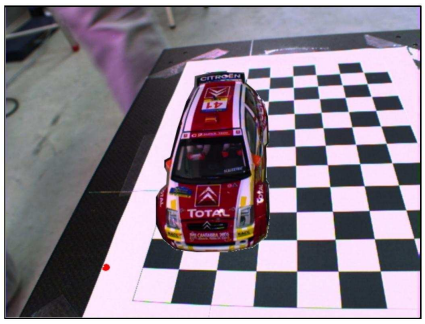

Figura 2.28: Escena original a reconstruir.

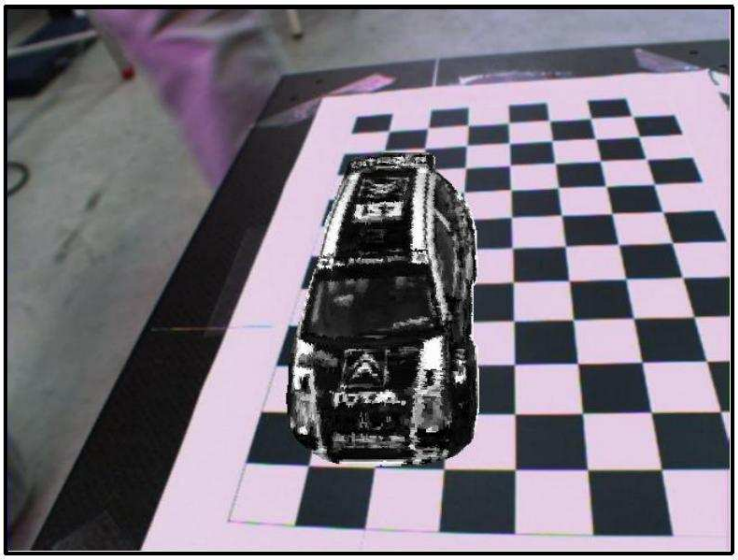

Figura 2.29: Instantánea del volumen reconstruido y texturizado.

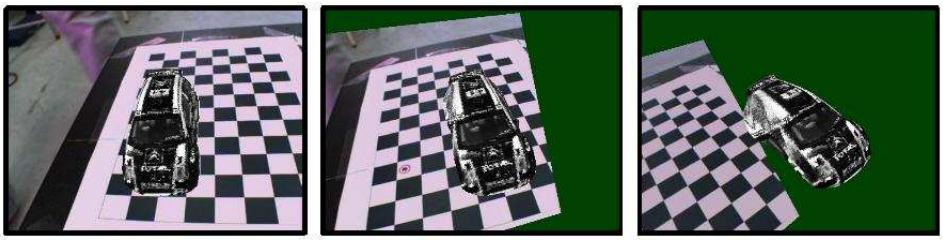

Figura 2.30: Rotación de la figura.

- La segunda parte del método consiste en asignar un color a cada uno de los voxels de la corteza, una vez realizada la diferenciación previa.

Para saber qué puntos forman parte de la corteza, trazaremos, como se indica en la Figura 2.31, un rayo desde la cámara hasta el punto a colorear.

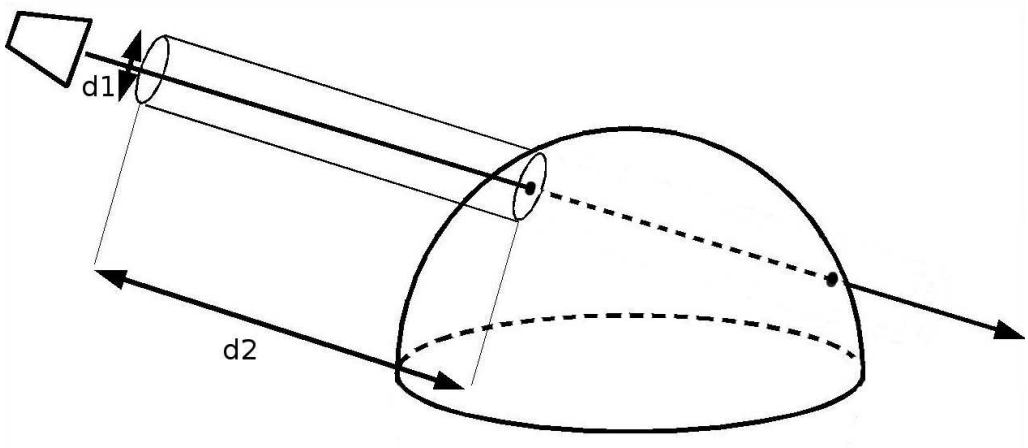

Figura 2.31: Algoritmo de coloreado 2.

De este modo quedan definidos:
$d_1 = \|\lambda \cdot D + T - X_p\|$ Como la anchura del haz trazado desde el centro óptico de la cámara hasta cada punto del objeto.
$d_2 = \|\lambda \cdot D\|$ Como la distancia entre el plano imagen y la superficie del objeto. Donde:

$$D = \frac{R'_c \cdot K^{-1} \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}}{\left\| R'_c \cdot K^{-1} \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \right\|} \tag{2.28}$$

$$\lambda = D^t \cdot X_p - D^t \cdot t \tag{2.29}$$

$$T = -R'_c \cdot T_c \tag{2.30}$$

Por lo tanto, podemos resumir la manera de calcular los puntos exteriores (corteza) del volumen de la siguiente manera:

1. **Cálculo de los voxels 3D que conformarán la escena:** Utilizando el método de obtención de rejilla tridimensional con homografías descrito en capítulos anteriores, se obtienen un conjunto de puntos 3D que determinan la escena.

2. **Cálculo de d1 para todos los voxels:** Se calcula d1 para la totalidad de los voxels, para lo cual hay que calcular con anterioridad D, λ y T. Estableciéndose previamente un umbral d1, que permite marcar el grosor del haz de forma dinámica. Tras este paso, del conjunto global de todos los voxels, sólo quedarán aquellos que estén en la intersección entre el rayo óptico y el volumen. Estos quedan representados en color azul en la Figura 2.32.

3. **Cálculo de d2 para voxels que pasan el umbral de d1:** Del conjunto de voxels que pasan la criba de d1 hay que quedarse con aquel que tenga una distancia d2 menor (o conjunto de voxels con un umbral d2 determinado). Estos voxels supervivientes de las dos umbralizaciones (d1 y d2) serán aquellos que forman parte de la corteza, que queda representada en color rojo en la Figura 2.32.

4. **Repetición:** Hay que volver a calcular este proceso tantas veces como voxels tenga la figura.

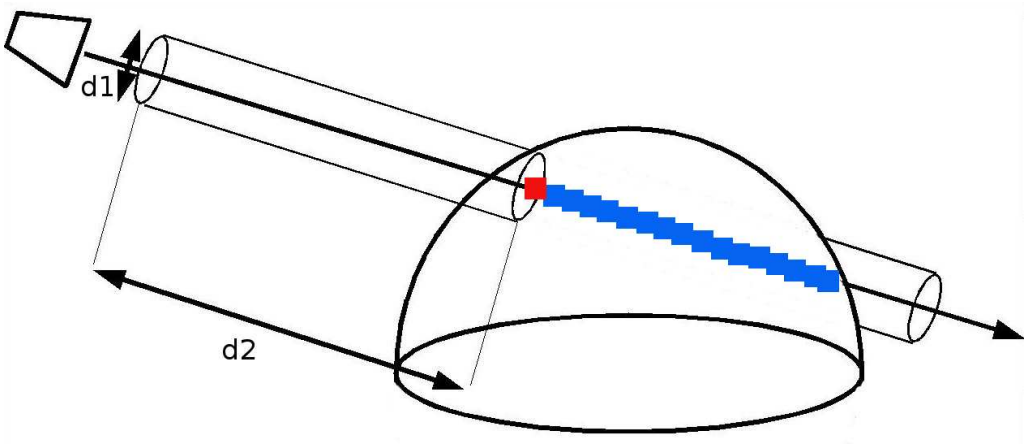

Figura 2.32: Algoritmo de coloreado 2, explicación.

De manera sintética, el pseudocódigo de este proceso se puede resumir de la siguiente manera:

```
/// Previamente ha  calculado la posición espacial (x,y,z)
/// de todos los voxels de la escena

/// ************************************************
/// *** Parte que calcula los voxels de la corteza ***
/// ************************************************

for(número_cámaras)
 {
 //carga de imagen segmentada vista por la cam
 load (imagen_segmentada)
 for(anchura_imagen_segmentada)
  {
        for(altura_imagen_segmentada)
        {
            if(pixel_segmentado)
            {
                for(número_voxels)
                {
                    Cálculo de T
                    Cálculo de D
                    Cálculo de L
                    Cálculo de d1
```

```
                    if(d1<umbral_d1)
                    {
                        vóxel semi-válido
                    }
                }

                for(número_voxels_semi-válidos)
                {
                    Cálculo de d2
                    if(d2<umbral_d2)
                    {
                        vóxel válido
                    }

                }
            }
        }
    }
}
```

Segunda parte del algoritmo:

```
/// ******************************************
/// *** Parte que asigna textura a los voxels ***
/// ******************************************
for(número_cámaras)
{
   //carga de la imagen original vista por la cam.
   load (imagen_textura)
   for(número_voxels_válidos)
   {
      //(u,v)=proyección 2D del vóxel 3D
      textura=color imagen_textura(u,v)
   }
}
```

En este punto es importante indicar que en el proceso de coloreado, cada vóxel de la corteza debe tomar el color de la cámara que lo está observando en primer plano; por lo tanto, en el momento de obtener los voxels con d2 mínimo, es decir aquellos que forman parte de la corteza, el programa, en un proceso intermedio les asigna además un color (no realista), con el fin de dejar emparejado el vóxel con la cámara que le dará su color verdadero. Como indica la Figura 2.33 y 2.34, el programa da a cada vóxel un color dependiendo de que cámara lo observa en primer plano. Así tenemos que la primera cámara debe colorear los puntos de color rojo, la segunda cámara a los de color verde, y así sucesivamente hasta completar la escena.

Esta aproximación, consume mucho tiempo de cómputo en el momento de calcular que puntos del objeto forman parte de la corteza, pues se trabaja con matrices muy grandes y sobre un número muy grande de puntos. El tiempo de cómputo es cuadrático con el número de puntos y de cámaras en la escena. Aparte de los problemas temporales, el algoritmo ha generado

Figura 2.33: Imagen original e Imagen coloreada según la cámara texturizadora.

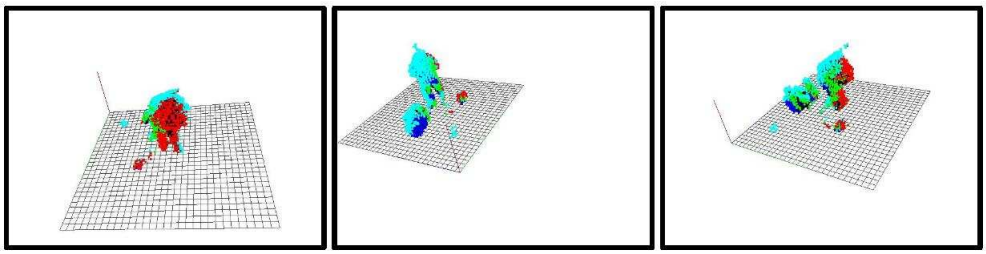

Figura 2.34: Distintas vistas de la escena reconstruida.

resultados muy fotorealistas y robustos.

2.5.2.2.1. Resultados obtenidos. Se realiza la reconstrucción de la escena de un robot dentro del espacio inteligente. En la Figura 2.35 se puede observar la captura original realizada por una de las cámaras.

Figura 2.35: Escena original a reconstruir.

Para ello se usan las 4 cámaras del espacio y se aplica un coloreado RGB (3 canales) basado

en perspectiva inversa anteriormente explicada. El resultado obtenido se muestra en la Figura 2.36 y en la Figura 2.37:

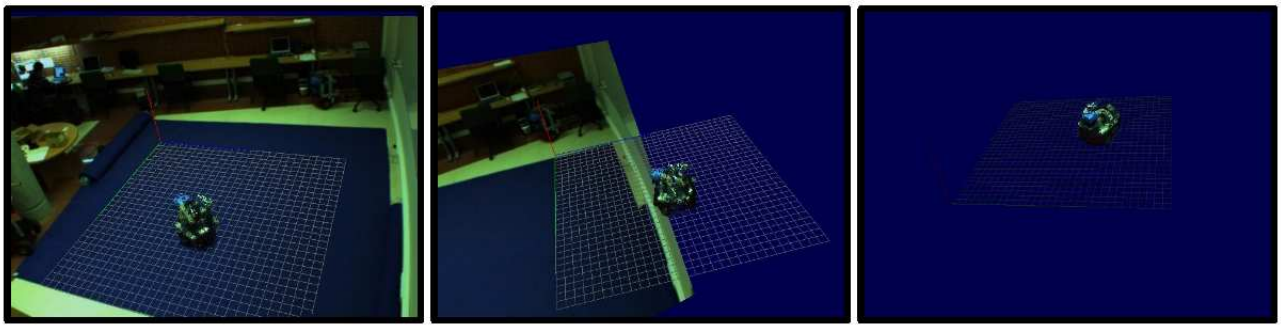

Figura 2.36: Escena reconstruida desde varias perspectivas.

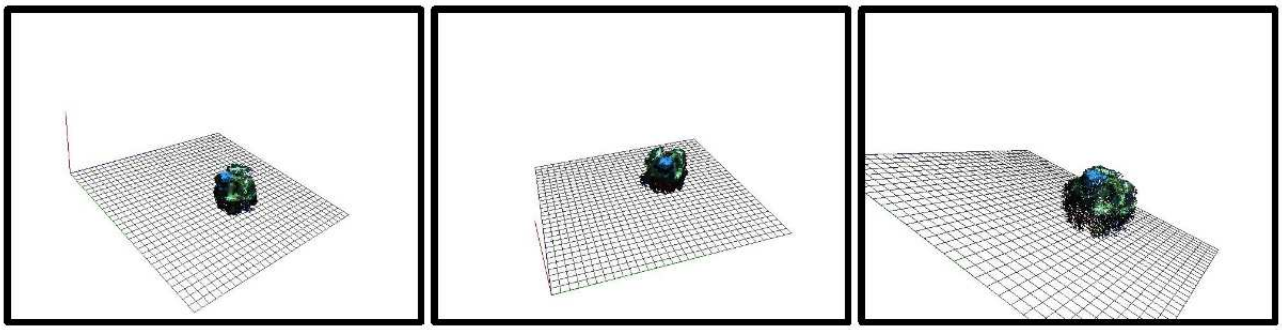

Figura 2.37: Escena reconstruida sin imagen de fondo.

2.5.2.3. Coloreado por el Algoritmo del Pintor.

El nombre "Algoritmo del Pintor" se refiere a un pintor que primero dibuja los elementos lejanos de una escena y después los cubre con los más cercanos. Este algoritmo ordena todos los polígonos de una escena en función de su profundidad y después los pinta en ese orden, pintando encima de las partes que no son visibles y solucionando así el problema de la visibilidad (figura 2.38).

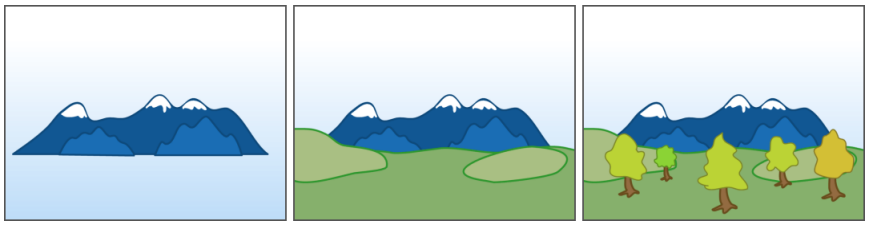

Figura 2.38: Orden de pintado de un escenario.

Se pintan primero las montañas lejanas, seguidas por el prado; finalmente se dibujan los objetos más cercanos, los árboles.

El algoritmo puede fallar en determinados casos. En la Figura 2.39, los polígonos A, B y C están superpuestos. No es posible determinar qué polígono está por encima de los otros o cuando dos se intersectan en tres dimensiones. En este caso, los polígonos en cuestión deben ser cortados de alguna manera para permitir su ordenación. El algoritmo de Newell propuesto en 1972 da una solución para cortar dichos polígonos. También se han propuesto numerosos métodos en el campo de la geometría computacional.

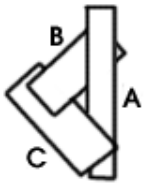

Figura 2.39: Problema en Algoritmo del Pintor.

En este proyecto se han aplicado técnicas de Z-Buffer, que pueden ser vistas como un desarrollo del Algoritmo del Pintor que resuelve los conflictos de profundidad píxel por píxel, reduciendo la necesidad de una ordenación por profundidad. Incluso en estos sistemas, a veces se emplea una variante del "Algoritmo del Pintor". Las implementaciones del Z-Buffer generalmente se basan en un buffer limitado de profundidad implementado por hardware, pueden producirse problemas de visibilidad debido a los errores de redondeo, provocando la superposición en la unión de dos polígonos. Para evitarlo, algunos motores gráficos realizan el "sobrerenderizado", dibujando los bordes de ambos polígonos en el orden impuesto por el Algoritmo del Pintor. Esto significa que algunos pixels se dibujan dos veces (como en el Algoritmo del Pintor normal), pero sólo ocurre en pequeñas zonas de la imagen y apenas afecta al rendimiento.

Para aplicar este algoritmo se define la distancia desde cada vóxel a la cámara Z', calculada a partir de la ecuación:

$$\begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} = R \cdot (\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + T) \tag{2.31}$$

Por lo tanto cada punto 3D del objeto a representar es proyectado a 2D y ordenado en función de la distancia a la cámara Z', de manera que se conocerán los puntos más cercanos a la cámara (corteza), para asignarla un color de manera precisa.

De manera sintética, el pseudocódigo de este proceso se puede resumir de la siguiente manera:

```
/// Previamente se calcula la posición espacial (x,y,z)
/// de todos los voxels de la escena

for(número_cámaras)
{
        // se inicializa a un valor conocido (-2) los arrays
        // de voxels y de profundidad de los voxels
        array_cam[][][]=-2;
        array_profundidad[][][]=-2;
}

/// ************************************************
```

```
/// *** Cálculo los voxels más cercanos a la cámara ***
/// ***************************************************

for(número_cámaras)
{
   for (número_voxels)
   {
      //cálculo de la profundidad 'z'
      [x;y;z]=[Rc_ext1*[X;Y;Z]+Tc];
      profundidad_z_voxel=z;

      if(array_profundidad[número_cámara][x][y]==-2)
      {
         array_profundidad[][][]=profundidad;
         array_cam[][][]=índice_voxel;
      }

      else if(array_profundidad[][][]>profundidad)
      {
         array_profundidad[][][]=profundidad;
         array_cam[][][]=índice_voxel;
      }

   }
}
```

```
/// ***********************************************************
/// *** Parte que asigna a los "voxels cercanos" su textura ***
/// ***********************************************************

for (número_cámaras)
{   // se carga la imagen original vista por la cámara
    // que aportará la textura
    load(imagen_textura);

    // se carga la imagen segmentada
    load(imagen_segmentada);

    for (anchura_imagen_textura)
    {
       for (altura_imagen_textura)
       {
           if (array_cam[número_cámara][anchura][altura]!=-2)
           {
              //se carga el vóxel apuntado (índice) por array_cam
              (x,y,z)=proyección 2D del vóxel 3D

              if(píxel[anchura][altura]_imagen_segmentada>umbral)
              {
                 vóxel_válido para la representación.
```

```
                    asignación de color.
                    color[número_cámara][índice_voxel]=color_calculado;

                }
            }
        }
    }
}
```

El punto fuerte de éste algoritmo es que el tiempo de cómputo es lineal con el número de puntos y de cámaras, por lo que para sistemas en tiempo real es la solución más acertada, aunque el coloreado no sea tan fiel como el conseguido con otras técnicas de coloreado.

Otro punto a favor es que se le pueden aplicar técnicas de media o mediana para el cálculo del color del píxel. Estas consisten, en asignarle a cada vóxel del objeto un array donde se guardan los distintos valores de color que le asigna cada cámara, siempre en el caso de que la cámara "vea" al citado pixel. Una vez guardados todos los valores se les aplican la media o la mediana para escoger el color más apropiado.

Los resultados de ambos algoritmos son bastante similares, como se puede observar, se reconstruye el coche de juguete y se aplican en el caso de la Figura 2.40 la mediana y la Figura 2.41 la media. Se aprecia que en la Figura 2.40 el resultado de la mediana tiene los rasgos algo más definidos y dado que consumen tiempos de cómputo similares, ha sido el método de coloreado finalmente escogido para el resto de pruebas.

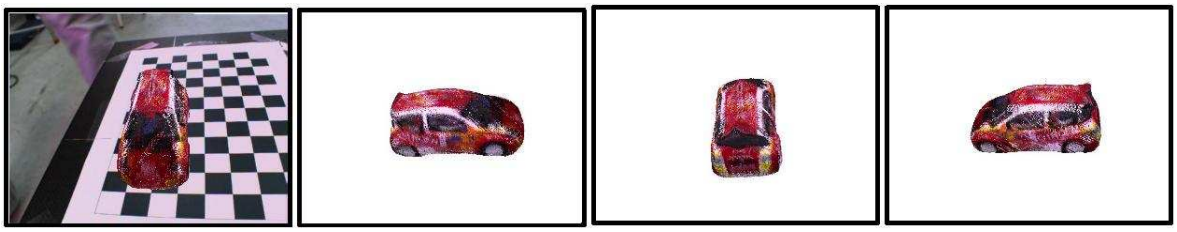

Figura 2.40: Textura con mediana del color.

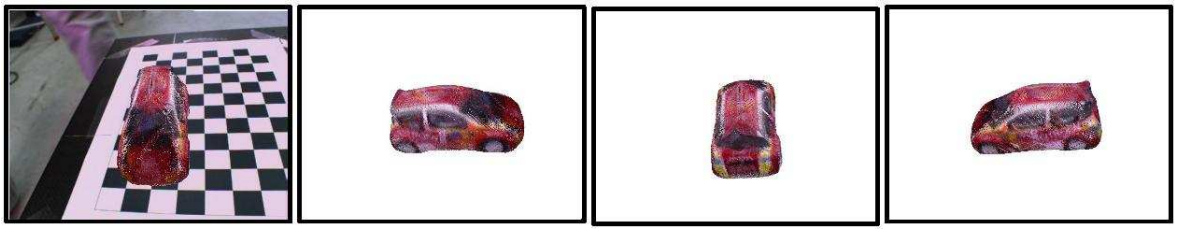

Figura 2.41: Textura con media del color.

2.5.2.3.1. Resultados obtenidos. Para hacer una muestra de los resultados obtenidos, se realiza una reconstrucción a partir de una serie de volúmenes, (tres papeleras, una escalera y un robot) mostrados en la Figura 2.42. Aplicando un coloreado en tres canales RGB basado en el Algoritmo del Pintor, se obtiene la reconstrucción 3D de la Figura 2.43, en la que se pueden apreciar los diferentes detalles de la escena, Figura 2.44.

Figura 2.42: Captura de la escena original.

Figura 2.43: Captura de los volúmenes reconstruidos y texturizados.

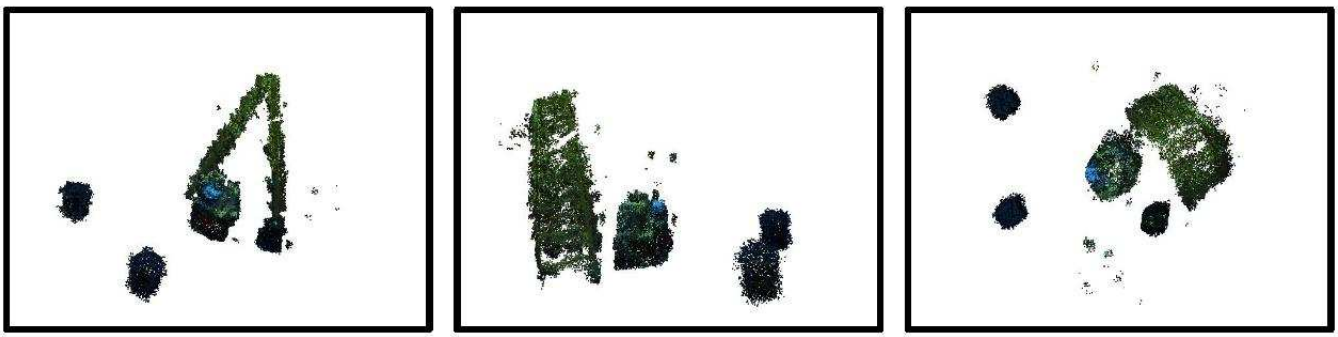

Figura 2.44: Distintas vistas, laterales y aéreas de la escena.

2.6. Reconstrucción volumétrica aplicada a espacios inteligentes.

En este capítulo y para mostrar los resultados obtenidos, se ha llevado a cabo en el espacio inteligente una reconstrucción típica de un conjunto de volúmenes.

2.6.1. Detección de múltiples objetos a partir del grid de ocupación.

Se ha realizado la reconstrucción de una escena compleja, formada con varios objetos independientes, como se muestra en las Figuras 2.45, 2.46, 2.47 y 2.48. En ellas se muestran una serie de 12 frames en los que se observa enmarcada en negro la escena original tomada por una de las cámaras del espacio inteligente. Enmarcada en azul la escena reconstruida (aplicando el Algoritmo del Pintor) generada con OpenSG. Enmarcado en verde el mismo frame con una vista aérea, situando la cámara virtual por encima de la escena y utilizando la información de 4 cámaras. Por último, enmarcada en rojo, la misma vista aérea comentada anteriormente aunque utilizando sólo la información de 3 cámaras.

Con los resultados obtenidos, se pueden observar los siguientes fenómenos:

1. **Ambigüedad de V.H**: Producida por las regiones ocluidas resultantes de la intersección de los conos visuales de las cámaras en las que no hay objeto. Queda de manifiesto en las vistas aéreas de varios frames, donde aparecen pequeños volúmenes en forma de manchas. Es un efecto mucho más acusado en el caso de 3 cámaras que en de 4, dado que con menos conos visuales, las regiones de intersección son más grandes y las probabilidades de que aparezca una ambigüedad también. En la Figura 2.49, se muestra un mismo frame recostruído por 1, 2, 3 y 4 cámaras, se observa como la geometría mejora con cada cámara que se añade.

2. **Zonas muertas**: Son las zonas en las que se tiene menos información, son consecuencia de la colocación de las cámaras. De esta manera se tienen puntos óptimos para observar la escena, es decir aquellos puntos de vista que coinciden con los puntos en los que se sitúan las cámaras, y las llamadas zonas muertas que se encuentran a la distancia media entre cámara y cámara. En estas zonas se producen unas reconstrucciones muy burdas de la geometría de la escena y pueden ser corregidas mediante software (constancia del color) o hardware (posicionando una nueva cámara en el punto crítico).

3. **Geometría incorrecta**: Como ya se explicó en el apartado dedicado a Visual Hull, la

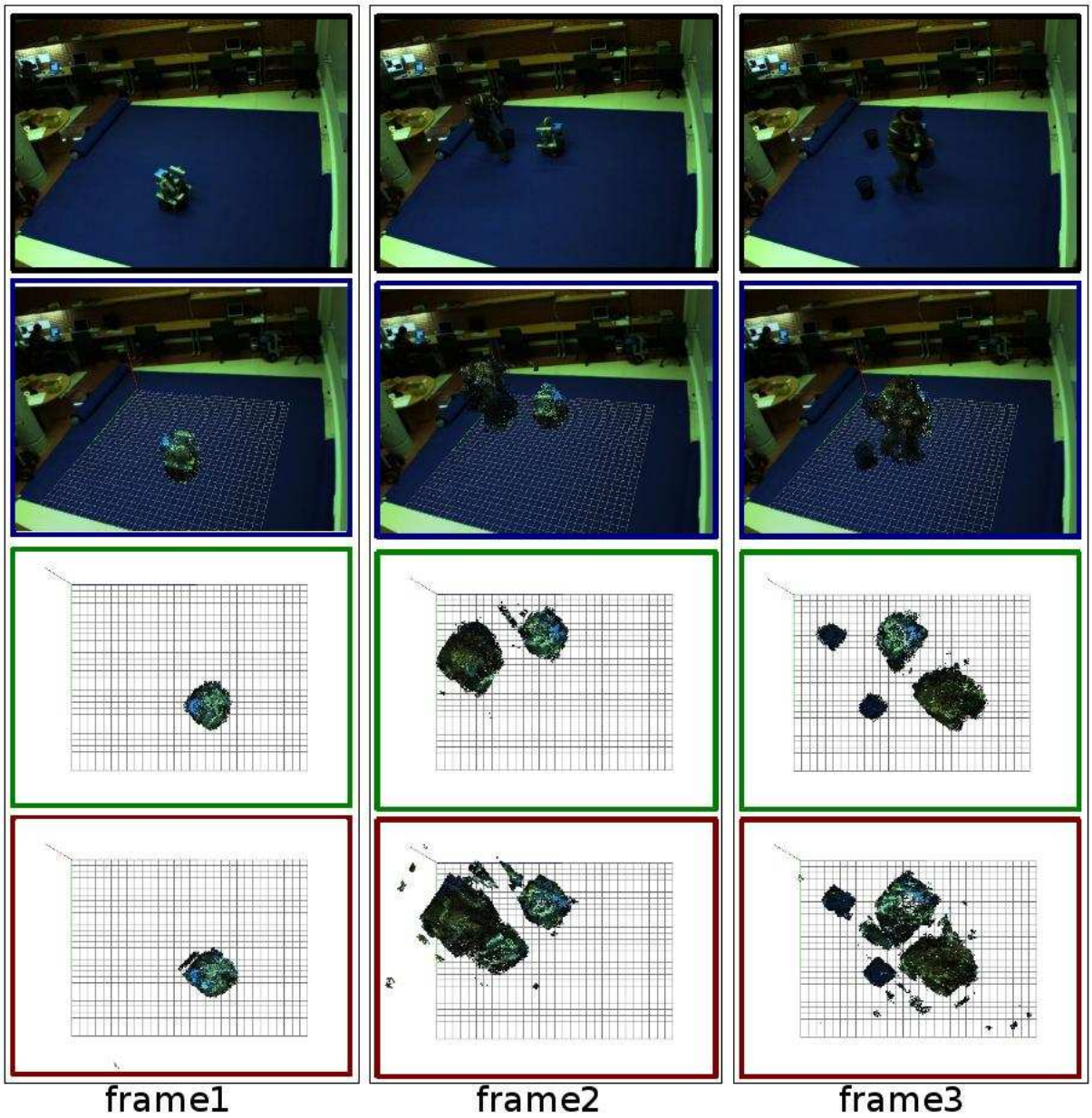

Figura 2.45: Tira de frames 1.

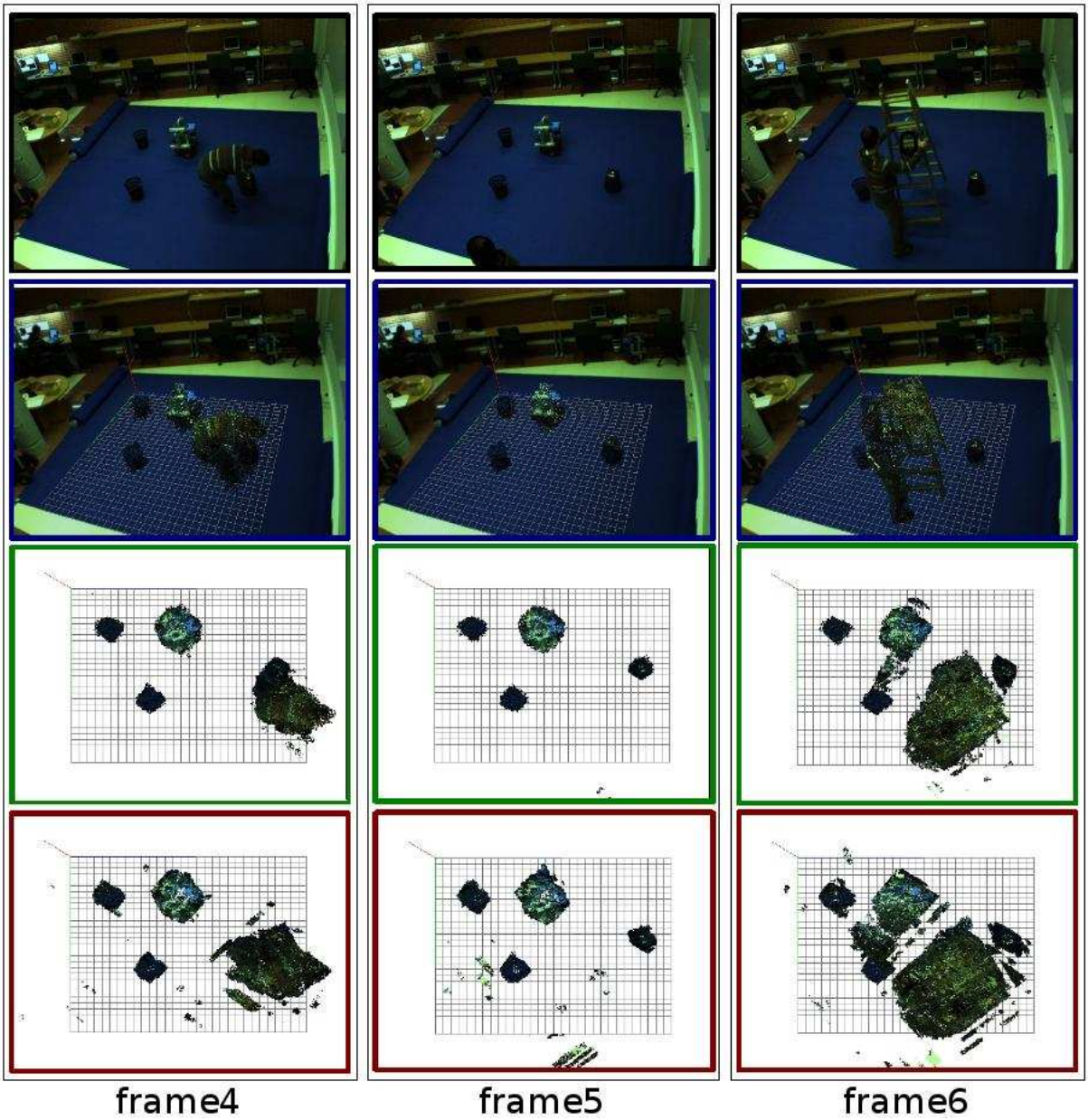

Figura 2.46: Tira de frames 2.

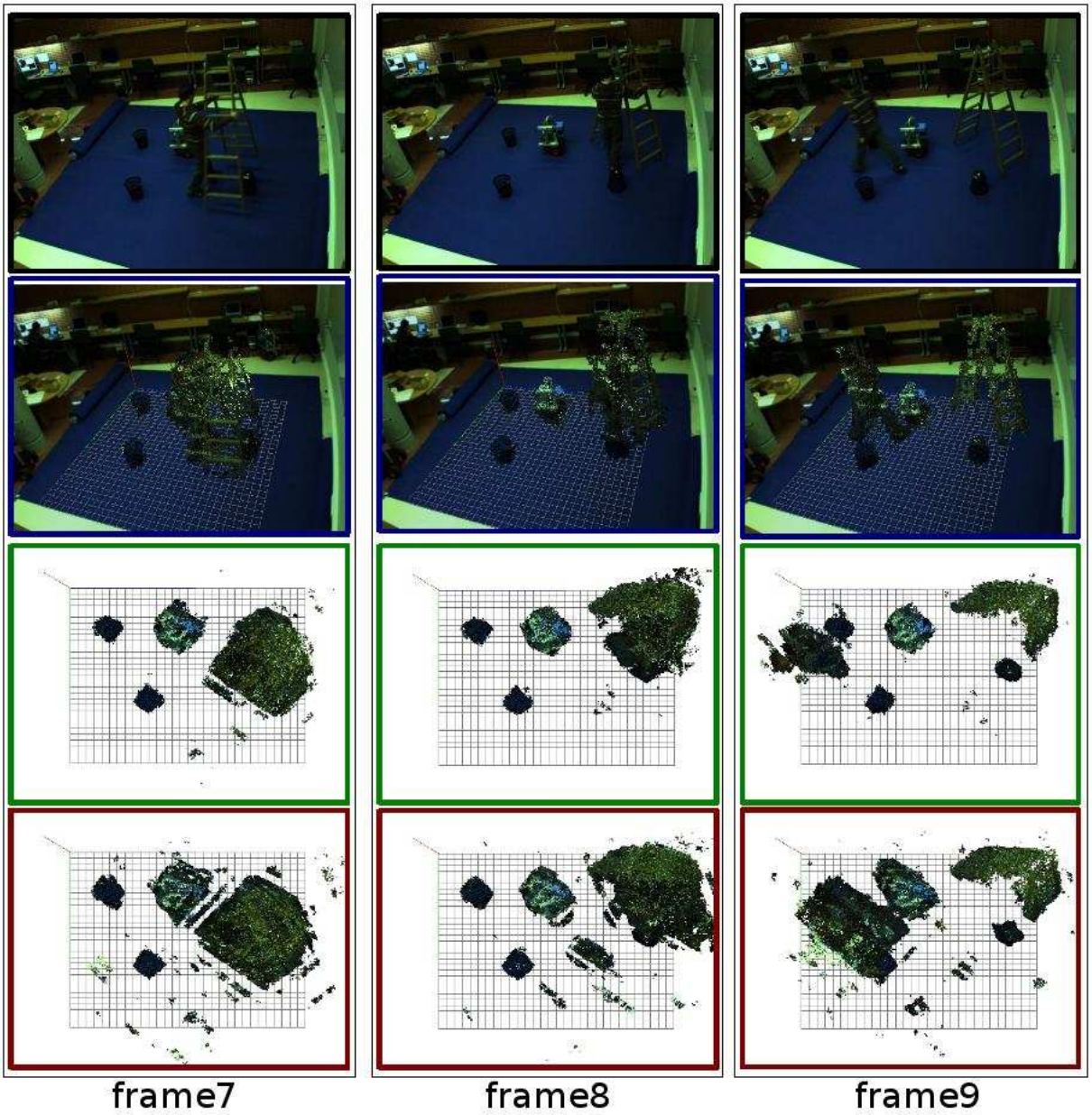

Figura 2.47: Tira de frames 3.

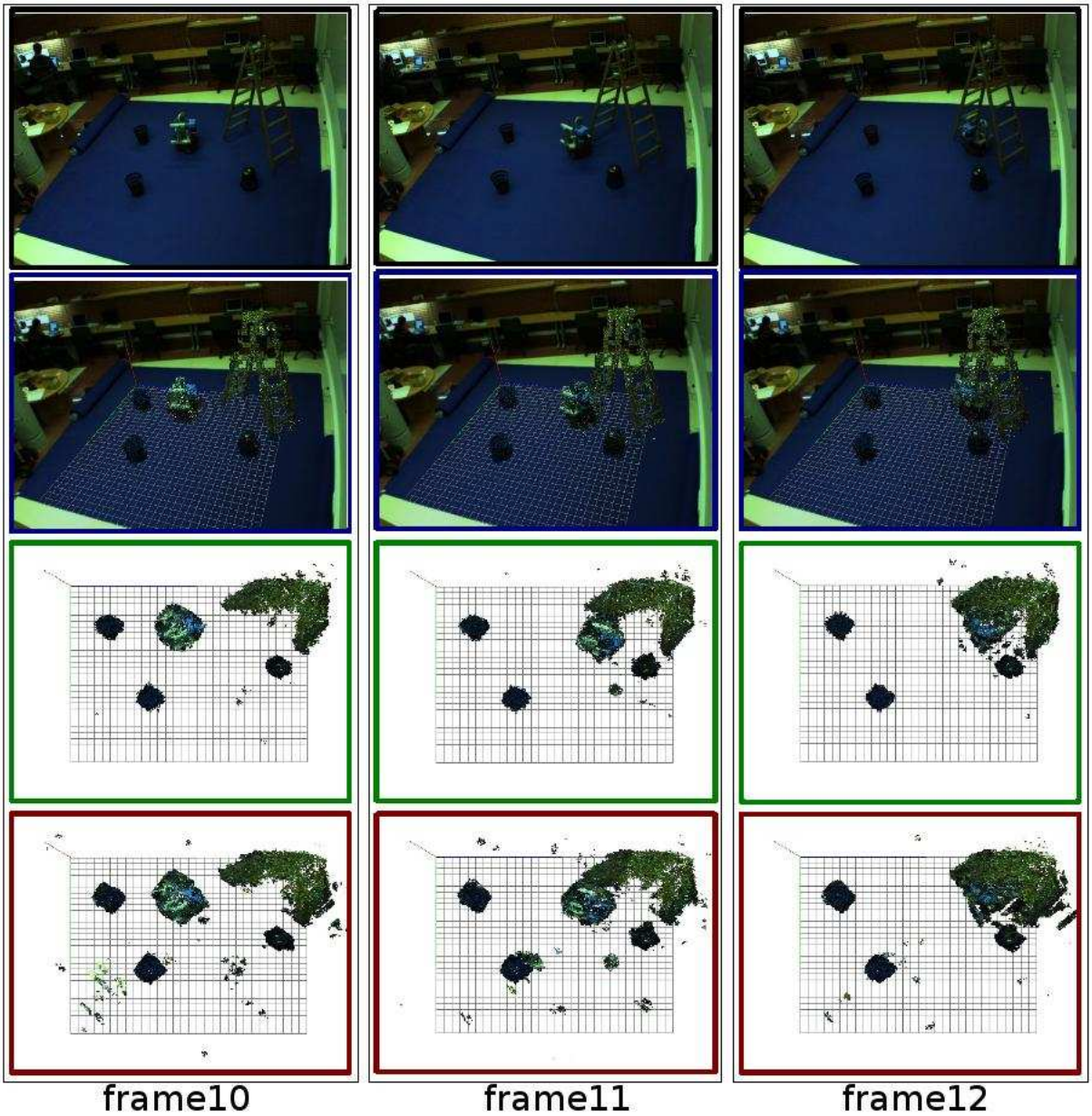

Figura 2.48: Tira de frames 4.

Figura 2.49: La misma escena reconstruida por distinto número de cámaras.

pérdida de cierta geometría de los objetos de la escena es un aspecto inherente al V.H. De esta manera y como se observa en la Figura 2.19, un objeto con un contorno circular se verá de manera trapezoidal si no se tienen las suficientes cámaras. En la vista aérea de la Figura 2.50 se ve como el contorno redondeado de las 3 papeleras y el robot (a la derecha) se reconstruyen como trapecios (marcados en rojo a la izquierda). Una posible solución para estos problemas sin añadir nuevas cámaras es de nuevo aplicar algoritmos de refinado de la geometría por la constancia del color.

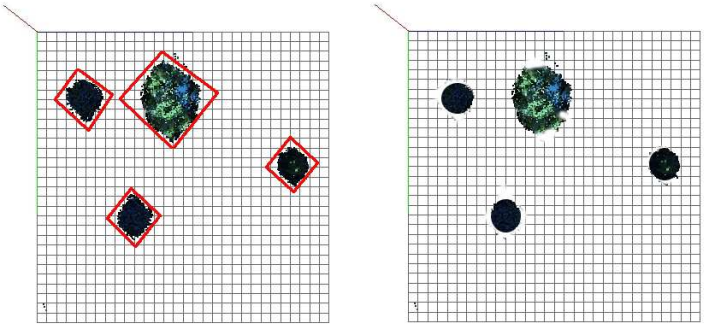

Figura 2.50: Geometría reconstruida vs geometría ideal.

El siguiente paso para detectar los diferentes objetos a partir del grid 3D generado, es aplicar técnicas de clustering o de filtro de partículas sobre las mismas.

2.6.2. Implementación hardware del sistema.

Como se ha descrito en el capítulo de "Geometría de Formación de la Imagen", tenemos dos posibles escenarios, con distintas implementaciones hardware, la mesa giratoria y el espacio inteligente:

- **La mesa giratoria**: Es un sistema sencillo que se diseñó con el fin de poder ensayar teorías y aplicaciones de manera controlada, como se observa en la Figura 2.51, la implementación hardware está formada por los elementos mecánicos que forman la mesa, la cámara y un computador encargado de capturar y computar los datos de la reconstrucción. Se diferencia aquí un solo tipo de enlace tipo Firewire IEEE 1394 entre el computador y la cámara.

El espacio inteligente tiene un hardware algo más complejo y es explicado en el siguiente apartado.

2.6.2.1. Sistema distribuido de adquisición y control de múltiples cámaras.

- **El Espacio Inteligente**: Es un sistema con una arquitectura en la que las conexiones están diseñadas con el fin de minimizar los cuellos de botella y mantener un sistema simple y flexible. Como se muestra en la Figura 2.52 hay una primera conexión entre las cámaras y los computadores denominados "servidores" de tipo Firewire IEEE 1394, esta conexión proporciona a los servidores la captura de los datos de la escena en los tiempos requeridos. Del mismo modo hay otra conexión de tipo Ethernet LAN entre los computadores servidores (proporcionan, sirven las imágenes) y el computador cliente que procesa las imágenes proporcionadas por los computadores mencionados.

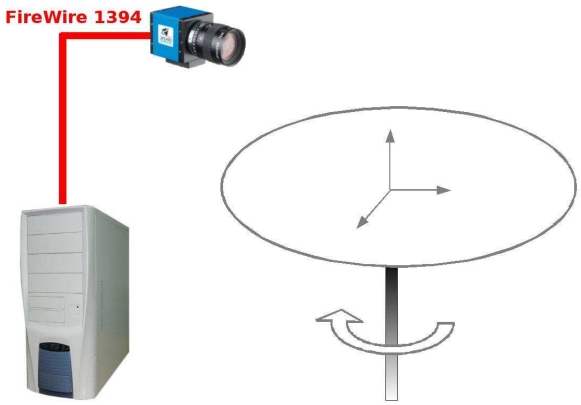

Figura 2.51: Implementación hardware en el escenario de la mesa.

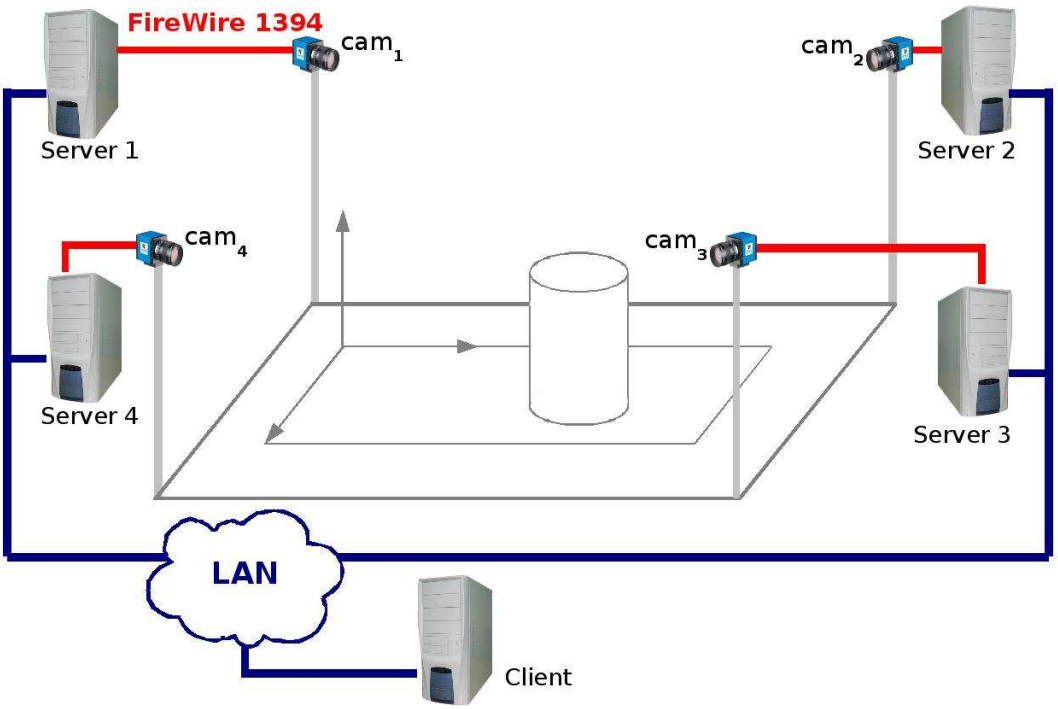

Figura 2.52: Implementación hardware en el escenario del espacio inteligente.

De forma adicional resaltar que en la Figura 2.52 se ha representado un computador por cada cámara; esta situación es flexible, ya que dependiendo de la naturaleza y requerimientos del sistema se puede colocar un computador con dos cámaras, con cuatro cámaras, etc... además y como se detallará en el capítulo de "Diseño distribuido para la Obtención de la rejilla de Ocupación", los servidores (nodos) pueden realizar otras tareas auxiliares además de servir las imágenes.

2.6.3. Implementación software del sistema.

En este apartado se explican las diferentes librerías y herramientas software utilizadas. También y de una manera concreta, se explica el flujograma de la aplicación final que lleva a cabo la reconstrucción volumétrica y las funciones más significativas desarrolladas. La medida de los tiempos medios de ejecución de cada parte de la aplicación es trascendental, para más adelante justificar la utilización de una arquitectura cliente-servidor para implementar el sistema de una manera distribuida.

2.6.3.1. Descripción de la herramienta OpenCV.

OpenCV (Open source Computer Vision library) es una librería de código libre desarrollado por Intel. Proporciona funciones de alto nivel para el tratamiento de imágenes en tiempo real. Son muy útiles para la captura, procesado y preprocesado de imágenes dado el alto número de algoritmos y funciones que incorpora.

En este proyecto las funciones que han sido programadas con esta librería son:

- **Captura de imágenes:** permite la operación básica de interactuar con el entorno a partir de una cámara conectada al computador.
- **Operaciones básicas para el procesado y análisis de imágenes:** para tareas simples como la extracción del fondo en las imágenes tomadas con la mesa giratoria o el cálculo de los contornos en los slices del volumen. También para funciones más complejas como por ejemplo la segmentación en tiempo real de la imagen original.
- **Operaciones matemáticas con matrices:** openCV proporciona gran cantidad de funciones óptimas para el tratamiento de matrices y operaciones de matemáticas (suma, producto, inversión, etc ...)
- **Calibración de cámaras:** también proporcionan funciones para el cálculo de parámetros intrínsecos y extrínsecos. Aunque por sencillez en este proyecto se han calibrado las cámaras con la herramienta Matlab.

2.6.3.2. Descripción de la herramienta OpenSG.

Open Scene Graph (OpenSG) es una librería de código abierto multiplataforma, enfocada a la creación de gráficos 3D, usada por especialistas en diversos campos, como son el CAD, la simulación de vuelo, juegos, realidad virtual, representación y modelado científico. Está basada por completo en programación orientada a objetos (C++) y OpenGL, situándose por encima de ésta y liberando al programador de codificar y optimizar llamadas a la generación de gráficos de bajo nivel y dotándole de utilidades adicionales para un desarrollo rápido. El propósito de utilizar esta librería específica para el tratamiento de gráficos es:

- Ocultar la complejidad de la interfaz con las diferentes tarjetas gráficas, presentando al programador una API única y uniforme.
- Ocultar las diferentes capacidades de las diversas plataformas hardware, requiriendo que todas las implementaciones soporten el conjunto completo de características de OpenGL (utilizando emulación software si fuese necesario).

La operación básica que se va a realizar es crear primitivas tales como puntos, líneas y polígonos, y convertirlas en pixels. Este proceso es realizado por una pipeline gráfica conocida como la Máquina de estados de OpenGL/OpenSG. La mayor parte de los comandos crean primitivas en el pipeline de la tarjeta gráfica o configuran cómo el pipeline procesa dichas primitivas.

2.6.3.2.1. Flujograma del programa. A modo de diagrama, la aplicación se puede explicar como en la Figura 2.53, se puede observar que está dividida en tres bloques principales. A groso modo se puede decir que el primero o "función main" se encarga de iniciar el control de la aplicación, el segundo bloque o "hilo" se encarga de todas las operaciones involucradas en obtener el grid 3D de la escena, y por último, el tercer bloque o "callback" realiza el coloreado y representación de esta.

El primer bloque, o bloque principal "main" lleva a cabo todo el control del programa y se encarga de efectuar las funciones de inicialización. De una manera más detallada los procesos que lo componen son:

- Se establecen los parámetros genéricos de la herramienta OpenSceneGraph, ayudas, menús y modelo principal:

```
/// Tamaño y posición de la ventana
putenv("OSG_WINDOW=500 250 640 480"); //posición x, y, anchura, altura.
putenv("OSG_THREADING=SingleThreaded");

/// Ayudas
viewer.addEventHandler(new osgViewer::HelpHandler(...);

/// Se crea el modelo
osgViewer::Viewer viewer(arguments);
osg::Group* root = new osg::Group;
```

- Se llaman una a una a las funciones que introducen modelos (elementos geométricos de la escena):
  - Axis que marca el origen del mundo (O_w) y crea una malla en el suelo que ayuda a diferenciar y reconocer la escena.

    ```
    //color de fondo de la escena
    color_fondo->setClearColor(osg::Vec4(1.0f,1.0f,1.0f,1.0f));
    root->addChild( crear_grid_suelo(Ah));
    ```

  - Parche de imagen de fondo: se genera un plano de fondo que le aporta realismo a la escena, situado detrás del volumen 3D:

```
//pintar plano de fondo de la escena
root->addChild( createBackground(-4500000) );
```

- Por último y más importante, se llama a la función que va a pintar todo el conjunto de polígonos que serán los voxels de la escena y representarán el volumen reconstruido. De momento esta rutina no dibujará nada en la escena porque no se ha comenzado a realizar el proceso de adquisición de datos, pero su llamada es necesaria para el correcto funcionamiento de la aplicación.

```
root->addChild( makeGalaxy(datos_pos,datos_color, 0, 1) );
```

- Las cámaras son establecidas (en posición y configuración) de manera acorde a las cámaras en el mundo real. Explicado al detalle en la siguiente sección.

```
/// Se generan los parámetros intrínsecos y extrínsecos para OpenSG
intrinsic2osg(zNear, zFar, cameraH, cameraW, KK[master_camera], intrinsic_frustrum);
extrinsic2osg(Tc_ext[master_camera],Rc_ext[master_camera],alpha);

/// Se cargan los parámetros intrínsecos y extrínsecos en la cámara OpenSG
viewer.getCamera()->setProjectionMatrixAsFrustum(intrinsic_frustrum[0],...intrinsic_frustr
viewer.getCamera()->setViewMatrix(extrinsic_osg);
```

- Otras funciones importantes que se cargan en la función main son: la optimización de la escena, la función de atención a la interrupción por teclado y la posible generación de ficheros para el análisis posterior .osg y .bmp.

```
///Función de optimización de la escena
osgUtil::Optimizer optimizer;
optimizer.optimize(root);

///Se añade el evento de atención al teclado
viewer.addEventHandler(new KeyboardEventHandler(viewer));

///Para generar el fichero .osg del modelo
osgDB::writeNodeFile(*root,"./geoemtry.osg");
viewer.setSceneData( root ); // add model to viewer.
```

- Finalmente antes de entrar la función main en un estado de reposo se crea un hilo, que es explicado a continuación y que dejará al programa en un estado continuo de captura y representación de datos.

```
pthread_create (&idHilo, NULL, funcionThread, NULL);
```

Una vez realizadas estas operaciones, el sistema queda a la espera de la interacción con el usuario y en un constante estado de adquisición, procesado y representación de información. Para ello el hilo que es lanzado (ver Figura 2.53) tiene como objetivo principal, tomar los frames adquiridos por las cámaras y procesarlos para obtener los voxels del volumen. Es importante resaltar que este hilo se ejecuta de manera cíclica continuada a la máxima velocidad posible, puesto que de él dependen de manera directa los frames por segundo de la reconstrucción. De manera esquemática, los procesos de este hilo son:

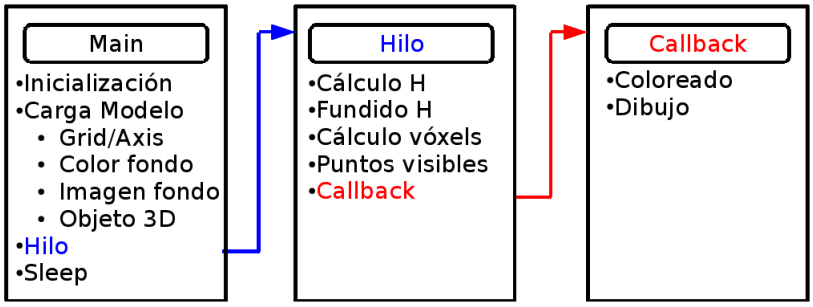

Figura 2.53: Flujograma del programa.

- Inicialización de todas las matrices necesarias para realizar las operaciones matemáticas.

```
///Inicialización de las matrices
cvInitMatHeader( &KK_cv[r], 3, 3, CV_64FC1, KK[r]);
cvInitMatHeader( &Rc_ext_cv[r], 3, 3, CV_64FC1, Rc_ext[r]);
cvInitMatHeader( &Tc_ext_cv[r], 3, 1, CV_64FC1, Tc_ext[r]);
```

- Captura de un frame segmentado y mediante la técnica del cálculo del fundido de las homografías obtener los voxels del volumen a reconstruir. Para lo cual hay que lograr la matriz H, calculada realizando operaciones matemáticas matriciales. Proyectar las imágenes segmentadas con ayuda de H y fundir los resultados para conseguir un contorno a una altura determinada. Las funciones más significativas son:

```
/// Funciones matemáticas para matrices
cvMatMulAdd( &Rc_ext_cv[camara], &incremento_h_cv, 0, &h_aux_cv );
cvAdd(&Tc_ext_aux_cv[camara], &h_aux_cv , &Tc_ext_aux_cv[camara]);
cvInvert( &H_incremento_cv ,&H_incremento_cv);

/// Proyección de la imagen segmentada con H
cvWarpPerspective( images_array[camara], imagen_origen,&H_incremento_cv);

/// Fusión (and) de las imágenes proyectadas de cada cámara
MeanImages(imagen_origen,imagen_salida,imagen_salida);
```

- Se activa la función *"Callback"* desde dentro de la función que dibuja el conjunto de polígonos. Se ocupará del coloreado y visualización, es explicada en el siguiente apartado.

```
/// Creación del callback de la escena
osg::MatrixTransform* transform = new osg::MatrixTransform();
transform->setUpdateCallback(new GalaxyCallback(10.0f));
transform->addChild(geode);
```

Por último y trabajando de manera paralela al hilo existe la función *Callback*, que es la manera que tiene OpenSceneGraph de denominar a las funciones que tienen como finalidad actualizar la geometría que se está representando. Las operaciones básicas que se llevan a cabo en este módulo son:

- Dependiendo del algoritmo de coloreado elegido, se pueden realizar operaciones de cálculo del color del vóxel. Se hacen aquí para que se ejecuten de manera paralela al hilo de cálculo del grid 3D con el fin de conseguir eficiencia máxima.

```
/// Los voxels se guardan uno a uno con su correspondiente color
vertices->push_back(osg::Vec3(vector_x,vector_y,vector_z));
colors->push_back(osg::Vec4((float)color_r,(float)color_g,(float)color_b);
```

- Son fijados parámetros propios de la representación de la escena, como la iluminación, opacidad de los elementos, primitiva a utilizar (puntos, líneas, parches ...), etc.

```
/// Muy importante, para borrar la anterior geometría antes de modificarla
galaxy->removePrimitiveSet(1);

/// Actualización de la nueva geometría
/// Admite POINTS, LINES, LINE_STRIP, POLYGON, QUADS, TRIANGLES ...
 galaxy->addPrimitiveSet(new osg::DrawArrays(osg::PrimitiveSet::POINTS ...));

/// Atributos de iluminación de la escena
set->setMode( GL_DEPTH_TEST, osg::StateAttribute::ON );
set->setMode( GL_LIGHTING, osg::StateAttribute::OFF );

///Atributos de tamaño del vóxel
osg::Point *point = new osg::Point();
point->setSize(tamano_voxel)
```

2.6.3.2.2. Definición de la cámara OpenSG. Es necesario que la visualización tridimensional de la escena que se realiza con la librería Open Scene Graph, sea idéntica a la visualización de la misma conseguida con el conjunto de cámaras, es decir, lograr una similitud realista entre la escena captada con las cámaras y la generada por ordenador. Para llevarlo a cabo es necesario que la posición y orientación de la *cámara virtual* de Open Scene Graph coincida con la de la *cámara real*, con ayuda de los parámetros extrínsecos (matrices de traslación y rotación) y que ambas tengan también los mismos parámetros intrínsecos. Open Scene Graph contiene la primitiva *setProjectionMatrixAsFrustum* que permite agregar a la cámara del visor de la imagen sus parámetros intrínsecos en forma de 6 parámetros:

```
viewer.getCamera()->setProjectionMatrixAsFrustum(p[0],p[1],p[2],p[3],p[4],p[5]);
```

Donde los parámetros pasados son:

```
p[0]=-zNear * cameraCx / cameraFx;
p[1]= zNear * (cameraW - cameraCx) / cameraFx;
p[2]=-zNear * cameraCy / cameraFy;
p[3]= zNear * (cameraH - cameraCy) / cameraFy;
p[4]= zNear;
p[5]= zFar;
```

siendo

```
double cameraCx=KK2_[2];
double cameraCy=KK2_[5];
double cameraFx=KK2_[0];
double cameraFy=KK2_[4];
```

Dado que el conjunto de parámetros intrínsecos de una cámara pueden ser descritos siguiendo la notación de Matlab como:

$$K = \begin{bmatrix} \alpha & \gamma & u_0 \\ 0 & \beta & v_0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{f}{dx} & 0 & u_0 \\ 0 & \frac{f}{dy} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.32}$$

Siendo:
f= Distancia focal.
dx= Tamaño horizontal del píxel del CCD.
dy= Tamaño vertical del píxel del CCD.
u_0= Distancia horizontal en pixels desde el origen al centro óptico.
v_0= Distancia vertical en pixels desde el origen al centro óptico.

Para fijar los parámetros extrínsecos en la cámara de OpenSceneGraph se usa la primitiva:

```
viewer.getCamera()->setViewMatrix(extrinsic_osg);
```

Siguiendo la notación clásica de la matriz de rotación **R**, que tiene como componentes los vectores unitarios de los ejes x, y, z. Es una matriz ortonormal:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2.33}$$

y la matriz de traslación **T** como:

$$T = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} \tag{2.34}$$

Es muy importante resaltar en este punto, que como se observa en la Figura 2.54 el sentido de los ejes de coordenadas que interpreta Matlab (utilizado para la calibración y cálculo de R y T) y el sentido de los ejes que interpreta Open Scene Graph no son los mismos. Para igualarlos es necesario introducir signos negativos en algunos elementos de las matrices. Así pues, la matriz "extrinsic osg" se calcula como:

```
cameraTrans.makeTranslate( Tx,-Ty,-Tz);
extrinsic_osg = cameraRotation * cameraTrans;
```

siendo cameraRotation:

$$cameraRotation = \begin{bmatrix} r_{11} & -r_{21} & -r_{31} & 0 \\ r_{12} & -r_{22} & -r_{32} & 0 \\ r_{13} & -r_{23} & -r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.35}$$

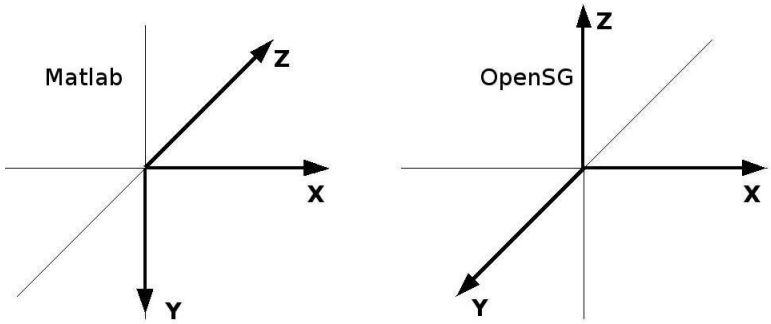

Figura 2.54: Comparación de ambos sistemas de coordenadas.

2.6.3.2.3. Creación de un escenario realista. Uno de los objetivos principales de la reconstrucción es realizarla de la manera más realista posible, para ello además de los esfuerzos realizados en la construcción, refinamiento y coloreado de la geometría, se ha incluido un plano en el fondo de la escena, con la textura original del fondo de la figura, es decir el espacio inteligente vacío o la mesa giratoria sin ningún objeto a reconstruir, con el fin de generar una imagen global mucho más completa y fiel a la realidad. El efecto de incluir este plano, queda reflejado en la Figura 2.55, donde se observa en (a) la escena real tomada por la cámara, en (b) el volumen reconstruido, en (c) el volumen reconstruido con el plano de fondo y en (d) un detalle de la escena donde se aprecia la situación del plano.

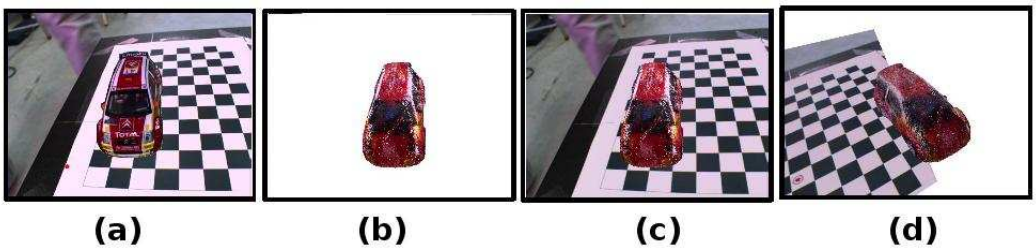

Figura 2.55: Detalles de la colocación del plano de fondo.

De manera intuitiva, se aprecia en la Figura 2.56, la situación de la cámara, los volúmenes y la cámara OpenSG:

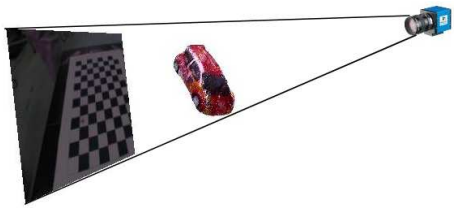

Figura 2.56: Situación del plano-volumen-cámara OpenSG.

Suponiendo que el plano tiene una anchura 'w' y una altura 'h', tendremos los siguientes puntos claramente diferenciados (teniendo en cuenta que anchura y altura están definidos con respecto al origen):
$origen = (\frac{-w}{2}, \frac{-h}{2}, z)$

$anchura = (w, 0, 0)$
$altura = (0, h, 0)$
Considerando que z es un valor negativo grande y que:

$$h = \frac{2 \cdot v_0 \cdot (-z)}{\beta} \tag{2.36}$$

$$w = \frac{2 \cdot u_0 \cdot (-z)}{\alpha} \tag{2.37}$$

Donde $u_0, v_0, \beta y \alpha$ son elementos de los parámetros intrínsecos.

Una vez calculados los puntos origen, anchura y altura, se sitúan en el espacio realizando la transformación:

$$puntoespacio = punto \cdot \begin{bmatrix} R' & 0 \\ -T'R' & 1 \end{bmatrix} \tag{2.38}$$

Nota: En este punto es muy importante tener en cuenta, como ya se describió en el apartado de "Definición de la cámara OpenSG" que si partimos de las matrices R (rotación) y T (traslación) con la notación de Matlab, hay que introducir algunos signos negativos con el fin de adaptar el sentido de los ejes de coordenadas a los ejes que utiliza OpenSG, de este modo las matrices quedan:

$$R' = \begin{bmatrix} R_{11} & R_{21} & R_{31} \\ -R_{12} & -R_{22} & -R_{23} \\ -R_{13} & -R_{23} & -R_{33} \end{bmatrix} \tag{2.39}$$

$$T' = \begin{bmatrix} R_{11} & -R_{12} & -R_{13} \end{bmatrix} \tag{2.40}$$

2.6.3.3. Arquitectura cliente-servidor para la adquisición y control.

Como se ha visto en el capítulo "Sistema distribuido de adquisición y control de múltiples cámaras", disponemos de un sistema como el mostrado en la Figura 2.57, formado por un conjunto de nodos (n servidores) que, en su configuración más básica, mediante una conexión local (1394, Firewire) realizan la adquisición de imágenes que mediante peticiones, son servidas al cliente, para su posterior procesamiento. La conexión cliente servidor se realiza a través de una red de área local, por protocolo TCP/IP mediante sockets (Figura 2.58). Un funcionamiento más avanzado será el desarrollado en el capítulo de "Diseño distribuido para la obtención de la rejilla de ocupación", donde los nodos servidores, realizarán tareas adicionales, además de la adquisición y envío de imágenes.

2.6.3.4. Medida de la temporización global del sistema.

Se va a estudiar de forma cuantitativa, y para los dos algoritmos de coloreado más óptimos sobre los que se han trabajado, el volumen de cómputo de cada parte de la aplicación y su repercusión en los tiempos de respuesta. Para hacer comparaciones relativas entre los distintos métodos de coloreado, se han hecho test de ejecución de programa para un número fijo de voxels (50.000 voxels), teniendo siempre en cuenta que el rendimiento y los tiempos dependen de forma directa de la potencia de la máquina sobre la que se ejecuta la aplicación.

2.6.3.4.1. Temporización para el cálculo de la rejilla 3D. Los procesos que intervienen en el cálculo de la rejilla 3D son los siguientes:

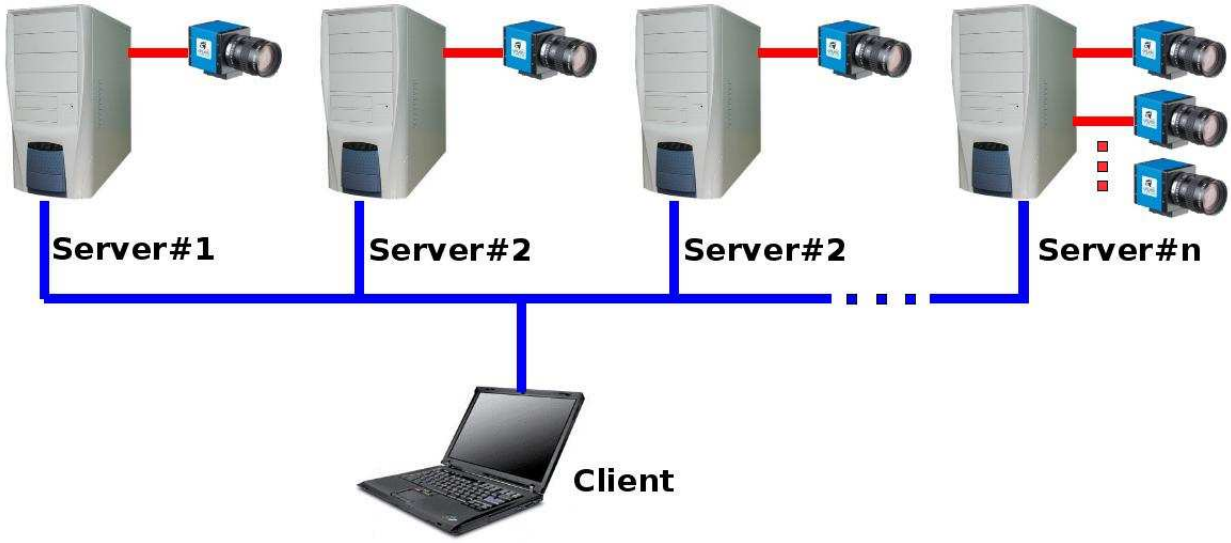

Figura 2.57: Arquitectura hardware cliente-servidor.

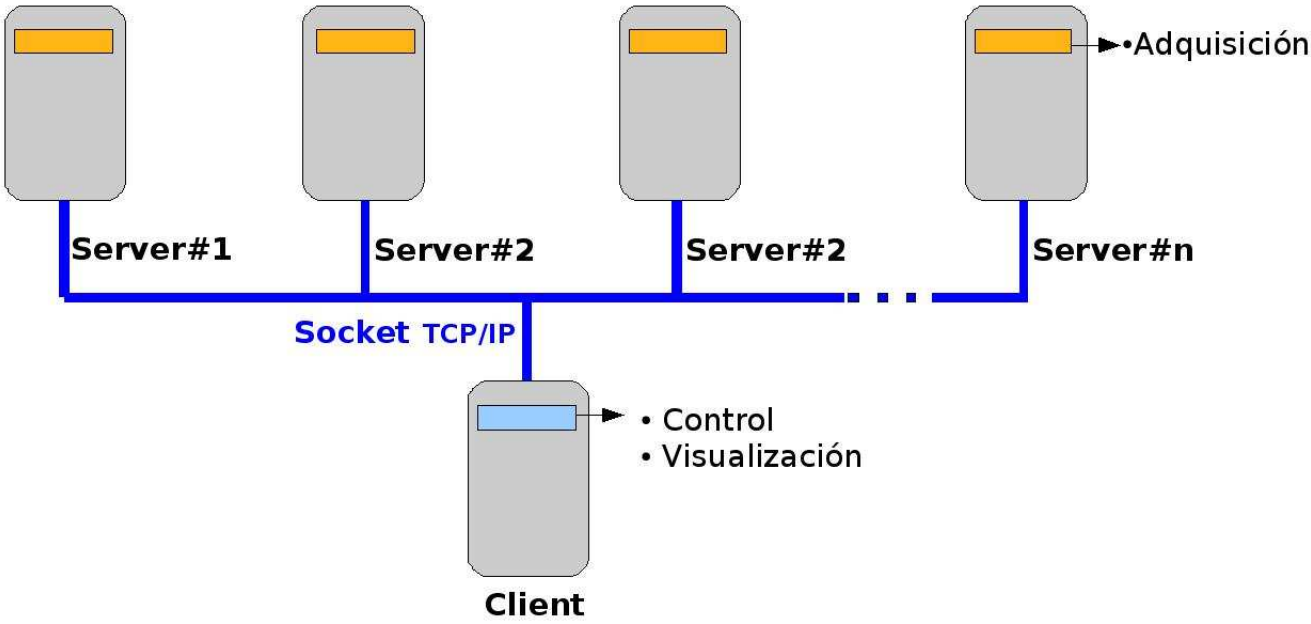

Figura 2.58: Arquitectura software cliente-servidor.

```
for(número_cámaras)
{
        //se cargan las imágenes originales de las n cámaras
        load(imagen_original[i]);
}

/// ************************************
/// *** Cálculo del contorno del objeto ***
/// ************************************

for(número_slices)
{
   for(número_cámaras)
   {
      //cálculo de la matriz H,
      //distinta para cada altura y cada cámara
      Tc_ext=Tc_ext+Rc_ext*[0;0;h];
      H=KK*[Rc_ext(:,[1:2]),Tc_ext];
      H=H*M;
      H=inv(H);

      //proyectar con la matriz H
      cvWarpPerspective(images_array[cámara],imagen_origen);

      //fundir los resultados de las matrices H
      MeanImages(imagen_origen,imagen_orig
   }

}

/// Cálculo de los voxels a partir de los contornos
```

A modo de diagrama, dividimos los procesos más importantes del cálculo de la rejilla 3D como se indica en la Figura 2.59.

Los tiempos medios obtenidos para cada una de estas partes, para una imagen de 50.000 voxels son:
Tiempo total del cálculo del grid: 11.394277s.
Tiempo de carga de imágenes: 0.018797s (Columna A).
Cálculo de H: 0.3221s (Columna B).
Tiempo de proyección con H: 10.405149s (Columna C).
Tiempo de fundido de proyecciones: 0.648231s (Columna D).

De una manera gráfica, estos tiempos se pueden representar como la Figura 2.60.

Analizando los resultados, se observa claramente como el proceso encargado de la proyección de la imagen segmentada con la transformada H (función “warp“ de OpenCV), copa más del 92 por ciento del tiempo total. Del mismo modo el proceso encargado de la fusión de las proyecciones creadas consume el 5.6 por ciento del tiempo. Por ello, es tarea prioritaria reducir los tiempos

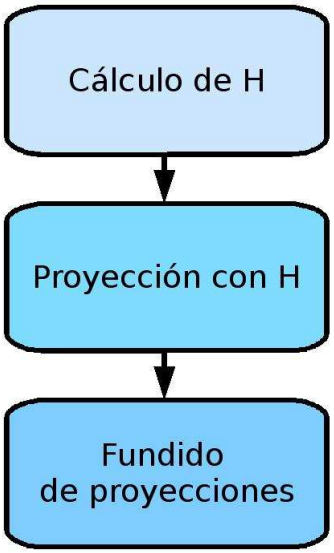

Figura 2.59: Procesos del cálculo de la rejilla 3D.

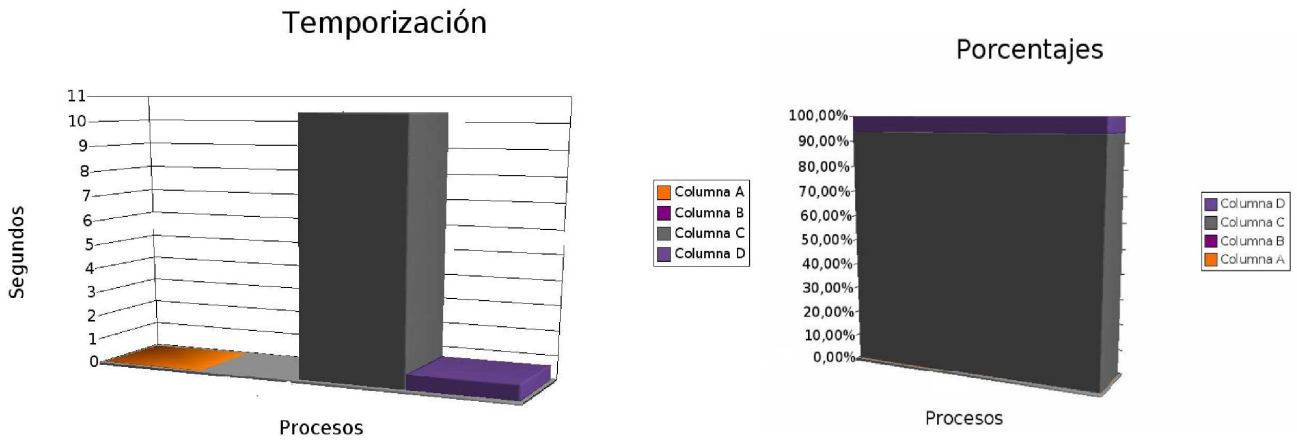

Figura 2.60: Representación de tiempos consumidos.

de la función Warp, esto se puede hacer de varias maneras:

- Con la tarjeta gráfica: La función warp es un proceso que realiza de manera directa la tarjeta gráfica de un computador. Para ello habría que buscar las primitivas que llevan a cabo esta función, y programando directamente sobre la GPU de la tarjeta realizar la operación. Por desgracia no hay mucha documentación al respecto, y aunque puede que sea la manera más óptima de realizarlo conlleva ciertas dificultades técnicas.
- Programando directamente la función "warp": OpenCV nos proporciona una función warp basada en realizar pixel por pixel a partir de la imagen destino una transformada H^{-1}, una interpolación y finalmente una transformada directa H. Estos procesos necesitan mucho volumen de cómputo y se podría intentar programar a mano una función directa y fiable.
- Dividiendo la imagen original: La función "warp" tiene un comportamiento exponencial con el número de pixels de la imagen sobre la que se aplica. Se podría dividir esta imagen en varias (2, 4...) y aplicar por separado y de manera independiente la transformación, para por último volver a juntar las distintas imágenes en una sola.

El sistema distribuido, centrará sus esfuerzos en estos puntos como se explicará a fondo en el capítulo "Diseño Distribuido para la Obtención de la rejilla de Ocupación".

2.6.3.4.2. Temporización para la texturización inversa. Este es el algoritmo que ha producido resultados de coloreado más fotorealistas, y el que más tiempo consume, no siendo factible su utilización en escenas con cierto nivel de complejidad (más de 10.000 voxels). El pseudocódigo en el que se basa este algoritmo que fue explicado en el capítulo de "Texturización basada en perspectiva inversa" es:

```
/// Previamente ha calculado la posición espacial (x,y,z)
/// de todos los voxels de la escena

/// ************************************************
/// *** Parte que calcula los voxels de la corteza ***
/// ************************************************

for(número_cámaras)
 {
 //carga de imagen segmentada vista por la cam
 load (imagen_segmentada)
 for(anchura_imagen_segmentada)
  {
        for(altura_imagen_segmentada)
        {
            if(pixel_segmentado)
            {
                for(número_voxels)
                {
                    Cálculo de T
                    Cálculo de D
                    Cálculo de L
                    Cálculo de d1
```

```
                    if(d1<umbral_d1)
                    {
                        vóxel semi-válido
                    }
                }

                for(número_voxels_semi-válidos)
                {
                    Cálculo de d2
                    if(d2<umbral_d2)
                    {
                        vóxel válido
                    }

                }
            }
        }
    }
}
```

```
/// ******************************************
/// *** Parte que asigna textura a los voxels ***
/// ******************************************
for(número_cámaras)
{
   //carga de la imagen original vista por la cam.
   load (imagen_textura)
   for(número_voxels_válidos)
   {
      //(u,v)=proyección 2D del vóxel 3D
      textura=color imagen_textura(u,v)
   }
}
```

Expresado a modo de diagrama de bloques, el algoritmo puede ser presentado como en la Figura 2.61.

Para calcular los tiempos medios obtenidos en cada una de estas partes, se ha representado una escena con 2.000 voxels, en lugar de una con 50.000 (como se hizo en el apartado anterior), puesto que los tiempos tienen una relación cuadrática con el número de voxels y cámaras y se producían unos tiempos muy elevados (alrededor de 35 minutos de cálculo). Así pues los tiempos son:

Tiempo total de la texturización del grid: 48.629256 s

Para la obtención de la corteza:
Tiempo de carga de imágenes: 0.012509s (Columna A).
Tiempo de umbralización de d1: 38.679078s (Columna B).
Tiempo de umbralización de d2: 0.151757s (Columna C).

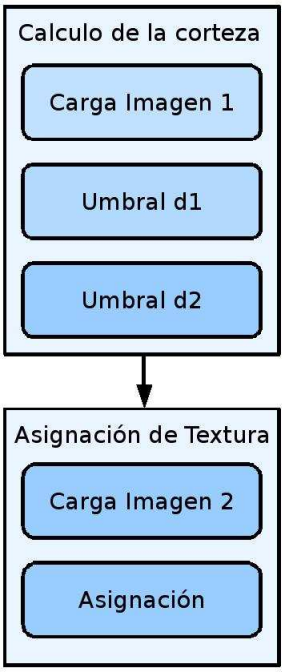

Figura 2.61: Procesos del cálculo de la texturización basada en perspectiva inversa.

Tiempo de bucles e iteraciones conjuntas para el cálculo de d2 y d1: 9.7859s (Columna D).

Tiempo total para el coloreado de los voxels de la corteza: 0.066061s.

Para la asignación de textura:
Tiempo de carga de imágenes: 0.064857s (Columna A).
Tiempo de asignación: 0.001204s (Columna B).

De una manera gráfica, estos tiempos se pueden representar en las siguientes figuras: en la Figura 2.62, el tiempo de los procesos que intervienen en la obtención de la corteza; en la Figura 2.63, los tiempos de los procesos en la asignación de la textura a la corteza y en la Figura 2.64, una comparativa entre el tiempo de cálculo de la corteza y el de aplicarla un coloreado.

Analizando los resultados, se observa que para obtener los voxels que forman parte de la corteza, el tiempo consumido para el cálculo de d1 y d2 es extremadamente alto, ya que existe una relación cuadrática entre el número de voxels de la escena y las cámaras de la misma. El hecho de que el umbralizado de d2 consuma un 90 por ciento menos que el umbralizado de d1, es debido a que este último es aplicado a un 90 por ciento menos de puntos, es decir, tras pasar la primera criba (la más costosa) el resto del proceso es relativamente rápido.

Respecto a los tiempos consumidos durante el coloreado de la corteza, más del 95 por ciento del tiempo se invierte en la apertura de la imagen (obtención del frame), que se verá solucionado por el sistema distribuido. Por último resaltar que el tiempo de coloreado es despreciable frente al tiempo de obtención de la corteza (Figura 2.64).

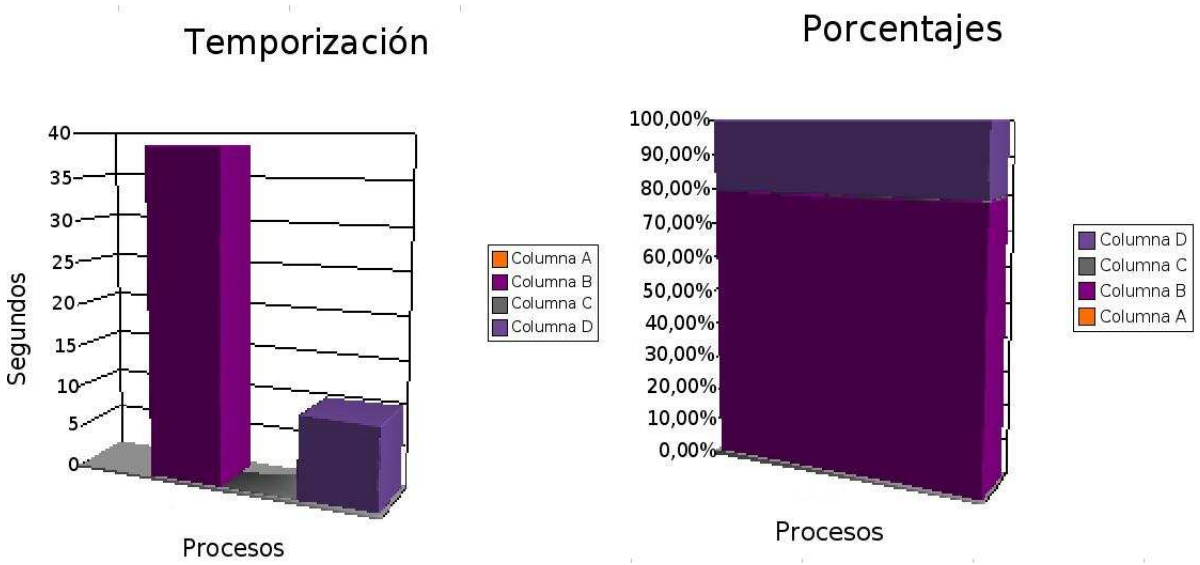

Figura 2.62: Representación de tiempos consumidos para la obtención de la corteza.

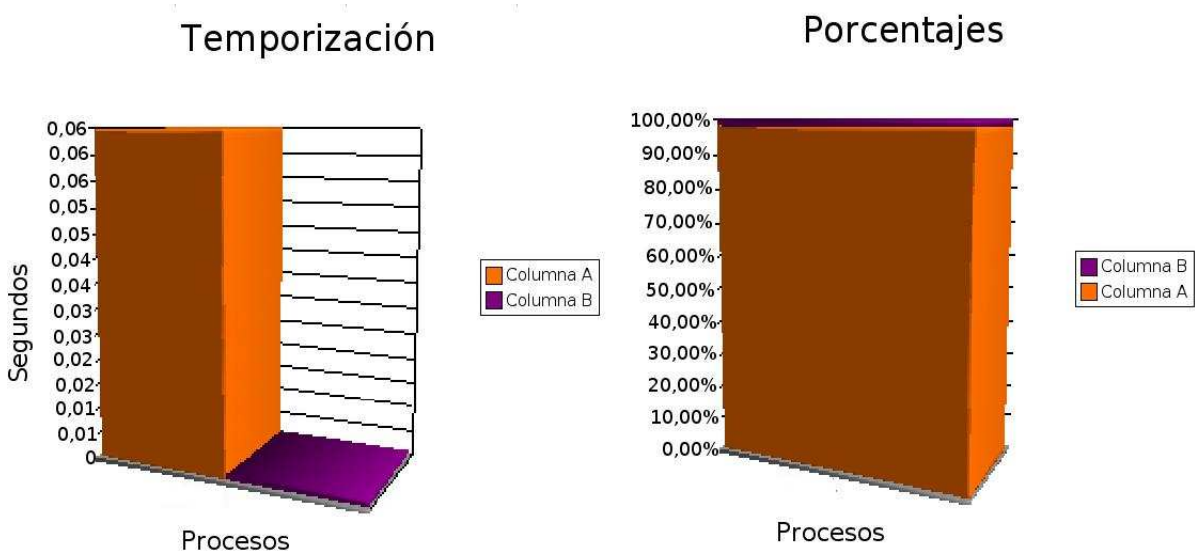

Figura 2.63: Representación de tiempos consumidos para el coloreado de la corteza.

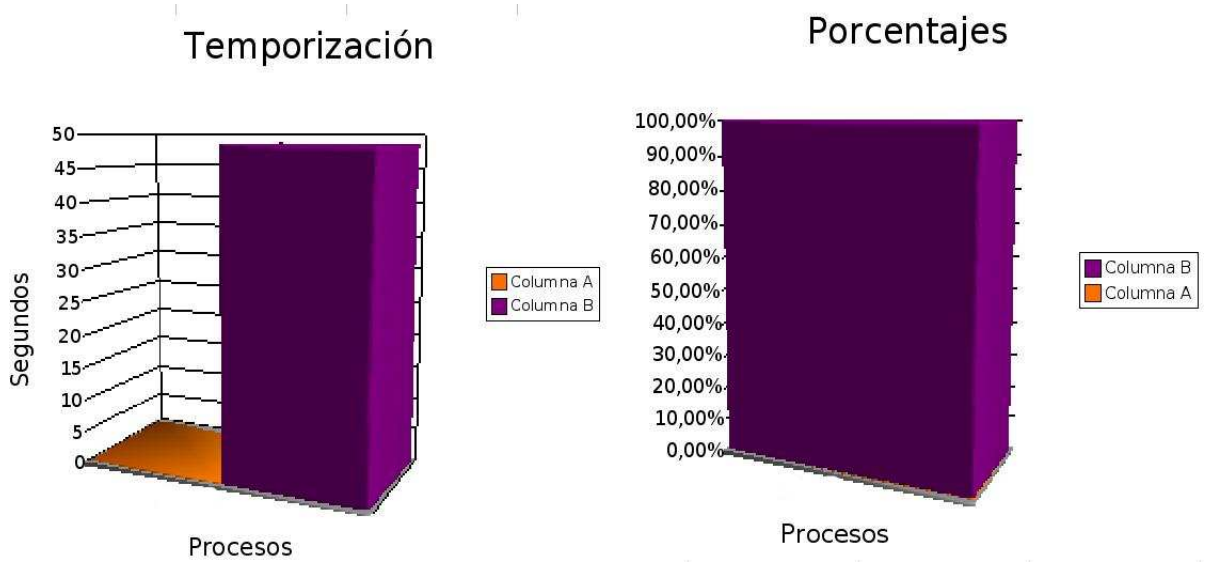

Figura 2.64: Tiempo de obtención de corteza (Columna A) VS coloreado (Columna B).

2.6.3.4.3. Temporización del Algoritmo del Pintor. Como se ha explicado en el capítulo "Coloreado por el Algoritmo del Pintor", las tareas principales que intervienen en este proceso, pueden ser explicadas en forma de pseudocódigo de la siguiente manera:

```
/// Previamente se calcula la posición espacial (x,y,z)
/// de todos los voxels de la escena

for(número_cámaras)
{
        // se inicializa a un valor conocido (-2) los arrays
        // de voxels y de profundidad de los voxels
        array_cam[][][]=-2;
        array_profundidad[][][]=-2;
}

/// ************************************************
/// *** Cálculo los voxels más cercanos a la cámara ***
/// ************************************************

for(número_cámaras)
{
   for (número_voxels)
   {
      //cálculo de la profundidad 'z'
      [x;y;z]=[Rc_ext1*[X;Y;Z]+Tc];
      profundidad_z_voxel=z;

      if(array_profundidad[número_cámara][x][y]==-2)
      {
         array_profundidad[][][]=profundidad;
```

```
            array_cam[][][]=índice_voxel;
        }

        else if(array_profundidad[][][]>profundidad)
        {
            array_profundidad[][][]=profundidad;
            array_cam[][][]=índice_voxel;
        }

    }
}
```

```
/// ************************************************************
/// *** Parte que asigna a los "voxels cercanos" su textura ***
/// ************************************************************

for (número_cámaras)
{   // se carga la imagen original vista por la cámara
    // que aportará la textura
    load(imagen_textura);

    // se carga la imagen segmentada
    load(imagen_segmentada);

    for (anchura_imagen_textura)
    {
        for (altura_imagen_textura)
        {
            if (array_cam[número_cámara][anchura][altura]!=-2)
            {
                //se carga el vóxel apuntado (índice) por array_cam
                (x,y,z)=proyección 2D del vóxel 3D

                if(píxel[anchura][altura]_imagen_segmentada>umbral)
                {
                    vóxel_válido para la representación.
                    asignación de color.
                    color[número_cámara][índice_voxel]=color_calculado;

                }
            }
        }
    }
}
```

A modo de diagrama de bloques, se puede dividir este proceso de la manera en que se muestra en la Figura 2.65.

Para calcular los tiempos consumidos en cada parte, se ha representado una escena con 50.000 voxels. De esta manera, los tiempos medios medidos son:

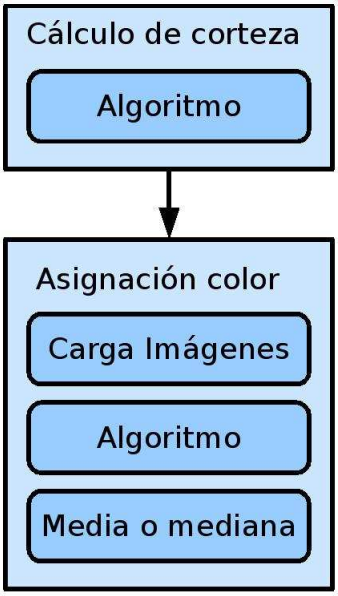

Figura 2.65: Procesos principales en el Algoritmo del Pintor.

Tiempo total para el coloreado por el Algoritmo del Pintor: 0.466496s.
Tiempo de cómputo del algoritmo primero (corteza): 0.450571s (Columna A).
Tiempo de carga de las imágenes: 0.078172s (Columna B).
Tiempo de cómputo del algoritmo segundo (colorea): 0.050950s (Columna C).
Tiempo de cómputo media: 0.058828s (Columna D).
Tiempo de cómputo mediana: 0.036265s (Columna E).

A raíz de los resultados obtenidos podemos generar las gráficas de la Figura 2.66.

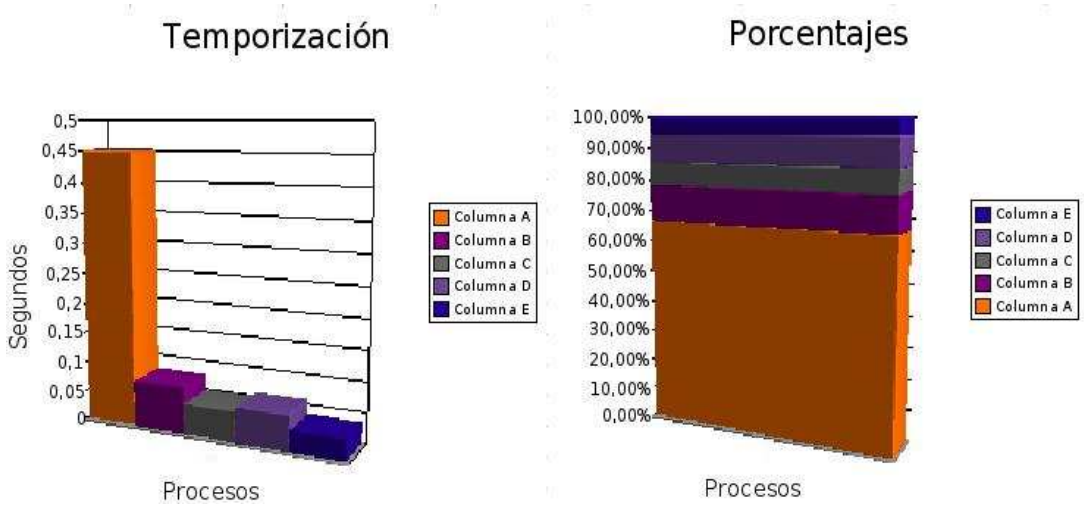

Figura 2.66: Consumo de tiempo de los procesos en el Algoritmo del Pintor.

Analizando la Figura 2.66, se observa que éste método es el mejor, pues es capaz de realizar el coloreado de todos los puntos de la escena compleja de 50.000 voxels en menos de 500 ms,

respondiendo a una relación lineal con el número de voxels de la escena y las cámaras de la misma. Al igual que en los otros algoritmos probados, la parte del proceso que más tiempo consume (67 por ciento del total), es aquel en el que se calculan los puntos de los objetos más cercanos a la cámara, y que se puede asemejar con el cálculo de la corteza. El resto de las partes tienen un balance parejo de temporización, no existiendo diferencia realmente importante en términos computacionales entre realizar una media o una mediana del color de los voxels.

2.6.3.4.4. Tiempos totales. En este último apartado, se van a relacionar los tiempos del programa completo, teniendo en cuenta todas las partes. Para esto se ha elegido la escena modelo de 50.000 voxels y el coloreado por el Algoritmo del Pintor, ya que realizando un coloreado con la texturización basada en perspectiva inversa, se obtienen unos tiempos tan altos que no merece la pena realizar el experimento. Así pues, el tiempo que tarda la aplicación en hacer todas las inicializaciones previas, necesarias para el correcto funcionamiento de Open Scene Graph es de aproximadamente 20ns, tiempo despreciable si además consideramos que esta parte se ejecuta una única vez al lanzar el programa. En la Figura 2.67 se relacionan el tiempo de cálculo de la rejilla de ocupación 3D y el tiempo de su texturización. Cabe indicar, que el comportamiento ideal es aquel en el que ambos procesos se ejecutan en tiempos similares, puesto que se lanzan en hilos diferentes y trabajan de manera paralela. El diseño distribuido se centrará en el cálculo del grid, para rebajar el porcentaje de consumo de tiempo (en un sistema no distribuido llega a ser del 95 por ciento). Tiempo de cálculo del grid 3D: 11.3942s (Columna A).
Tiempo de coloreado con Algoritmo del Pintor: 0.46649s (Columna B).

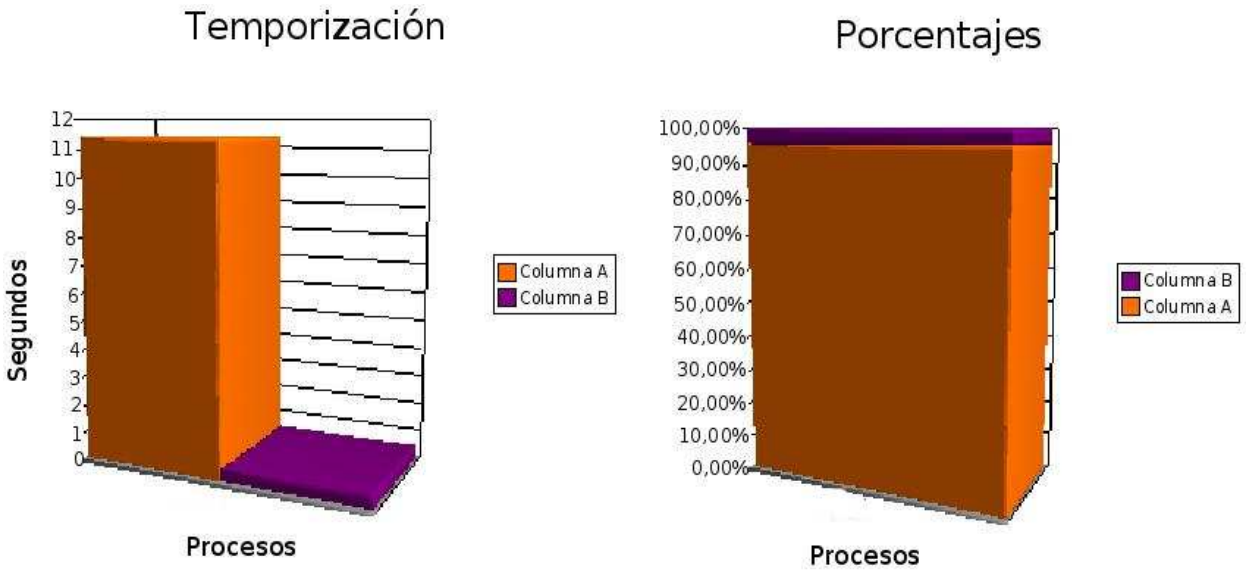

Figura 2.67: Consumo de tiempos totales de la aplicación completa.

2.6.3.5. Diseño distribuido para la obtención de la rejilla de ocupación.

En este apartado, se explicará la arquitectura software para que la ejecución de la aplicación se realice de manera óptima. Para la obtención de la rejilla de ocupación tridimensional y su coloreado, se ha elegido un diseño distribuido basado en nodos, ya que como se ha estudiado en el capítulo de "Medida de la Temporización Global del Sistema", el programa requiere un volumen

de cómputo muy elevado que permite ser distribuido en varios módulos para su procesado en paralelo. De esta forma y como se indica en la Figura 2.68, partimos de una serie de nodos y un cliente que se conecta a ellos.

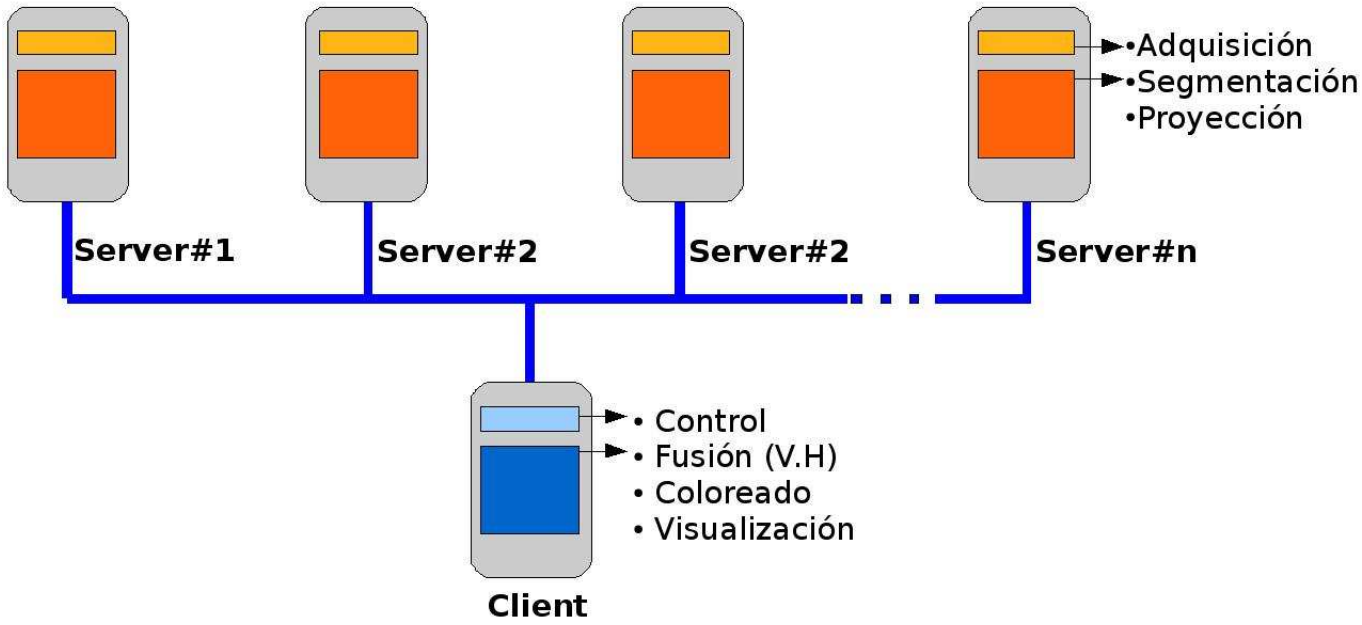

Figura 2.68: Arquitectura software cliente-servidor.

Los procesos que se llevan a cabo en cada servidor son los siguientes:

1. **Adquisición:** Cada nodo tiene conectado una o varias cámaras mediante un enlace 1394, que se encargará de inicializar los periféricos de manera adecuada y de realizar la captura o adquisición de las imágenes.

2. **Segmentación:** Cada nodo es también responsable de aplicar la segmentación a las imágenes adquiridas, diferenciando de esta manera los objetos a representar del resto (fondo) de la imagen.

3. **Proyección:** Consiste en el cálculo de la Homografía y proyección de la imagen segmentada haciendo uso de la misma. Como se vio en el capítulo dedicado al cálculo de tiempos en la aplicación, la tarea de la proyección de la imagen segmentada haciendo uso de la matriz H, es el proceso que consume más tiempo. Implementando un sistema distribuido para esta tarea, podemos dividir el tiempo entre los distintos nodos, haciendo así una utilización eficiente del sistema. Estas imágenes segmentadas y proyectadas, serán aquellas que se le entregarán al servidor.

Los procesos que se llevan a cabo en el cliente son:

1. **Control:** De manera general, podemos denominar así a la labor de control de flujo del programa, llamando a las distintas funciones de manera ordenada y adecuada y realizando la comunicación con los nodos servidores de un modo correcto.

2. **Fusión:** El servidor se encarga, una vez tomadas las imágenes proyectadas de los distintos nodos servidores, de realizar el fundido u operación “and” para obtener el contorno de la escena a representar a una altura dada.

3. **Coloreado:** Una vez realizadas todas las operaciones necesarias para conseguir el grid de ocupación 3D del objeto u objetos a representar, el computador cliente realiza el coloreado o texturización de la imagen, tratando cada vóxel de manera independiente.

4. **Visualización:** Finalizadas todas las tareas previas, se visualiza la escena deseada, con las proporciones y características deseadas.

2.7. Conclusiones y trabajos futuros.

Como conclusiones se puede afirmar que se han alcanzado con éxito los objetivos iniciales planteados. Se han conseguido reconstrucciones volumétricas en tres dimensiones partiendo del uso de varias cámaras. A lo largo del proyecto se ha justificado el uso de homografías para el cálculo de la rejilla de ocupación tridimensional. Del mismo modo se han probado y estudiado diferentes técnicas de coloreado, haciendo uso de la más eficiente. Para conseguir esto ha sido necesario gestionar los recursos hardware y software, optimizando siempre el tiempo de cómputo con el objetivo de conseguir en el futuro un sistema de similares características y en tiempo real.

2.7.1. Líneas futuras de trabajo.

En cuanto a posibles líneas de trabajo, para completar o continuar el presente proyecto, se plantean varios caminos a seguir:

- **Sistema en tiempo real:** La más inmediata e interesante es hacer que la aplicación lleve a cabo la tarea en tiempo real. Para ello hay que mejorar los tiempos en el coloreado de la escena y lo que es más importante, mejorar el tiempo de creación de la rejilla de ocupación tridimensional, cuyo principal cuello de botella se encuentra en la proyección con H de la imagen segmentada. Dependiendo de la aplicación para la que se diseñe el sistema, una posible mejora es trabajar con imágenes de 320x240, en lugar de las de 640x480 con las que se trabaja en este proyecto. Para lo cual la matriz H debería sufrir una transformación para amoldarse a imágenes de menor tamaño haciendo:

$$H = H' \cdot \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.41}$$

  De esta manera, el algoritmo de proyección de la imagen segmentada que es la parte de la aplicación que más tiempo consume, debería tardar 4 veces menos; puesto que este tiempo depende cuadráticamente del tamaño de la imagen.

- **Subdividir las capturas:** Otra opción es mediante tratamiento con OpenCV, “recuadrar” o diferenciar el objeto a reconstruir del resto de la imagen y hacer la proyección con H sólo de la parte que nos interesa y no de la imagen completa. En escenas complejas densamente pobladas, hay que enmarcar cada objeto por separado y aplicarle de manera independiente la proyección con H. El último paso sería volver a juntar en la escena cada objeto ya tratado por separado.

- **Subdividir el espacio:** Dividiendo la totalidad del espacio a reconstruir en 8 cubos, eliminando aquellos en los que no se encuentra parte alguna del objeto a reconstruir. Así se siguen manteniendo aquellos cubos en los que si hay parte del objeto, para después

volverlos a subdividir el 8 cubos más, y así sucesivamente hasta formar la geometría como se indica en la Figura 2.69. Es una técnica que puede acelerar y mejorar la representación de los voxels.

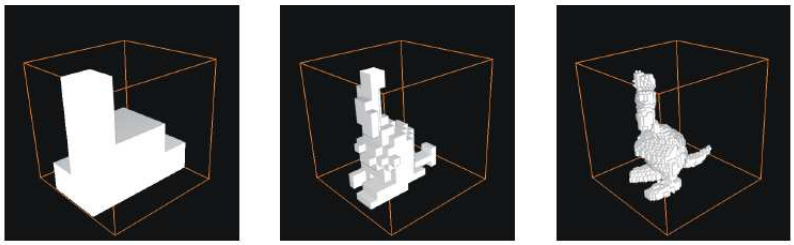

Figura 2.69: Ejemplo de refinado octree.

- **Mejora del coloreado:** Uno de los caminos más evidentes para mejorar la texturización y la carga computacional de la aplicación, pasa por el uso activo de la tarjeta gráfica. Es decir utilizar las operaciones de proyección mediante homografía que ésta es capaz de realizar de manera automática, y mejorar el proceso de coloreado haciendo uso del z-buffer para averiguar qué voxels se encuentran en primer plano de manera inmediata.
- **Mallado del volumen:** Por último y no menos importante, resaltar la necesidad de un algoritmo de mallado del volumen reconstruido. De esta forma a partir de la nube de voxels se puede generar un objeto sólido mallado que mejoraría en muchos aspectos la aplicación:
    - En primer lugar, haciendo mucho "más ligera" y robusta la visualización con OpenSG, ya que se reduciría el volumen de voxels a representar. Como se muestra en la Figura 2.70, los parches generados en el mallado simplifican la escena. Además existen importantes líneas de investigación de técnicas de reducción de puntos en la malla, para que partiendo de un volumen muy denso de puntos, donde muchos de estos no aportan información, se eliminen aquellos innecesarios optimizando la escena, como se ilustra en la Figura 2.71.

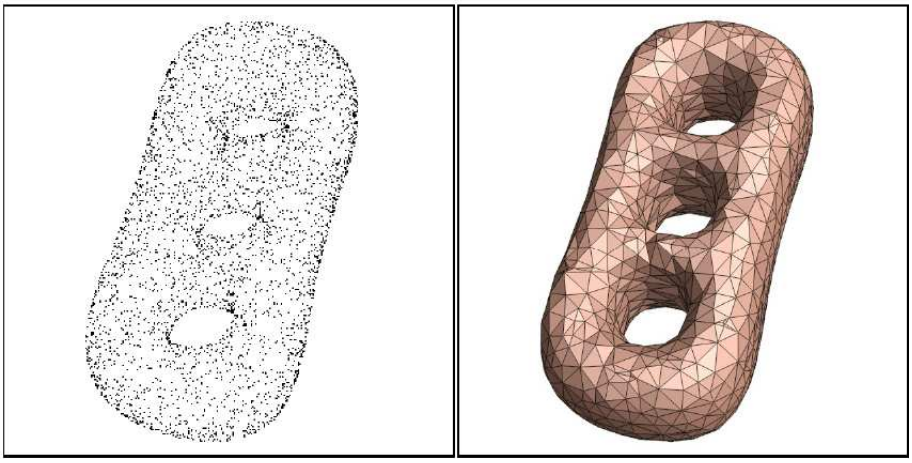

Figura 2.70: Mallado de una nube de puntos.

    - El coloreado podría hacerse más sencillo ya que calculando la normal de cada parche o "semiplano" generado por el mallado y comparándolo con la normal de la cámara,

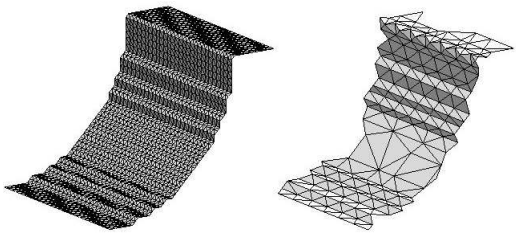

Figura 2.71: Refinado del mallado.

se calcula de manera inmediata si la cámara “ve” ese parche y le puede aplicar color o no. Adicionalmente, esta técnica nos aporta datos sobre el grado de visión del color.

Capítulo 3

Manual de Usuario

3.1. Manual

En esta sección se resumirán las principales funciones que se han desarrollado para la aplicación.

Esta función se utiliza para adaptar el conjunto de parámetros intrínsecos según la notación de Matlab, pues el programa que se utiliza para calcularlos, a la notación de OpenSG, que es la herramienta utilizada para la representación 3D.

```
1
2  /// ***************************************************
3  /// *** intrinsic2osg function                      ***
4  /// *** generate the frustrum from the intrinsic matrix ***
5  /// ***************************************************
6
7  int intrinsic2osg(float zNear,float zFar,float cameraH,float cameraW,
8                    double intrinsecos[9],double intrinsic_frustrum[6])
9  {
10         double cameraCx=KK[master_camera][2];
11         double cameraCy=KK[master_camera][5];
12         double cameraFx=KK[master_camera][0];
13         double cameraFy=KK[master_camera][4];
14
15         intrinsic_frustrum[0]=-zNear*cameraCx/cameraFx;
16         intrinsic_frustrum[1]= zNear*(cameraW-cameraCx)/cameraFx;
17         intrinsic_frustrum[2]=-zNear*cameraCy/cameraFy;
18         intrinsic_frustrum[3]= zNear*(cameraH-cameraCy)/cameraFy;
19         intrinsic_frustrum[4]= zNear;
20         intrinsic_frustrum[5]= zFar;
21         return 0;
22 }
```

Esta función se utiliza para adaptar el conjunto de parámetros extrínsecos según la notación de Matlab, pues el programa que se utiliza para calcularlos, a la notación de OpenSG, que es la herramienta utilizada para la representación 3D.

```
1
2  /// ***************************************************
3  /// *** extrinsic2osg function generate the extrinsic ***
4  /// ***  values to the OSG from R and T matrices      ***
5  /// ***************************************************
6
7  int extrinsic2osg(double Tc_ext2_[],double Rc_ext2_[],float alpha)
8  {
9          cameraRotation(0,0)=Rc_ext[master_camera][0];
10         cameraRotation(1,0)=Rc_ext[master_camera][1];
11         cameraRotation(2,0)=Rc_ext[master_camera][2];
12         cameraRotation(3,0)=0.0;
13         cameraRotation(0,1)=-Rc_ext[master_camera][3];
14         cameraRotation(1,1)=-Rc_ext[master_camera][4];
15         cameraRotation(2,1)=-Rc_ext[master_camera][5];
16         cameraRotation(3,1)=0.0;
```

```
17          cameraRotation(0,2)=-Rc_ext[master_camera][6];
18          cameraRotation(1,2)=-Rc_ext[master_camera][7];
19          cameraRotation(2,2)=-Rc_ext[master_camera][8];
20          cameraRotation(3,2)=0.0;
21          cameraRotation(0,3)=0.0;
22          cameraRotation(1,3)=0.0;
23          cameraRotation(2,3)=0.0;
24          cameraRotation(3,3)=1.0;
25
26          incremento(0,0)=cos(alpha*3.141592/180.0);
27          incremento(0,1)=sin(alpha*3.141592/180.0);
28          incremento(0,2)=0.0;
29          incremento(0,3)=0.0;
30          incremento(1,0)=-sin(alpha*3.141592/180);
31          incremento(1,1)=cos(alpha*3.141592/180);
32          incremento(1,2)=0.0;
33          incremento(1,3)=0.0;
34          incremento(2,0)=0.0;
35          incremento(2,1)=0.0;
36          incremento(2,2)=1.0;
37          incremento(2,3)=0.0;
38          incremento(3,0)=0.0;
39          incremento(3,1)=0.0;
40          incremento(3,2)=0.0;
41          incremento(3,3)=1.0;
42
43          cameraTrans.makeTranslate( Tc_ext[master_camera][0],
44         -Tc_ext[master_camera][1],-Tc_ext[master_camera][2]);
45          extrinsic_osg =   incremento * cameraRotation * cameraTrans;
46          return 0;
47  }
```

Funcion utilizada para realizar la operación “and”, entre dos imágenes, tiene como parámetros de entrada las dos imágenes sobre las que se desea realizar la operación y sobre la tercera imagen devuelve el resultado.

```
1
2   /// ****************************
3   /// *** Mean_Images function  ***
4   /// ****************************
5
6   void MeanImages(IplImage *src1,IplImage *src2, IplImage *dst)
7   {
8           for(unsigned i=0;i<(dst->width*dst->height);i++)
9           dst->imageData[i]=((unsigned char)src1->imageData[i]&
10                             (unsigned char)src2->imageData[i]);
11  }
```

Función de ordenación de un array de elementos, utilizada para realizar la mediana del color.

```
1
```

```
2  /// ****************************
3  /// *** función ordenar array  ***
4  /// ****************************
5
6  int ordenar(const void *a,const void *b)
7  {
8          if(*(int *)a<*(int *)b)
9          return(-1);
10                 else if(*(int *)a>*(int *)b)
11                 return(1);
12                         else
13                         return(0);
14 }
```

Capítulo 4

Pliego de condiciones

4.1. Requisitos de Hardware

Para el funcionamiento de las aplicaciones descritas en el proyecto se necesita disponer, al menos, del siguiente material:

- Computador cliente, PC Intel Pentium D 3.00GHz, 4GB RAM.
- Tarjeta gráfica nVidia GeForce 6600 GT.
- Cámaras ImagingSource DFK 21BF04 (4).
- Soportes omnidireccionales para las cámaras.
- Computadores servidores, Mac Mini 1.83 Ghz (4).
- Conjunto de cables de red ethernet y FireWire 1394.

4.2. Requisitos de Software

Es necesario el software indicado, con todas las dependencias resueltas que cada programa requiera:

- Sistema operativo GNU/Linux Ubuntu 7.04
- Librerías OpenCv 1.0
- Librerías OpenSeceneGraph 2.0

Capítulo 5

Presupuesto

5.1. Presupuesto

El presupuesto de ejecución material consta de tres elementos: el coste de mano de obra por tiempo empleado, el coste de equipos y el coste del material utilizado.

5.1.1. Coste de Equipos

- Cliente, PC Intel Pentium D 3.00GHz, 4GB RAM ..... 960 €
- Tarjeta gráfica nVidia GeForce 6600 GT ..... 199 €
- Cámara ImagingSource DFK 21BF04 x 4 ..... 340 x 4= 1360 €
- Soporte cámara omnidireccional ..... 8.3 x 4= 33 €
- Servidores, Mac Mini 1.83 Ghz ..... 480 x 4= 1920 €
- Impresora HP Laserjet 2100 ..... 90 €

5.1.2. Coste por tiempo empleado

- **Ingeniero Electrónico**
  - Coste por hora ..... 70 €
  - Total horas empleadas ..... 640 horas
  - **Coste total. 44.800 €**
- **Mecanógrafo**
  - Coste por hora ..... 12 €
  - Total horas empleadas ..... 40 horas
  - **Coste total. 480 €**

5.1.3. Coste total del presupuesto de ejecución material

- Coste total del material ..... 4.562 €

- Coste total de la mano de obra ..... 45.280 €

- **El coste total de ejecución material asciende a 49.800 €**

5.1.4. Gastos generales y beneficio industrial

En este apartado se incluyen los gastos generados por las instalaciones donde se ha realizado el proyecto más el beneficio industrial. Para estos conceptos se estima el 25 % del presupuesto de ejecución del proyecto.

- El valor de los gastos generales y beneficio industrial asciende a 12.450 €

5.2. Presupuesto de ejecución por contrata

Este concepto incluye el presupuesto de ejecución material y los gastos generales y beneficio industrial.

- Presupuesto de ejecución por contrata 62.250 €

5.2.1. Coste total

- Subtotal 62.250 €

- 16 % IVA 9.960 €

- **TOTAL 72.000 €**

El importe total asciende a: **Setenta y dos mil euros.**

Alcalá de Henares a de de 2008

Fdo: Javier Rodríguez López

Ingeniero Electrónico

Capítulo 6

Bibliografía

Bibliografía

[1] M. Weiser, "The computer for the 21st century," Master's thesis, SIGMOBILE Mob. Comput. Commun. Rev., 3(3):3-11., 1999.

[2] ——, "Some computer science problems in ubiquitous computing." Master's thesis, Communications of the ACM, b 1999.

[3] ——, "Parc builds a world saturated with computation," Master's thesis, Science (AAAS), pages 3-11, a 1993.

[4] M. H. Coen, "Design principles for intelligent environments," Master's thesis, In Proceedings of the fifteenth national, tenth conference on Artificial Intelligence Innovative Applications of Artificial Intelligence, pages 547-554, Menlo Park, CA, USA. American Association for Artificial Intelligence., 1998.

[5] A. Pentland, "Smart rooms," Master's thesis, Scientific American, 1996.

[6] D. Se, S. Lowe and J. Little, "Vision-based mobile robot localization and mapping using scale-invariant features," Master's thesis, In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Seoul, Korea., 2001.

[7] H. I. T. Sogo and T. Ishida, "Acquisition of qualitative spatial representation by visual observation," Master's thesis, In IJCAI99, 1999.

[8] H. H.-L. J.H. and A.Ñ, "Self-identification of distributed intelligent networked device in intelligent space," Master's thesis, In ICRA, pages 4172-4177, 2003.

[9] L. J.-M. K. and H. H, "Intelligent space and mobile robots," Master's thesis, The Industrial Information Technology Handbook, pages 1-15, 2005.

[10] S. P.-E. M. and D. R, "Mephisto. A modular and extensible path planning system using observation," Master's thesis, Lecture Notes in Computer Science, 1542:361-375, 1999.

[11] V. J. M. U. J. M. M. H. A. Álvarez F. J. García J. J. Marziani C. D and A. D., "Improvement of ultrasonic beacon-based local position system using multi-access techniques." Master's thesis, WISP05, page CDROM Edition., 2005.

[12] B. E. P. D.Ñ. I. and M. M., "Multiple camera calibration using point correspondences oriented to intelligent spaces," Master's thesis, TELEC-04 International Conference, 2004.

[13] P. D.-S. E. and M. M., "Simultaneous localization and structure reconstruction of mobile robots with external cameras," Master's thesis, ISIE05, 2005.

[14] A. Hoover and B. D. Olsen, "Sensor network perception for mobile robotics." Master's thesis, ICRA, pages 342-347, 2005.

[15] ——, "A real-time occupancy map from multiple video streams," Master's thesis, In ICRA, pages 2261-2266, 1999.

[16] B. D. Olsen and A. Hoover, "Calibrating a camera network using a domino grid," Master's thesis, PR, 34(5):1105-111, 2001.

[17] E. Kruse and F. M. Wahl, "Camera-based observation of obstacle motions to derive statistical data for mobile robot motion planning," Master's thesis, ICRA, pages 662-667., 1998.

[18] B. Baumgart, "Geometric modeling for computer vision." PhD thesis, Stanford University, 1994.

[19] W. Martin and J. Aggarwal., "Volumetric descriptions of objects from multiple views." Master's thesis, IEEE Transactions on Pattern Analysis and Machine Intelligence, 5(2):150-174, March 1983.

[20] Y. Kim and J. Aggarwal, "Rectangular parallelepiped coding: A volumetric representation of three dimensional objects." Master's thesis, IEEE Journal of Robotics and Automation,RA-2:127-134, 1986.

[21] M. Potmesil., "Generating octree models of 3D objects from their silhouettes in a sequence of images," Master's thesis, Computer Vision, Graphics and Image Processing, 40:1-20, 1987.

[22] y. S. A. H.Ñoborio, S. Fukuda, "Construction of the octree approximating three-dimensional objects by using multiple views." Master's thesis, IEEE Transactions Pattern Analysis and Machine Intelligence, 10(6):769-782, Noviembre 1988.

[23] N. A. y J. Veenstra., "Generating octrees from object silhouettes in orthographic views." Master's thesis, IEEE Transactions Pattern Analysis and Machine Intelligence, 11(2):137-149, Febrero 1989.

[24] K. S. y A. Pujari., "Volume intersection with optimal set of directions." Master's thesis, Pattern Recognition Letter, 12:165-170, 1991.

[25] R. Szeliski., "Rapid octree construction from image sequences." Master's thesis, Computer Vision, Graphics and Image Processing: Image Understanding, 58(1):23-32, Julio 1993.

[26] A. Laurentini., "The visual hull : A new tool for contour-based image understanding." Master's thesis, In Proceedings of the Seventh Scandinavian Conference on Image Analysis, 993-1002, 1991.

[27] ——, "The visual hull concept for silhouette-based image understanding." Master's thesis, IEEE Transactions Pattern Analysis and Machine Intelligence, 16(2):150-162, Febrero 1994.

[28] ——, "How far 3D shapes can be understood from 2D silhouettes," Master's thesis, IEEE Transactions on Pattern Analysis and Machine Intelligence, 17(2):188-195, 1995.

[29] ——, "The visual hull of curved objects." Master's thesis, In Proceedings of International Conference on Computer Vision (ICCV'99), Septiembre 1999.

[30] M. O. y T. Kanade., "A multiple-baseline stereo." Master's thesis, IEEE Transactions on Pattern Analysis and Machine Intelligence, 15(4):353-363, 1993.

[31] K. K. y S. Seitz., "A theory of shape by space carving." Master's thesis, International Journal of Computer Vision, 38(3):199-218, 2000.

[32] y. P. G. S. Moezzi, L. Tai, "Virtual view generation for 3D digital video." Master's thesis, IEEE Computer Society Multimedia, 4(1),, Enero-Marzo 1997.

[33] I. K. y D. Metaxas., "3D human body model acquisition from multiple views." Master's thesis, International Journal on Computer Vision, 30(3):191-218, 1998.

[34] Q. D. y O. Faugeras., "3D articulated models and multi-view tracking with silhouettes." Master's thesis, In Proceedings of International Conference on Computer Vision (ICCV'99), Septiembre 1999.

[35] A. B. y A. Laurentini, "Non-intrusive silhouette based motion capture." Master's thesis, In Proceedings of the Fourth World Multiconference on Systemics, Cybernetics and Informatics SCI 2001, páginas 23-26, Julio 2000.

[36] L. M. y. S. G. C. Buehler, W. Matusik, "Creating and rendering imagebased visual hulls. Technical Report MIT-LCS-TR-780,," Master's thesis, MIT, 1999.

[37] S. B. Kong-man (German) Cheung and T. Kanade, "Shape-From-Silhouette Across Time," Master's thesis, Carnegie Mellon University, 1999.

[38] K. Cheung, "Visual Hull Construction, Alignment and Refinement for Human Kinematic Modeling, Motion TRacking and Rendering," Master's thesis, PhD thesis, Carnegie Mellon University, 2003.