INGENIERÍA DE TELECOMUNICACIÓN - INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Escuela Técnica Superior de Ingeniería de Telecomunicación

Curso académico 2014-2015

Proyecto Fin de Carrera

Realidad aumentada con interacción física desde una cámara móvil usando JdeRobot

Autor: Daniel Azuara Pérez

Tutor: José María Cañas Plaza

Es difícil vencer a quien nunca se rinde

Agradecimientos

Me gustaría dar mi agradecimiento a todas las personas que me han apoyado de un modo u otro a lo largo de estos años de Universidad.

Primero me gustaría dar las gracias a José María Cañas Plaza por su esfuerzo, por su pasión en todo lo que hace y por esas reuniones capaces de resolver cualquier duda y de las que salía con energías para comerme el mundo.

También agradecer a todos los compañero del grupo de Robótica por compartir sus experiencias y estar siempre dispuestos a resolver dudas. Es un placer formar parte de este ecosistema de motivación y talento.

Por supuesto agradecer a mi familia por su apoyo constante, en especial a mis padres, que siempre han sabido aconsejarme y motivarme para conseguir cualquier cosa que me proponga. Agradecer sus buenos consejos, sus ánimos y su confianza. Lo dan todo por mí y no hay palabras que describan lo mucho que lo aprecio. Y como no, a Yaz, por no dejarme rendirme nunca, por sacarme siempre una sonrisa y por hacerme cada día mejor persona.

Por último, pero no menos importante, agradecer a todos los amigos que he ido conociendo durante todos estos años, con los que he compartido penas y alegrías, y con los que espero seguir compartiendo risas y buenos momentos durante mucho, mucho tiempo.

¡Muchas gracias a todos!

Resumen

En los últimos años los avances en técnicas y algoritmos de autolocalización visual y la mejora de prestaciones de dispositivos como *smartphones*, *tablets* e incluso nuevos dispositivos como *gafas inteligentes* hacen que el desarrollo de aplicaciones de realidad aumentada sea un campo de creciente interés.

El trabajo que se recopila en esta memoria pretende adentrarse en las bases para el desarrollo de aplicaciones de realidad aumentada. Para ello se ha implementado un algoritmo de autolocalización propio y experimentado con librerías gráficas y de simulación de físicas para desarrollar una aplicación que muestra las posibilidades de estas tecnologías. En concreto la aplicación muestra un personaje virtual que camina hacia una pelota ficticia hasta golpearla, para que después ésta colisione con un objeto real y acabe cayéndose de la mesa en la que se encuentra.

Para el desarrollo de la aplicación se ha empleado la plataforma JdeRobot, se ha utilizado lenguaje de programación C++ y se han usado múltiples librerías, siendo *Ogre*, *Bullet*, *ApilTags* y *OpenCV* las más importantes.

Índice general

Índice de figuras

Capítulo 1

Introducción

La vista no es sólo uno de los sentidos más importantes para el ser humano, si no que además es uno de los que más curiosidad despierta. Las tareas relacionadas con la visión requieren gran parte de la actividad cerebral y nos permiten obtener alrededor de un 80 % de las sensaciones, emociones e información que obtenemos a través de nuestros sentidos.

No todas las experiencias que recibimos a través de nuestros ojos son reales, en ocasiones nuestro cerebro nos *engaña* y nos hace pensar que ilusiones de geometría, iluminación y/o perspectiva se presentan delante nuestra con tanta veracidad como el resto de elementos que nos rodean.

Esta incertidumbre que en ocasiones experimentamos entre lo real y lo ficticio, más allá de ser considerada una limitación de nuestra percepción o un defecto, puede ser considerada como una puerta al mundo de la imaginación y de la ilusión. Como ejemplos con cierta historia, tenemos los trucos de ilusionismo o el arte de ilusiones ópticas. Más recientemente tenemos el caso de los videojuegos, en el que cada vez se consigue de forma más acertada que escenas virtuales se parezcan a escenas reales. Además la tendencia es que cada vez la experiencia sea más inmersiva, bien mediante el uso de la realidad virtual, conseguir la sensación de estar dentro del juego simulando todo un entorno virtual a nuestro alrededor, o bien mediante el uso de la realidad aumentada, la introducción de elementos imaginados

en nuestra percepción de la realidad.

Si bien parece que uno de los campos en los que más futuro puede tener este contrapunto entre realidad y ficción es el de los videojuegos o el entretenimiento en general, lo cierto es que también existen muchas posibilidades en el sector del marketing o en el de las aplicaciones móviles. Son tanto la proyección a futuro de las tecnologías, como la fascinación que despierta el jugar con ilusiones, las motivaciones que han llevado a cabo el desarrollo de este Proyecto Fin de Carrera.

En este primer capítulo se pretende dar una visión general del contexto del proyecto, el cuál se encuadra dentro del campo de la visión artificial, la autolocalización visual y la realidad aumentada. Además se busca dar una idea del estado actual de estas tecnologías, así como mostrar sus posibilidades.

1.1. Visión artificial

La visión artificial se puede entender al menos desde dos puntos de vista. Uno es el análisis del sistema visual humano para obtener un modelo computacional que lo pueda reproducir. Otro es el diseño de sistemas autónomos que reproduzcan algunas de las tareas que realiza el sistema visual humano.

El origen de este campo de estudio se puede situar en 1960 cuando Larry Roberts [8] realiza su tesis doctoral en el MIT y estudia las posibilidades de extraer información 3D a partir de imágenes 2D de bloques tomadas desde diferentes perspectivas. A partir de este trabajo se desarrollan más investigaciones sobre la visión artificial aplicada a un mundo formado por bloques geométricos. Más tarde se llega a la conclusión de que para poder aplicar estas ideas a imágenes reales es necesario trabajar en técnicas que permitan *tratar* estas imágenes antes de procesarlas.

El abaratamiento del *hardware* y el aumento exponencial de la capacidad de proce-

samiento han favorecido el desarrollo de este campo de investigación, pudiendo observar múltiples y variadas aplicaciones:

- **Industriales**: En sistemas de producción industrial las tareas de inspección y control de calidad son fundamentales y mediante sistemas de visión artificial se logra cumplir un máximo de exigencia sin afectar al ritmo de producción, dado la rapidez de procesado que se pude alcanzar. En la Figura 1.1 podemos ver un ejemplo de un sistema de inspección de botellas y de un robot de envasado automático.

(a) Sistema de control de calidad

(b) Robot ABB de envasado automático

Figura 1.1: Aplicaciones en sistemas de producción industrial

- **Vigilancia y seguridad**: Desde vigilancia en grandes eventos mediante identificación de personas hasta seguridad privada en el hogar, pasando por sistemas de prevención de accidentes de tráfico o monitorización de personas mayores que viven solas.

- **Identificación**: En un mundo cada vez más digital, el reconocimiento biométrico (de cara, huellas, ojo, etc.) permite olvidarnos de usar llaves, tarjetas de identificación o contraseñas.

- **Medicina**: Uno de los campos más interesantes y posiblemente en el que más se investiga es el de la medicina. Las posibilidades son muchas, detección y control de

tumores, análisis de imágenes para modelado 3D, asistencia en operaciones de cirugía o estudio de enfermedades como el *Alzheimer* o la *Schizophrenia*, por mencionar algunas. En la Figura 1.2 podemos ver el sistema TrueVision 3D [1] asitiendo a un cirujano en una operación de corrección de astigmatismo.

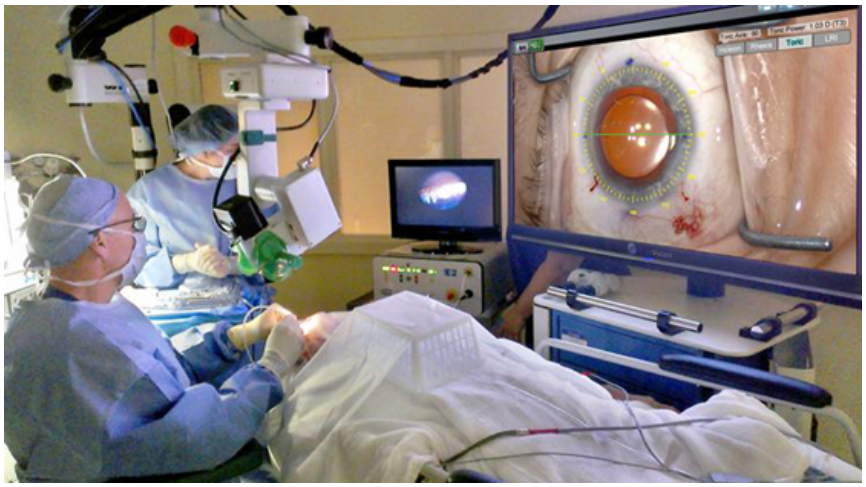


Figura 1.2: Asistencia en una operación de corrección de astigmatismo

- **Control de tráfico**: Como mejora a los sistemas de control de tráfico, poco a poco se van incorporando funciones como la lectura de matrículas, reconocimiento de tipos de vehículos, detección de velocidad y análisis de puntos con alto riesgo de accidentes.

- **Deportes**: La aplicación más conocida es, sin duda, el ojo de halcón empleado en tenis para saber con precisión cuál a sido la trayectoria de la pelota en una jugada. Más recientemente se emplea la visión artificial para el análisis biomecánico de deportistas o para la repetición de jugadas de béisbol cambiando el ángulo de visión, como se puede ver en la Figura 1.3.

- **Robótica**: Una de las tareas más complicadas para un robot es interpretar el entorno que le rodea a través de sus sensores, como por ejemplo una cámara de vídeo. Con información visual se puede localizar, detectar objetos e incluso seguir a los mismos.

[1] http://www.truevisionsys.com/

(a) Sistema de análisis biomecánico en el mundial de natación de Barcelona en 2013

(b) Sistema freeD de repetición de jugadas

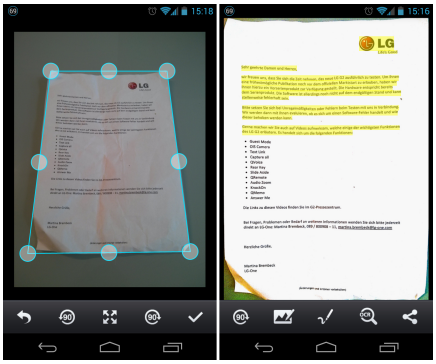
Figura 1.3: Aplicaciones en el mundo del deporte

- **Entretenimiento**: En el sector de los videojuegos aporta una forma diferente de interactuar con los juegos, reconociendo los movimientos del jugador o llevando el escenario de la partida al mundo real con la realidad aumentada. En la retransmisión de eventos deportivos y espectáculos también se puede ver el uso de la visión por computador.

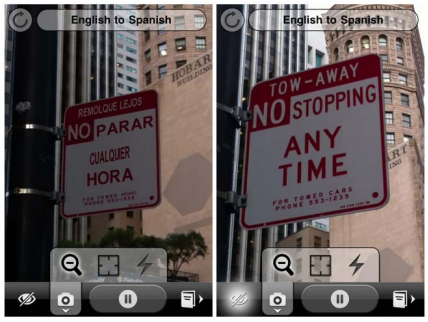
- **Aplicaciones móviles**: Con la llegada de los *smartphones* y cientos de aplicaciones que hacen de ellos auténticas navajas suizas, el uso de la visión artificial ha visto muchas posibilidades en este sector. En la Figura 1.4 tenemos los ejemplo de CamScanner, un scanner de bolsillo, y de Word Lens que traduce cualquier texto al que enfoquemos con la cámara de nuestro móvil, mostrando el resultado sobre la propia imagen empleando realidad aumentada.

1.2. Autolocalización visual

Dentro de la visión artificial podemos encontrar el campo de la autolocalización visual, el cual estudia cómo un sistema con una cámara puede localizarse, es decir calcular su posición y orientación, dentro de un entorno que pude ser conocido a priori o no. Este problema se suele querer resolver para un sistema móvil que emplea la autolocalización para orientarse y navegar por el escenario en el que se encuentra. Para ubicarse dentro de



(a) Aplicación CamScanner



(b) Aplicación Word Lens

Figura 1.4: Aplicaciones para móvil

un escenario el sistema puede utilizar balizas visuales que sean conocidas y le sirvan de referencia o bien puede obtener una serie de puntos característicos a partir de las imágenes que procesa e ir fabricando un mapa con el que orientarse.

Como cabe esperar la autolocalización visual emplea como única fuente de información imágenes de cámara, frente a la autolocalización genérica que puede emplear diversos tipos de sensores (laser, ultrasonidos, encoders, etc) y además combinarlos para obtener mejores resultados. Si bien el problema se complica al disponer únicamente de datos visuales, también se obtienen determinadas ventajas. Por un lado un sensor de cámara está mucho más extendido y se puede encontrar en dispositivos variados como teléfonos móviles, vehículos, drones o *wearables*, frente a otros sensores que sólo se encuentran en robots o sistemas similares. Por otro lado las cámaras son sensores más baratos que otros, como por ejemplo los láseres.

Además, la cámara o cámaras (en el caso de disponer más de una en el sistema) puede ser RGBD o simplemente RGB. La diferencia consiste en que las cámaras RGBD obtienen información de profundidad mediante el uso infrarrojos, además de la imagen en RGB, y si bien esto supone una ventaja, ésta se pierde cuando el escenario es exterior y hay luz natural.

Veamos ahora algunas de las técnicas de autolocalización visual que existen agrupán-

dolas en estas tres familias:

1.2.1. Odometría visual

La odometría visual se basa en la estimación de la posición mediante el cálculo de incrementos entre los distintos fotogramas que se van obteniendo y consigue evitar algunos de los inconvenientes de la odometría convencional como problemas de precisión debidos a los sensores que se emplean (por ejemplo un sensor de rotación introduce error si la rueda patina), o la limitación por el sistema de locomoción empleado (la odometría tradicional se suele emplear en sistemas con ruedas).

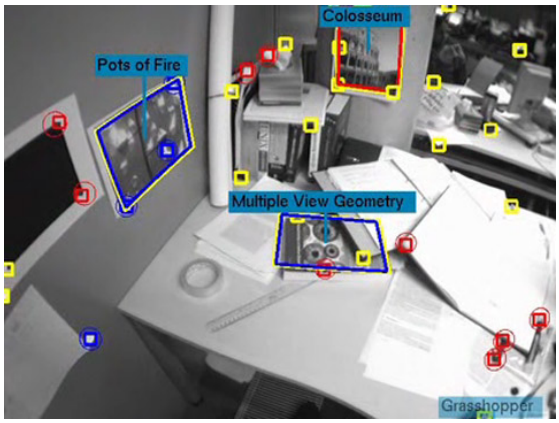
El método más empleado para odometría visual es el basado en la extracción de puntos característicos, que permiten comparar dos imágenes, obtener correspondencias (por correlación) entre puntos de las mismas y aplicar alguna técnica para calcular la estimación del movimiento relativo entre ellos, como por ejemplo un filtro de Kalman.

Cabe mencionar la implementación del algoritmo llamado *SVO: Fast Semi-Direct Monocular Visual Odometry* por Christian Forster, Matia Pizzoli y Davide Scaramuzza [2] del grupo de Robótica y Percepción de la Universidad de Zurich. Más adelante podremos ver su uso en autolocalización de cuadricópteros en la Figura 1.7.

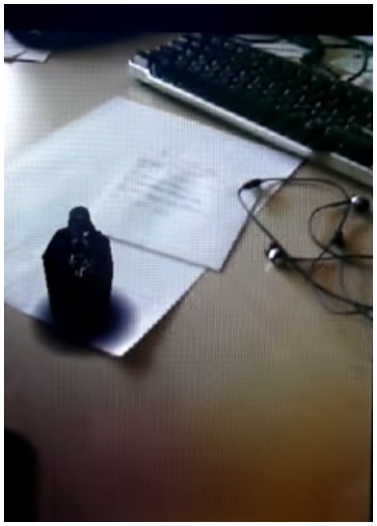
1.2.2. VisualSLAM

Visual SLAM es una técnica de autolocalización visual que trabaja los problemas de la localización y de la construcción del mapa al mismo tiempo. Para poder estimar la posición y orientación de la cámara que proporciona la imagen, primero se obtienen una serie de puntos característicos mediante un algoritmo de procesado 2D (como FAST) con los que se va construyendo un mapa. Una vez que se tiene esta información se puede aplicar algún algoritmo de procesado 3D para estimar la posición 3D de la cámara tomando el mapa como sistema de referencia.

De la mano del trabajo *monoSLAM* de Andrew J. Davison [2] [1] se propone resolver este problema a través de una única cámara RGB como sensor y realizando el mapeado y la localización simultáneamente. El algoritmo propuesto por Davison utilizaba un filtro extendido de Kalman para estimar la posición y orientación de la cámara, así como la posición de una serie de puntos en el espacio 3D. Originalmente para determinar la posición inicial de la cámara, es necesario dotar al filtro de Kalman de información a priori con la posición en 3D de al menos 3 puntos. A partir de ese momento el algoritmo es capaz de situar la cámara 3D y de generar nuevos puntos para crear el mapa y servir como apoyo a la propia localización de la cámara. Podemos ver un ejemplo de la implementación de Davison en la Figura 1.5a, donde se emplea el algoritmo en una aplicación de reconocimiento de objetos



(a) Ejemplo con monoSLAM



(b) Ejemplo con PTAM

Figura 1.5: Diferentes implementaciones de VisualSLAM

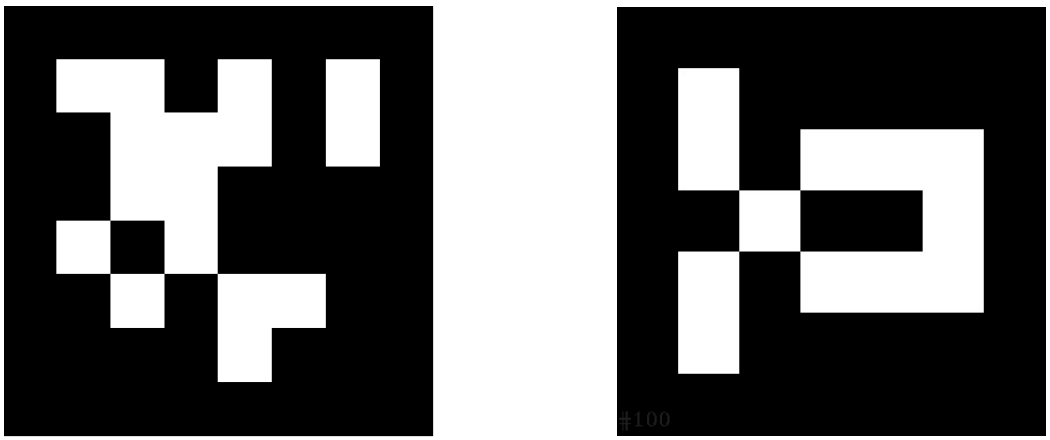
Por otro lado, cabe también destacar la trascendencia que ha tenido el trabajo PTAM [4] propuesto por George Klein [3] [4] como algoritmo alternativo a *monoSLAM*. El mayor problema de los algoritmos basados en monoSlam es que su tiempo de cómputo aumenta exponencialmente con el número de puntos. Esto ocurre ya que en cada iteración, el al-

[2]http://www.doc.ic.ac.uk/ ajd/
[3]http://www.robots.ox.ac.uk/ gk/

goritmo realiza tanto el mapeado como la localización. George Klein sugiere abordar este problema separando el mapeado de la localización, de tal modo que solo la localización debe funcionar en tiempo real mientras que el mapeado puede funcionar de modo asíncrono, ya que parte de la idea de que sólo es necesario funcionar en tiempo real en la parte de la localización. Este algoritmo hace uso de *keyframes*, es decir, fotogramas claves que se utilizan tanto para la localización como el mapeado. También cabe mencionar que PTAM hace uso de una técnica de optimización mediante ajuste de haces. Podemos ver una aplicación de realidad aumentada que prueba el algoritmo en la Figura 1.5b.

1.2.3. Localización visual con balizas

Para simplificar el problema de la autolocalización, se puede partir de conocer el mapa del entorno y trabajar únicamente en la localización. Algunos elementos de este mapa serán balizas, es decir objetos reconocibles que se pueden utilizar para saber la ubicación dentro del mapa. Para que las diferentes balizas sean útiles es importante poder distinguir bien entre ellas y confundirlas el menor número de veces posible.



(a) Baliza de AprilTags (b) Baliza de ArUco

Figura 1.6: Ejemplos de balizas

Como balizas, también llamados marcadores o etiquetas, se pueden utilizar imágenes que guarden un identificador único empleando algún tipo de codificación. Como ejemplos

tenemos los marcadores de AprilTags o ArUco (Figura 1.6) que son códigos de barras bidimensionales, similares a los QR, diseñados para almacenar menos información que estos pero de forma más robusta. Existen librerías que permiten detectar estos marcadores dentro de un imagen y obtener su posición y orientación respecto de la cámara. Esta información puede ser utilizada por un algoritmo para calcular una estimación de la localización de la cámara, sabiendo la posición (y orientación) de las balizas dentro del escenario.

1.2.4. Ejemplos de aplicaciones relevantes

En cuanto a las aplicaciones para la autolocalización visual existen muchas y además han ido evolucionando bastante en los últimos tiempos, pero quizás lo más interesante es ver algunas de las más actuales.

Una primera aplicación es la autolocalización en un tipo de robots que ha ganado una gran popularidad en los últimos años, los voladores, es decir los drones. Muchos de ellos disponen de cámaras integradas y procesadores por lo que un algoritmo de autolocalización visual puede ser muy apropiado, otra opción es utilizar un dispositivo móvil acoplado al cuadricóptero para que lleve a cabo la autolocalización y el manejo. De esta forma se puede utilizar este tipo de robots para labores de vigilancia, mensajería o búsqueda de personas con un coste económico relativamente bajo. En la Figura 1.7 podemos ver un dron que estima su posición 3D con una técnica de odometría visual.

Otra aplicación interesante y reciente es su uso en domótica, como sistema de localización para robots domésticos. Como ejemplo tenemos la aspiradora Dyson 360 [4] que posee una percepción visual de 360 grados y un algoritmo de autolocalización que emplea para crear un mapa de su entorno y desplazarse por este, como podemos ver en la Figura 1.8.

Por último comentar un uso con bastante potencial y que es el empleado en este proyec-

[4] https://www.dyson360eye.com/

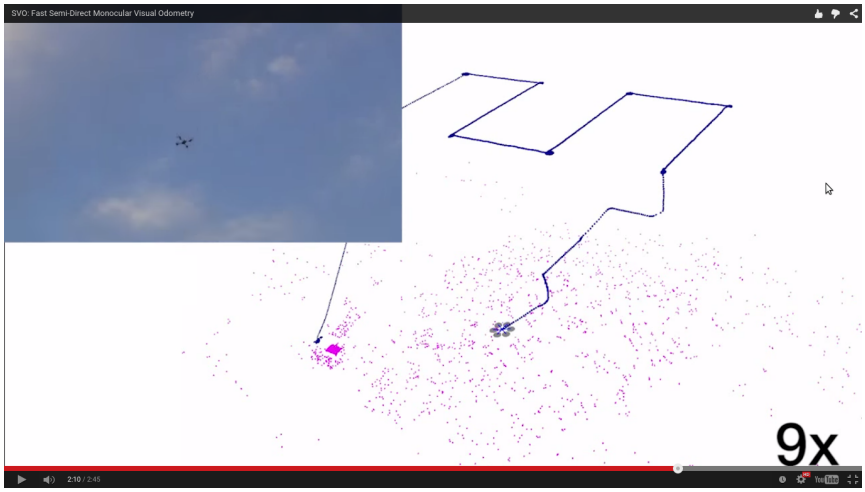


Figura 1.7: Drone autolocalizado mediante SVO (*Semi-direct Visual Odometry*)

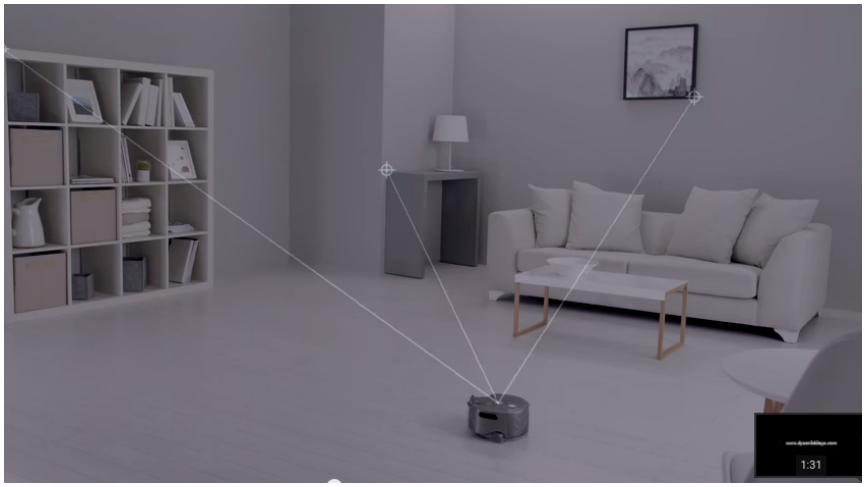


Figura 1.8: Aspiradora Dyson 360

to, la localización visual como base para aplicaciones de realidad aumentada, trataremos más este tema en el siguiente apartado y a lo largo del resto de la memoria.

1.3. Realidad Aumentada

La realidad aumentada consiste en enriquecer un flujo de imágenes de cámara con elementos virtuales, de modo que realidad y ficción se unan. Para que está ilusión sea creíble es imprescindible basarse en un buen sistema de autolocalización visual.

Partiendo de una estimación de la posición y orientación de la cámara se puede super-

poner un mundo 3D virtual que es visualizado desde el mismo punto que la cámara real. Además, teniendo el conocimiento del mundo que rodea a la cámara real se puede dotar de comportamientos físicos a elementos reales, para que puedan interactuar con los objetos del mundo virtual.

Para construir una aplicación de realidad aumentada hay que combinar varias tecnologías. Por un lado hay que reproducir el mundo virtual con alguna tecnología de *renderizado* 3D, como puede ser *OpenGL* u *Ogre*. También es indispensable tener algún tipo de autolocalización, como ejemplos tenemos las librerías de *ARToolKit* o *ArUco* que ofrecen la funcionalidad mínima necesaria para hacer aplicaciones de AR, como podemos ver en la Figura 1.9. Por último, si se quiere que los objetos ficticios tengan un comportamiento real, se puede utilizar alguna tecnología de simulación de físicas como la que emplea la biblioteca Bullet.

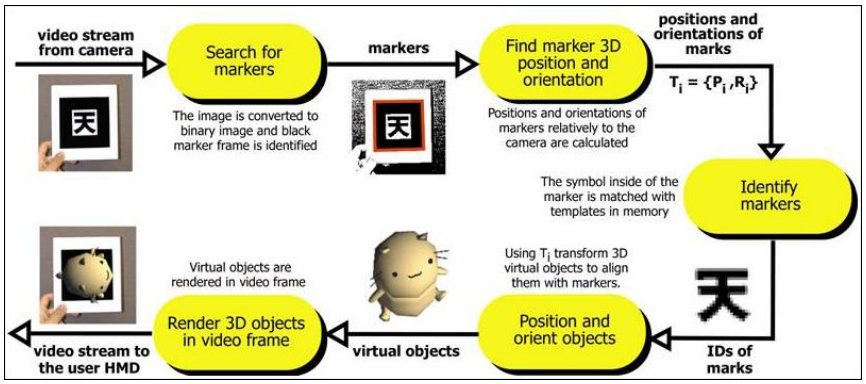


Figura 1.9: Funcionamiento de ARToolKit

Podemos clasificar la aplicaciones de realidad aumentada por diferentes características. En un principio las primeras aplicaciones mostraban el efecto de realidad aumentada sin mover la cámara, como en los anuncios que se pueden ver en los partidos de fútbol proyectados sobre el campo, y al ir mejorando las técnicas de autolocalización visual o al tener datos de sensores como los accelerómetros de los móviles, se empezaron a desarrollar aplicaciones con cámara móvil. También podemos clasificar las aplicaciones en función de si los elementos virtuales que muestran son estáticos o dinámicos, en especial en el ámbito

de lo videojuegos se deseará que los objetos se muevan y estén animados, para lo que será necesario el uso de potentes librerías gráficas. Otra clasificación posible es en función de si los elementos ficticios cumplen comportamientos físicos o si solamente son un efecto gráfico, de nuevo en el sector de los videojuegos puede ser muy interesante que los elementos virtuales interaccionen no sólo entre ellos, si no incluso con objetos reales.

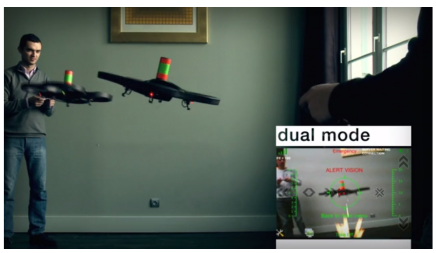
Centrándonos en las aplicaciones, una de las que antes apareció es el uso de realidad aumentada en videojuegos. Uno de los juegos más antiguos es *Invizimals* que fue lanzado en 2009 para la videoconsola PSP de Sony, basado en marcas visuales. Con la mejora de los dispositivos móviles en los últimos años podemos ver juegos de realidad aumentada para Android o iOS como por ejemplo AR Defender. La Figura 1.10 nos muestra ambos ejemplos.

(a) Invizimals, para PSP

(b) AR Defender, para iOS

Figura 1.10: Videojuegos de realidad aumentada

Otro forma de jugar es la que propone el AR Drone 2.0 de Parrot que permite diferentes modos de juegos con realidad aumentada, como se puede ver en la Figura 1.11, añadiendo así un incentivo más para adquirir uno de los robots que están más de moda.



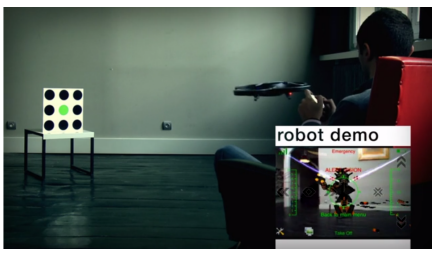
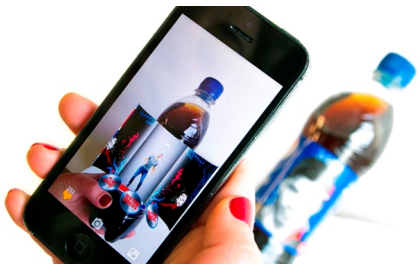


Figura 1.11: Modos de Juego con el AR Drone 2.0 de Parrot

Para el mundo de la publicidad la realidad aumentada se presenta como una nueva forma de atraer a los clientes. Una de las formas que tiene de llamar la atención del consumidor es mediante el *escaneo* de un anuncio, o del mismo producto, para ver una animación de realidad aumentada. De este modo se consigue que la interacción entre publicidad y público sea mucho más activa y al mismo tiempo obtener realimentación de cómo de efectiva está siendo una campaña de publicidad. Ejemplos actuales son los de la app Blippar, que colabora con diversas marcas, o la campaña de Toyota en España (Figura 1.12).



(a) Escaneo de una botella de bebida, con la app blippar

(b) Campaña de Toyota basada en AR

Figura 1.12: Ejemplos de realidad aumentada en publicidad

En el sector del turismo podemos encontrar el uso de la realidad aumentada para orientar a los turistas de una forma diferente a la hora de recorrer un nuevo lugar, tanto para buscar restaurantes y lugares de ocio [5] como para descubrir la historia y cultura del mismo. [6]

En cuanto al entretenimiento encontramos algunas aplicaciones en la retransmisión de espectáculos deportivos que emplean la realidad aumentada para innovar y sorprender al espectador. En la Figura 1.13 vemos cómo se usa para mostrar el marcador de una forma diferente y para nombrar a los invitados famosos, durante uno de los eventos más vistos en todo el mundo, el partido de los *All Stars*.

Otra aplicación con mucho potencial es la de emplear la realidad aumentada en edu-

[5] Layar - www.layar.com
[6] Across Air

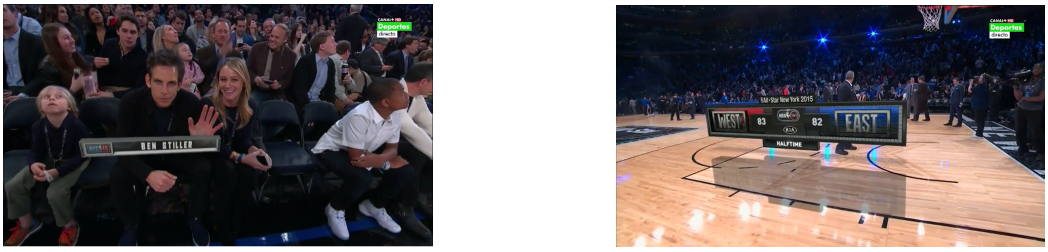
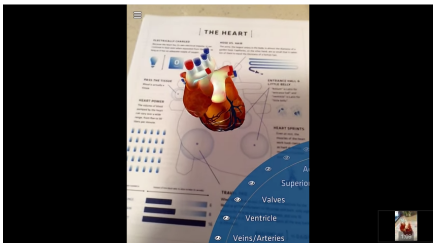
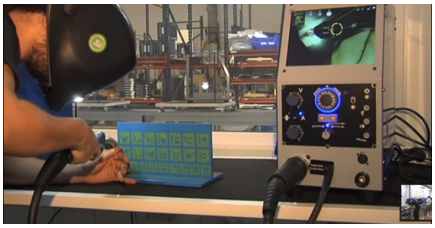


Figura 1.13: Ejemplos de AR en la retrasmisión del All Star 2015

cación. Desde ayudar a los más pequeños a aprender de una forma más divertida e interactiva hasta facilitar la formación práctica en determinadas tareas que supongan un alto coste de recursos. Como ejemplo que ha revolucionado un curso de soldadura de Formación Profesional, está la aplicación Soldamatic (Figura 1.14b), desarrollada por la empresa española Seabery, y que además de ayudar a los profesores a preparar la teoría que imparten, permite realizar ejercicios prácticos simulando piezas a soldar mediante realidad aumentada. Otro ejemplo es Anatomy4D que permite aprender anatomía con ejemplos como el corazón de la Figura 1.14a que late mientras observamos sus partes usando la app.



(a) Fichas educativas de Anatomy4D



(b) Práctica de soldadura con Soldamatic

Figura 1.14: Ejemplos de realidad aumentada en educación

Queda por nombrar el que posiblemente sea el campo en el que más se investiga y uno de los más importantes, el campo de la medicina. En el apartado de visión artificial ya vimos que existían muchas aplicaciones en este sector, pero las más recientes y punteras están enfocadas a la realidad aumentada, tanto como ayuda en el aprendizaje de procedimientos quirúrgicos como sobre todo en la asistencia en operaciones, en lo que se conoce como cirugía guiada por imágenes. En la Figura 1.15 podemos observar lo útil que puede ser una aplicación de realidad aumentada a la hora de operar con técnicas poco intrusivas como

puede ser una laparoscopia.

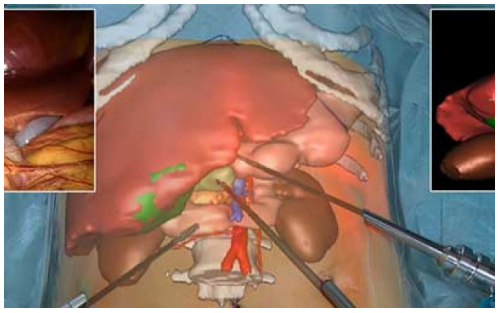


Figura 1.15: Realidad aumentada en una intervención por laparoscopia

1.4. Autolocalización Visual y Realidad Aumentada en el Grupo de Robótica de la URJC

Viendo las posibilidades que ofrece la autolocalización visual, dentro del Grupo de Robótica de la Universidad Rey Juan Carlos se han llevado a cabo diversas investigaciones que sirven como antecedentes al presente Proyecto Fin de Carrera.

En 2005 Alberto López [5] enfoca su PFC a resolver el problema de la autolocalización de robots móviles, empleando dos algoritmos diferentes, uno basado en mallas de probabilidad y otro con filtro de partículas.En 2010 Luis Miguel López [6] diseña e implementa un algoritmo de autolocalización para un sistema móvil con una única cámara ($monoSLAM$) basándose en los trabajos de Davison.

(a) Juego de realidad aumentada

(b) AutoNav

Figura 1.16: Autolocalización visual en la URJC

Más recientemente, en 2014, Alejandro Hernández [3] trabaja en la implementación de varios algoritmos de autolocalización visual, siendo el más importante uno basado en PTAM. Para probar este último desarrolla el juego de realidad aumentada que podemos ver en la Figura 1.16a. Recientemente José Manuel Villarán [9] ha desarrollado un algoritmo de autolocalización basado en balizas visuales que se ha utilizado en el proyecto AutoRob [7] de guiado de robots de carga industriales. Podemos ver uno de estos robots guiándose con marcadores en la Figura 1.16b.

Motivado por el juego de Alejandro Hernández, este Proyecto Fin de Carrera busca dar un paso más en el ámbito de la realidad aumentada al añadir efectos de física a los elementos simulados e incluso hacer que estos interactúen con objetos reales. Además, persigue utilizar bibliotecas de visualización más avanzadas y abstractas que *OpenGL*. Para resolver la autolocalización de la cámara se ha decidido implementar un algoritmo propio basado en balizas visuales.

A continuación, en el capítulo 2, se exponen los objetivos marcados y la metodología empleada. En el capítulo 3 se describen los elementos *hardware* y *software* en los que se apoya este trabajo. Después se hablará del desarrollo de la solución y de los experimentos que permiten validarla, para ya en el capítulo 6 terminar con las conclusiones obtenidas de los resultados y comentando las futuras líneas de trabajo que pueden surgir de este proyecto.

[7] http://www.jderobot.org/AutoRob

Capítulo 2

Objetivos

A continuación se exponen los objetivos de este proyecto, los requisitos que debe cumplir el resultado final y la metodología empleada para su desarrollo.

2.1. Descripción del problema

El objetivo principal de este proyecto es el desarrollo de una aplicación de realidad aumentada realista que integre diferentes tecnologías. Más concretamente se busca diseñar e implementar una aplicación que muestre elementos ficticios interactuando, no sólo entre ellos, si no también con elementos reales. Además se validará la solución tanto en un entorno simulado como en uno real.

Este objetivo principal se ha articulado en los siguientes tres subobjetivos:

- Desarrollo de un algoritmo de autolocalización visual que deberá ser capaz de estimar la posición y orientación 3D de la cámara en cada momento, a partir de las imágenes RGB que obtiene la misma y apoyándose en el reconocimiento de unas balizas de las que se conoce su ubicación dentro del escenario. Esta posición estimada servirá para colocar la cámara que observa el mundo virtual en el lugar que le corresponde.

- Creación de una escena virtual mediante el uso de librerías gráficas y de físicas que contenga objetos que se mueven, que pueden estar animados y que se ven afectados por propiedades físicas (como choques por colisiones o caídas aplicando la fuerza de la gravedad), para que la realidad aumentada sea más atractiva.

- Integración de los puntos anteriores para desarrollar una aplicación de realidad aumentada que podrá ser observada desde cualquier posición por la que se desplace la cámara, mientras se tenga alguna baliza como referencia. Se validará experimentalmente su comportamiento.

2.2. Requisitos

Además de los objetivos marcados, la solución desarrollada en este proyecto debe cumplir con los siguientes requisitos:

- Se hará uso de la plataforma JdeRobot 5.2 y se desarrollarán componentes modulares programados en el lenguaje C++.

- El componente se podrá ejecutar sobre Ubuntu 12.04 tanto en arquitecturas de 32 como de 64 bits.

- Se debe cumplir un rendimiento que permita una ejecución en tiempo real fluida y sin decalajes entre distintos fotogramas, para que el efecto de realidad aumentada sea lo más realista posible.

- La localización será lo suficientemente robusta para permitir cualquier movimiento de la cámara mientras éste sea suave.

- Se debe permitir cargar la configuración de la escena (posición de las balizas, calibración de la cámara, posición inicial de los elementos ficticios, etc) de forma sencilla y dinámica, para no depender de un único escenario.

2.3. Metodología

El modelo de ciclo de vida utilizado ha sido el espiral basado en prototipos. Este método de trabajo permite desarrollar el proyecto de forma incremental, aumentando su complejidad progresivamente y haciendo posible la generación de prototipos funcionales.

Este tipo de modelo de ciclo de vida permite obtener productos parciales que pueden ser evaluados, total o parcialmente, facilitando la adaptación a los cambios requeridos, algo que sucede habitualmente en los proyecto de investigación.

El modelo en espiral se realiza por ciclos donde cada uno representa una fase del proyecto. Dentro de cada ciclo del modelo en espiral se diferencian cuatro partes principales que pueden verse en la Figura 2.1, donde cada una de las mismas tiene un fin distinto:

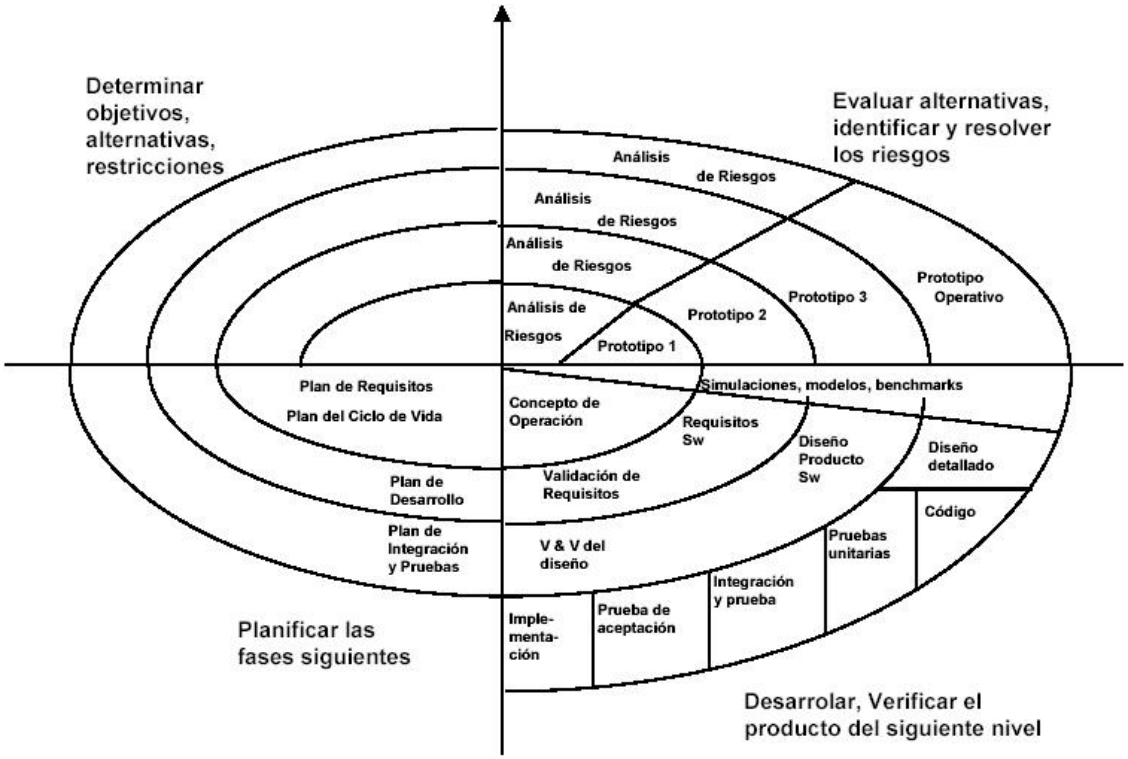


Figura 2.1: Modelo en espiral (Barry Boehm, 1986)

- Determinar objetivos: Se establecen las necesidades que debe cumplir el sistema en

cada iteración teniendo en cuenta los objetivos finales. Típicamente según avancen las iteraciones aumentarán el coste del ciclo y su complejidad.

- Evaluar alternativas: Determina las diferentes formas de alcanzar los objetivos establecidos en la fase anterior, utilizando distintos puntos de vista, como el rendimiento que puede tener en espacio y tiempo, las formas de gestionar el sistema, etc. Además, se consideran explícitamente los riesgos, intentando mitigarlos lo máximo posible.

- Desarrollar y verificar: Desarrollar el producto siguiendo la mejor alternativa para poder alcanzar los objetivos del ciclo. Una vez diseñado e implementado el producto se realizan las pruebas necesarias para comprobar su funcionamiento.

- Planificar: Teniendo en cuenta el funcionamiento conseguido por medio de las pruebas realizadas, se planifica la siguiente iteración revisando los posibles errores cometidos a lo largo del ciclo y comenzando un nuevo ciclo de la espiral.

Como parte de la metodología se han realizado reuniones semanales con el tutor en las que se comentaban los problemas encontrados, se revisaban objetivos y se estudiaban las distintas alternativas de avance en el desarrollo del proyecto. El detalle del progreso del trabajo se puede consultar en el cuaderno de bitácora[1] que ha ido recogiendo los avances más significativos del proyecto así como alguno de los problemas enfrentados, ilustrando los mismos con imágenes y vídeos.

2.4. Plan de trabajo

A continuación se describen las diferentes etapas que componen este proyecto y que se corresponden con ciclos del modelo en espiral:

[1] http://JdeRobot.org/Dazuara-pfc

- **Familiarización con el entorno de JdeRobot:** El objetivo de esta fase es aprender a utilizar los diferentes componentes y *plugins* de la plataforma JdeRobot, así como obtener una visión general de las tecnologías con las que se trabaja en el departamento de Robótica de la URJC.

- **Familiarización con el simulador Gazebo:** Aprendizaje en el uso del simulador Gazebo y en la creación de escenarios virtuales para el mismo. Además se estudiará cómo desarrollar *plugins* para este simulador.

- **Familiarización con Ogre y Bullet:** El objetivo de esta fase es estudiar las bases para el desarrollo de aplicaciones gráficas con Ogre y para simular físicas con Bullet. Se analiza cómo definir escenas, crear actores y controlar como estos interactuan. En esta primera aproximación a Ogre y Bullet se trabajará con ellos por separado.

- **Desarrollo de un algoritmo de autolocalización visual con balizas:** El objetivo de esta fase es el desarrollo de un algoritmo que permita estimar la posición de una cámara procesando sus imágenes. El algoritmo estará basado en una autolocalización visual con balizas, empleando marcadores de AprilTags.

- **Desarrollo de una aplicación de realidad aumentada:** El objetivo es diseñar e implementar un componente JdeRobot que muestre una realidad aumentada empleando el algoritmo de autolocalización visual con balizas desarrollado. Además esta realidad aumentada deberá combinar efectos gráficos y físicos empleando los conocimientos aprendidos sobre Ogre y Bullet.

Capítulo 3

Infraestructura

En este capítulo se verán las diferentes soluciones *hardware* y *software* en las que se ha apoyado el desarrollo de este proyecto.

3.1. Hardware

Tanto para el desarrollo de este de este proyecto, como para la ejecución y pruebas de la aplicación se ha usado un portátil Asus F550C con procesador *Intel(R) Core(TM) i7 3537U @ 2.00 Ghz*, con sistema operativo *Ubuntu 12.04 LTS* una de las distribuciones de *GNU/Linux* más extendidas.

(a) Portatil Asus F550C

(b) Cámara Logitech pro 9000

Figura 3.1: Hardware empleado

La cámara empleada ha sido una *Logitech WebCam Pro 9000* conectada al ordenador por puerto USB. Su rendimiento típico es de 30fps de tasa de refresco y resolución de 640x480px.

3.2. JdeRobot

JdeRobot [1] es una colección de librerías y componentes *open source* que facilitan el desarrollo de aplicaciones de robótica y visión artificial. Sus funcionalidades permiten abstraerse de la comunicación con el *hardware* para dedicar todo el esfuerzo al diseño y desarrollo de aplicaciones. Ofrece tanto una serie de componentes que conectan con sensores y actuadores reales, como una serie de *plugins* para Gazebo. Este último es uno de los simuladores de escenarios más potentes y fiables que se puede encontrar y JdeRobot ofrece una excelente integración con él.

Esta comunicación, ya sea con entorno real o simulado, se soporta en el *middleware* ICE que permite transmitir información por protocolo TCP/IP y siendo transparente para el desarrollador tanto la arquitectura como el lenguaje de desarrollo utilizado en el otro extremo de la conexión.

JdeRobot, además, incluye las librerías necesarias para trabajar con los datos recibidos, permitiendo adaptar su formato y ofreciendo operaciones para procesarlos. En este trabajo se usan algunas de estas librerías para procesar la imagen y realizar operaciones matemáticas. La versión empleada es JdeRobot 5.2.0.

El resultado de este proyecto es una aplicación JdeRobot formada por componentes que interoperan. Se emplea el componente *cameraServer* para obtener el flujo de imágenes y se crea el componente *demoAR* que implementa la aplicación de realidad aumentada. También se desarrolló un plugin de Gazebo para una cámara teledirigida y un componente

[1] http://www.jderobot.org/

para controlar dicha cámara, siguiendo la filosofía de JdeRobot. Veamos en detalle algunos de estos elementos JdeRobot.

3.2.1. Componente cameraServer

Este componente es un servidor de imágenes basado en OpenCV y en ICE. Es capaz de obtener fotogramas de un archivo de vídeo (aceptando una gran variedad de formatos) o leer de un dispositivo conectado, empleando un captador de vídeo de OpenCV. El componente adapta la imagen al formato, tamaño y *framerate* configurados y la ofrecerá a través de una conexión ICE. Como ya se ha comentado, en el proyecto se usa para servir el flujo de imágenes que alimenta al componente demoAR cuando se trabaja con una cámara real.

3.2.2. Plugin flyingCamera

Como extensión al plugin de cámara para Gazebo que existe en JdeRobot se desarrolla otro plugin que permite actuar sobre la posición y orientación de un objeto en Gazebo, para obtener así una cámara teleoperada simulada. Este nuevo plugin permite tanto leer la posición 3D de un objeto como variarla y en este trabajo se empleará en la fase de experimentos para poder validar el algoritmo de autolocalización, obteniendo de esta cámara tanto la imagen que alimenta al algoritmo como su posición verdadera en el mundo Gazebo para comparar el resultado de la estimación.

3.2.3. Componente moveCamera

También se colabora en el desarrollo de este componente que permite manejar una cámara teledirigida mediante una interfaz gráfica que muestra la imagen de la cámara y unos controles para variar su posición y orientación, como podemos ver en la Figura 3.2. Mediante un interfaz ICE se comunica con el plugin para actualizar la posición y para

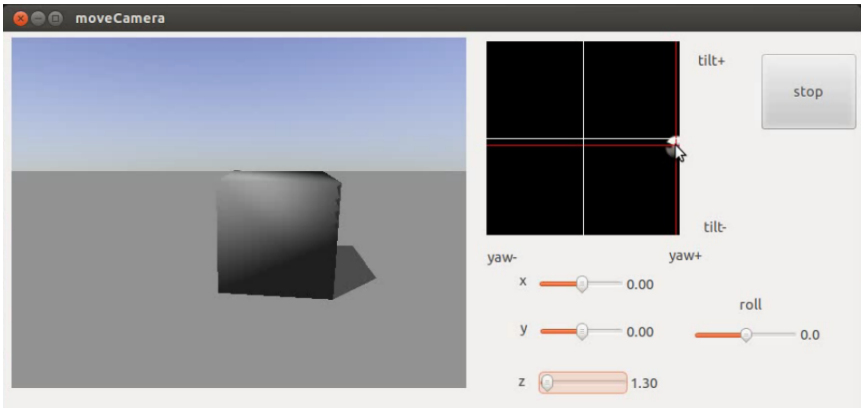


Figura 3.2: Componente moveCamera

obtener la imagen, podemos ver esta interacción en la Figura 3.3. Al igual que el plugin, se utiliza en varios experimentos para mover la cámara simulada en Gazebo.

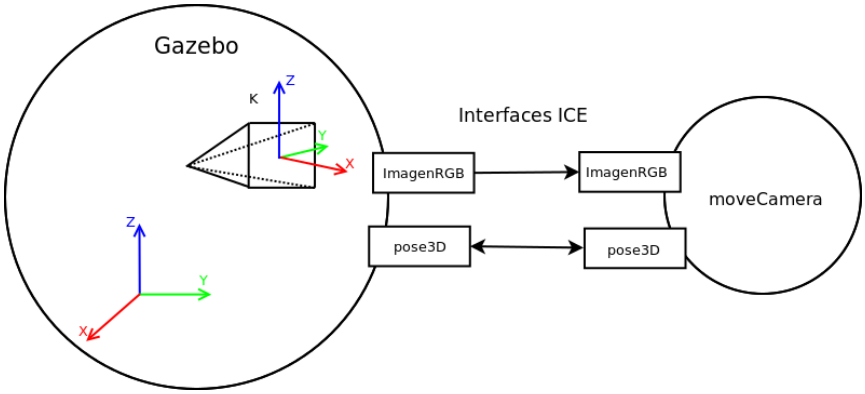


Figura 3.3: Interfaces ICE

3.2.4. Librería ParallelICE

Uno de los problemas que presenta el uso de ICE es la latencia que se produce cuando se pide actualizar un dato y se espera la respuesta del servidor. Además, el problema se acentúa cuando existen varias fuentes de datos. Para solucionar este inconveniente la plataforma JdeRobot ofrece una librería, llamada ParallelICE, que crea un hilo dedicado para consultar continuamente (a la frecuencia configurada) los datos del servidor, guardando una copia de los mismos en memoria local. Cuando el programa principal solicita actualizar sus datos, éstos se obtienen de memoria local en lugar de tener que hacer la consulta al

servidor, mejorando así el rendimiento. En la Figura 3.4 podemos ver un diagrama con el funcionamiento.

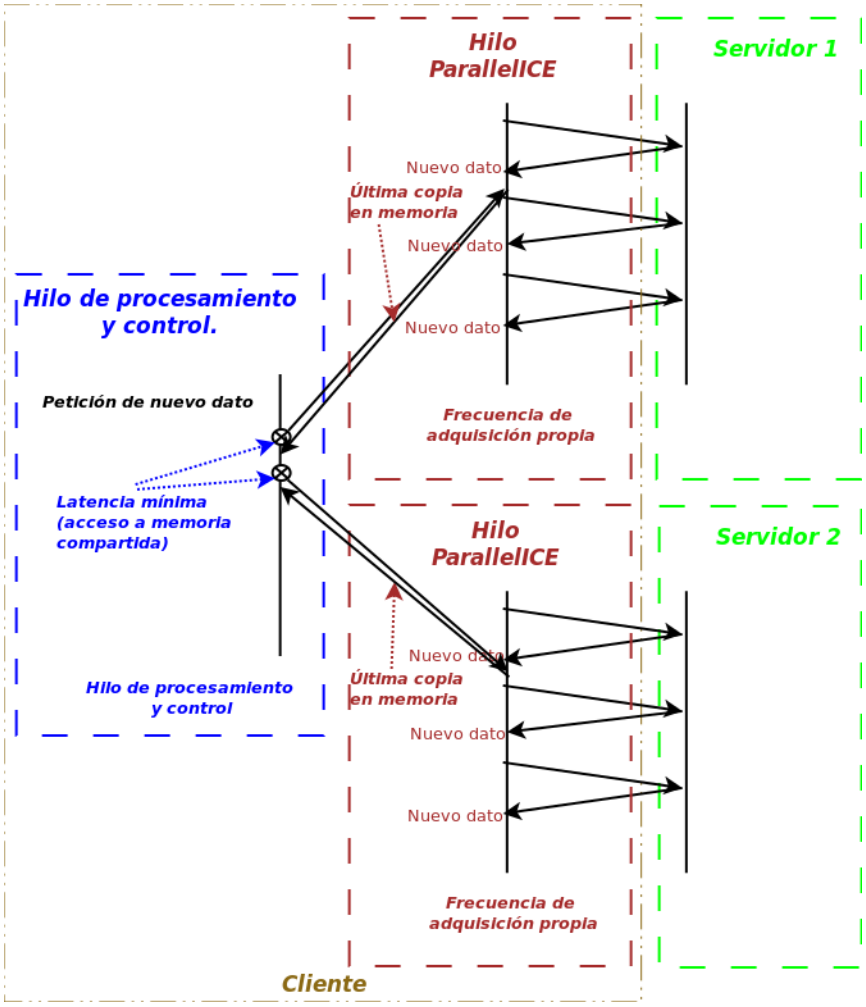


Figura 3.4: Librería ParallelICE

3.3. ICE

ICE [2] (The Internet Communications Engine) es una biblioteca que permite la comunicación entre distintos componentes de aplicaciones distribuidas de una forma sencilla y orientada a objetos. Emplearla permite abstraerse de tareas a bajo nivel como abrir conexiones, serializar y deserializar los datos o reintentar conexiones fallidas. Además, cada

[2]https://zeroc.com/ice.html

componente distribuido puede estar desarrollado en diferentes lenguajes de programación y ejecutarse sobre diferentes sistemas operativos, pero la comunicación será tan simple como empleando sistemas idénticos.

En el proyecto se usa la versión 3.4 para la comunicación entre los diferentes componentes de la aplicación.

3.4. Gazebo

Trabajar con un simulador en desarrollos de robótica es fundamental para tener un periodo de aprendizaje y de inicio menos costosos. Permite también no tener dependencia de la disponibilidad de los componentes *hardware* a emplear en la versión final. En este proyecto se utiliza el simulador Gazebo.

El desarrollo de Gazebo [3] comenzó en 2002 con la meta de crear un simulador de alta fidelidad para escenarios exteriores. En Julio de 2013 Gazebo se emplea como entorno simulado para el torneo *DARPA Robotics Challenge* fomentado por el departamento de defensa norteamericano. Ganando popularidad y financiación se posiciona a la cabeza de los mejores simuladores de robótica.

En este trabajo se emplea la versión 1.8.1 para obtener el flujo de imágenes de una cámara simulada, dentro de un entorno monitorizado. Se facilita así el desarrollo y la depuración de todas las fases del proyecto, ya que permite conocer la localización exacta (posición y orientación) de la cámara. Este conocimiento se puede usar tanto para no depender de la autolocalización visual en algún momento del desarrollo, como para las validaciones en la fase de experimentos. Como ya se ha comentado, la implementación del plugin de cámara teledirigida que se usa es propia, se puede ver funcionando junto al componente que la opera en la Figura 3.5.

[3] http://gazebosim.org/

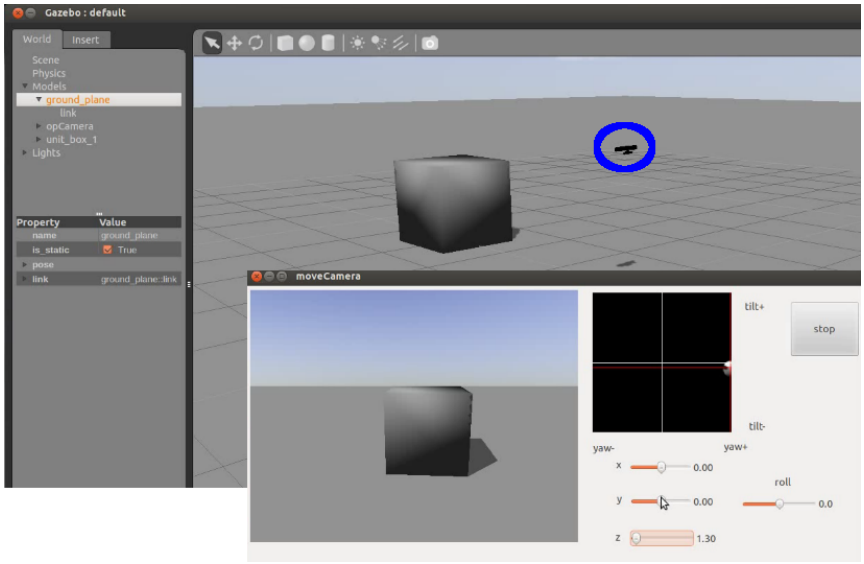


Figura 3.5: Gazebo con el plugin de cámara teledirigida

3.5. OpenGL

OpenGL [4] es una especificación que define una serie de funciones para el *renderizado* de imagen, es decir para generar una imagen sintética a partir de un modelo 3D que generalmente cuenta con información de materiales, colores e iluminación.

Esta especificación es independiente tanto del lenguaje de programación como de la plataforma y existen implementaciones de múltiples fabricantes de *hardware*. Es utilizado en diferentes industrias siendo quizás la de los videojuegos la más conocida.

En este proyecto se emplea en las primeras fases del mismo para obtener conocimientos sobre la representación de gráficos 3D a bajo nivel.

3.6. Ogre

Ogre [5] es un motor de representación de escenas 3D. Se encarga de gestionar tareas de bajo nivel, como el uso de funciones de OpenGL y Direct3D, para que el desarrollo

[4] https://www.opengl.org/
[5] http://www.ogre3d.org/

(a) Juego desarrollado con OpenGL

(b) Juego desarrollado con Ogre 3D

Figura 3.6: Librerías de renderizado

de aplicaciones de gráficos 3D sea más sencillo e intuitivo. Es muy útil para el desarrollo de aplicaciones con escenas 3D complejas, con muchos elementos móviles y efectos de sombras y texturas realistas. Un claro ejemplo de este tipo de escenas pueden ser las de un videojuego.

Permite que se definan una serie de objetos en una escena, encargándose de refrescar la imagen con los elementos que aparezcan frente a la *cámara* que graba la escena y abstrayendo tareas complejas como hacer que anden o muevan partes de su cuerpo.

En la aplicación que aquí se expone se emplea la versión 1.8.1 de la librería para el *renderizado* del mundo virtual desde la posición de la cámara, para hacer que este se superponga al flujo de imágenes real.

3.7. Bullet

Bullet [6] es un motor de físicas. Al igual que Ogre permite trabajar con objetos dentro de una escena ficticia y se encarga de aplicar a cada uno las fuerzas que les afecten, teniendo en cuenta posibles colisiones entre ellos. Este paralelismo entre ambas bibliotecas hace que su integración sea natural y sencilla.

[6] http://bulletphysics.org/

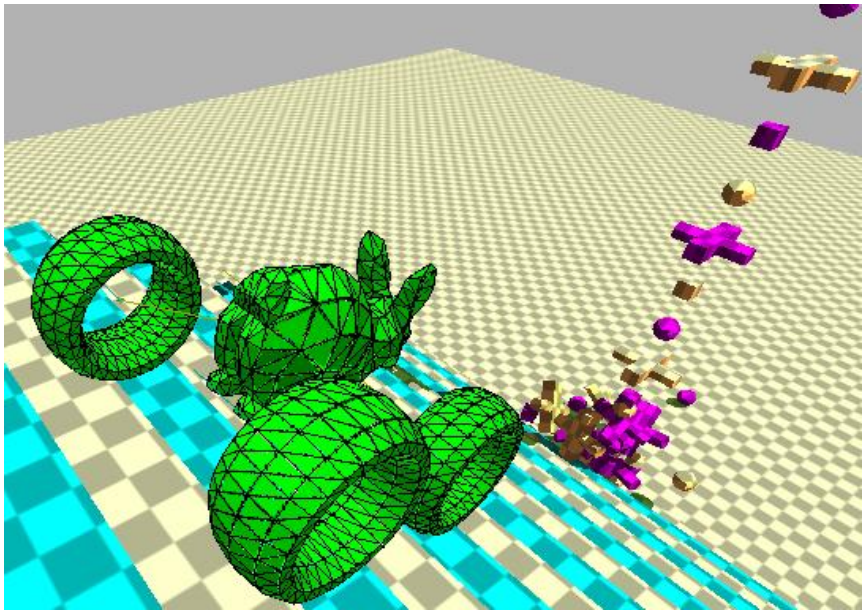


Figura 3.7: Simulación de físicas con Bullet

Es interesante destacar que de cara al motor de físicas existen varios tipos de objetos:

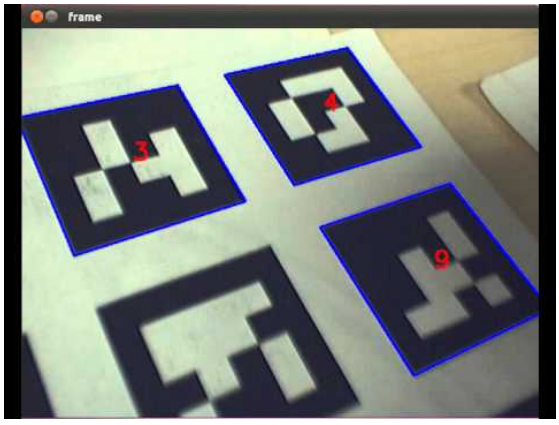
- Inamovibles, no se ven afectados por las fuerzas físicas, ni tampoco se mueven, pero se tienen en cuenta en las colisiones. Por ejemplo, el suelo.
- Afectados por las fuerzas físicas. Su posición en cada momento la indica el motor de físicas.
- Cinemáticos, su movimiento no depende de fuerzas. Son objetos que cambian su posición por orden de la aplicación y no del motor de físicas, pero que pueden colisionar. El motor recibe las diferentes posiciones de su trayectoria en cada instante de tiempo, para saber cómo interactua con el resto de elementos del ecosistema.
- Mezcla de los dos últimos, en el sentido de que se manejan por la aplicación, pero sí se ven afectados por las leyes físicas.

En este trabajo se usa la versión 2.82 para gestionar las reacciones físicas de los elementos ficticios. La incorporación de simulaciones físicas a una aplicación de realidad aumentada hace que ésta sea más realista.

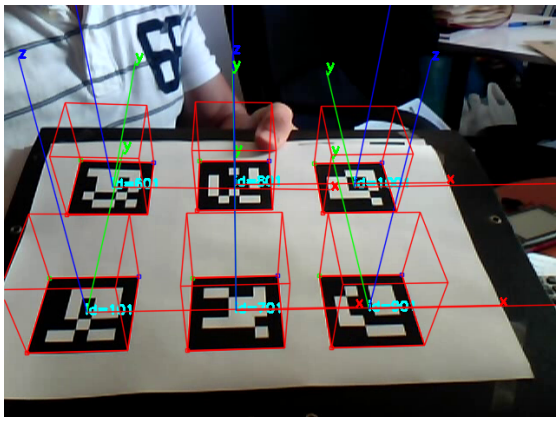
3.8. AprilTags y ArUco

Las AprilTags [7] son códigos de barras bidimensionales, similares a los QR, diseñados para almacenar menos información que estos pero de forma más robusta. Estas características hacen que las AprilTags sean una buena elección como balizas artificiales y puedan ser de gran utilidad en aplicaciones de autolocalización visual. Fueron ideadas por Edwin Olson [7] para su uso en aplicaciones de robótica y más adelante se desarrolló una librería para C++ que permite detectar AprilTags y obtener su identificador único y su posición en la imagen.

Existen diferentes *familias* de marcadores, en función del número de bits que contienen las imágenes. A mayor número de bits (el máximo es de 36) mayor robustez, es decir son más difíciles de confundir con falsos positivos.



(a) Detección de marcadores con AprilTags



(b) Efecto de realidad aumentada con ArUco

ArUco [8], desarrollada por el grupo de investigación de *Aplicaciones de la Visión Artificial* de la Universidad de Córdoba, es una librería basada en OpenCV que ofrece la funcionalidad mínima necesaria para crear aplicaciones de realidad aumentada. Esta funcionalidad básica consiste en la detección de marcadores en una imagen y la estimación de la posición 3D de dichos marcadores respecto de la cámara conociendo su calibración. Una

[7] http://april.eecs.umich.edu/wiki/index.php/AprilTags
[8] http://www.uco.es/investiga/grupos/ava/node/26

vez que se tiene la posición y orientación de una baliza, se puede usar alguna biblioteca de renderizado 3D (*como OpenGl*) para *pintar* objetos virtuales sobre esta y tener así una aplicación sencilla de realidad aumentada. El problema es que al no saber la posición exacta de cada elemento del escenario (cámara y objetos) dentro del mapa, estas aplicaciones son muy limitadas.

En comparación con AprilTags, ArUco ofrece una detección de marcadores menos robusta, sus marcadores no contienen tantos bits de información, pero una estimación de su posición y orientación más fiable.

En la aplicación desarrollada como demostración final del trabajo se emplean marcadores de AprilTags (familia 36h11) como balizas. Dentro del módulo de autolocalización se llama a funciones de AprilTags en su versión 1.0.0, para la detección de los marcadores y se emplea la versión 1.2.5 de ArUco para obtener la posición 3D de cada detección respecto a la cámara. El resultado que se obtiene con ArUco es el punto de partida para calcular la posición 3D de la cámara (respecto del punto de referencia del mundo) que se empleará en el módulo gráfico para *renderizar* la escena desde el punto adecuado y con la orientación adecuada.

3.9. OpenCV

OpenCV [9] (Open source Computer Vision) es una librería de visión artificial que bajo licencia BSD ofrece más de 2500 algoritmos para detectar y reconocer caras, identificar objetos, clasificar acciones humanas y un largo etcétera. Es multi-lenguaje, multi-plataforma y fue diseñada para que fuera eficiente en aplicaciones de tiempo real.

En este trabajo se emplean estructuras de datos de esta librería (en su versión 2.4.10), para almacenar y adaptar las imágenes durante su procesado, también para el calibrado de la cámara y para la conversión de vectores de rotación a matrices en la autolocalización.

[9] http://opencv.org/

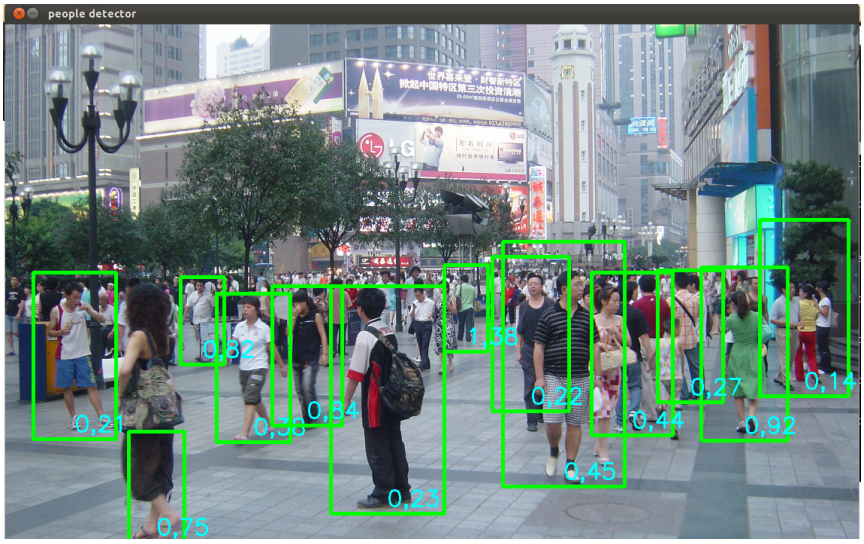


Figura 3.8: Detección de personas con OpenCV

Aún más importante es su uso dentro de la librería ArUco, ya que la función que obtiene la posición 3D de un marcador respecto a la cámara se basa en invocar la función *solvePnP* de OpenCV para resolver la ecuación del modelo de cámara *pinhole* que modela la proyección de un objeto 3D sobre la imagen que captura una cámara. Veremos el detalle sobre uso en el capítulo de desarrollo 4.2.2.

3.10. Eigen

Eigen [10] es una biblioteca para cálculos de álgebra lineal que permite trabajar con matrices, vectores y ofrece métodos de resolución y reducción de sistemas lineales.

En el proyecto se aplica, en su versión 3.0.5, durante los cálculos matriciales necesarios para hallar una estimación de la posición 3D de la cámara, permitiendo aplicar las transformaciones de rotación y traslación necesarias.

[10] http://eigen.tuxfamily.org/

Capítulo 4

Desarrollo

A continuación se detalla el desarrollo del componente JdeRobot que es resultado final de este proyecto. Esta aplicación, llamada demoAR, permite visualizar un mundo virtual sobrepuesto a uno real que es capturado por una cámara. Además, el mundo se podrá ver desde cualquier posición mientras que el módulo de autolocalización pueda estimar la ubicación de la cámara a partir de las balizas visuales que hay en las escena. Como ejemplo de demostración se creará un mundo ficticio conteniendo un robot, una mesa y una pelota que se verán afectados por efectos físicos, además se simulará un cilindro que ha de coincidir con un objeto real para obtener una escena de realidad aumentada que muestre un robot golpeando una pelota que a su vez rebota en el objeto real (como puede ser una taza) para acabar cayendo de la mesa al alcanzar el borde.

Primero se dará una visión general del componente mencionando los diferentes módulos que lo forman, para después entrar a ver el detalle de cómo funciona cada uno de ellos.

4.1. Diseño global

La aplicación desarrollada tiene como único dato de entrada un flujo continuo de imágenes de cámara. Para obtener estas imágenes se emplea ICE, por lo que será necesario configurar la ip y puerto del componente ICE que sirve la información en un archivo de

configuración. En cuanto a la fuente de información, la versión final beberá del componente *cameraServer* el cual lee de una cámara conectada por USB, pero en el apartado de experimentos veremos cómo las imágenes pueden venir igualmente de una cámara simulada en Gazebo.

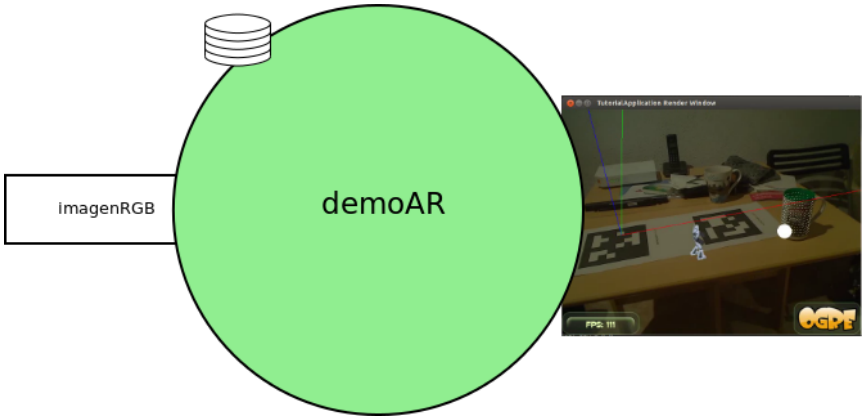


Figura 4.1: Diagrama de entradas y salidas

El componente, cuyo diseño podemos ver en la Figura 4.2, se puede dividir en tres partes que intercambian información a través de un bloque de memoria compartida, y que ejecutan al mismo tiempo en diferentes hilos de procesado.

- Una de estas partes, que podemos denominar *módulo de autolocalización*, se encarga de actualizar la imagen guardando el último fotograma en la memoria compartida y de llamar a la función que estima la posición 3D de la cámara, guardándola también en el bloque compartido.

- También tenemos un módulo que se encarga de la simulación de los efectos de física sobre los distintos objetos de la escena, para ello se pide al motor de físicas que avance la simulación en cada iteración aplicando las fuerzas y colisiones que se produzcan.

- La última parte del componente se encarga de la interfaz gráfica y para ello crea una aplicación de Ogre que *renderiza* dentro de una ventana una escena ficticia con la imagen real proyectada de fondo aplicando una textura dinámica. En cada ciclo

de *renderizado* se sitúa la cámara en la última posición 3D estimada, se actualiza la posición de los objetos virtuales y se refresca la textura del fondo.

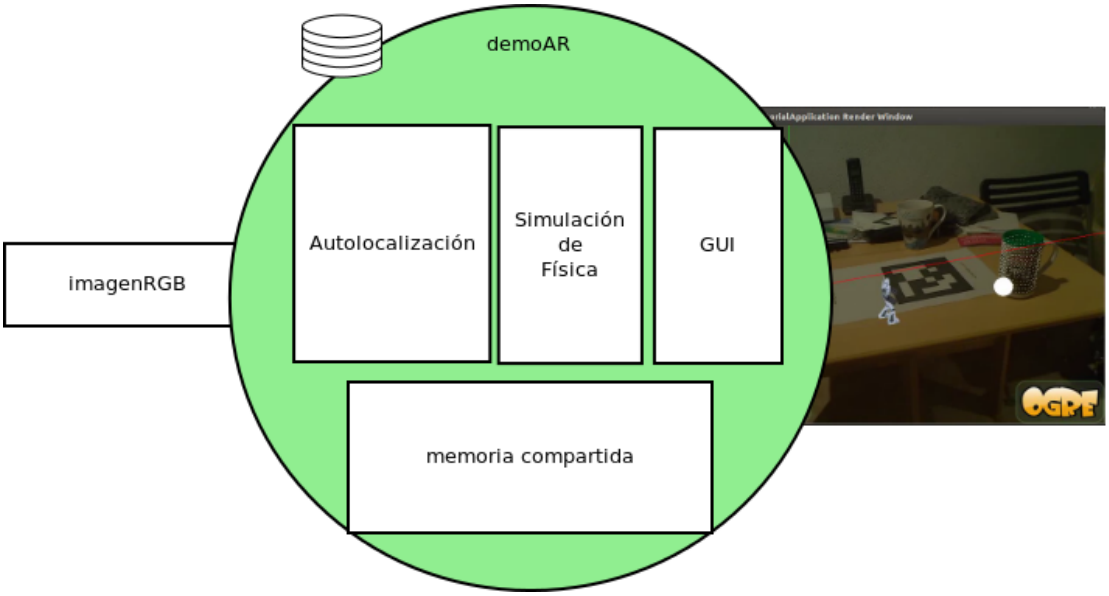


Figura 4.2: Diagrama de bloques

Como ya se ha comentado antes, el componente dispone de un fichero de configuración que contiene los parámetros de la conexión con ICE y además las características de la escena: posición inicial de robot y pelota, ubicación de la taza, altura de la mesa, y lo más importante, la posición y orientación de cada una de las balizas identificándolas por su número. Además, comentar que se emplea la librería parallellICE para tener un hilo que refresque constantemente la imagen llamando a ICE para que cuando la aplicación solicite actualizar la imagen esta se obtenga de memoria local directamente evitando así afectar al rendimiento.

4.2. Autolocalización visual

Este módulo se encarga de procesar las imágenes para calcular una estimación de la posición 3D de la cámara que, a través de la memoria compartida, será usada por el módulo gráfico para situar la cámara virtual desde la que se renderiza la escena ficticia.

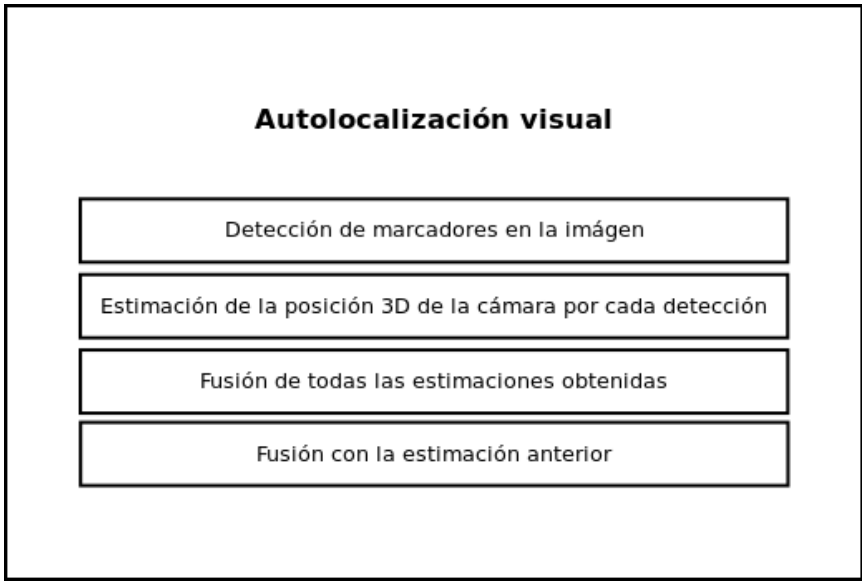


Figura 4.3: Diagrama de bloques

La autolocalización esta dividida en cuatro fases, como podemos ver en el diagrama de bloques de la figura 4.3. Primero se trabaja la detección de balizas en la imagen usando funciones de la librería AprilTags que procesan la imagen para buscar marcadores e identificarlos. Después, por cada detección obtenida se estima la ubicación de la cámara (posición y orientación) empleando funciones de ArUco que permiten calcular los parámetros extrínsecos de la cámara en forma de matriz de transformación. Por último, se toman todas las estimaciones (una por cada baliza detectada) y se fusionan en una única mediante una media ponderara por la calidad de cada detección (lo que llamaremos peso). Además, esta estimación resultado de una fusión local también se fusiona con la posición 3D calculada en la iteración anterior para obtener una mayor robustez.

4.2.1. Detección de balizas con AprilTags

Para la detección de balizas con AprilTags es necesario crear una instancia de *TagDetector* configurando la familia de marcadores que se usa, en nuestro caso, la familia 36h11. Esta instancia de un detector de AprilTags necesita recibir una imagen en escala de grises para procesarla, por tanto, para adaptar la imagen emplearemos una función de OpenCV.

Una vez tenemos la imagen lista, llamamos al método *extractTags* pasando la imagen en escala de gises y guardamos el resultado en un vector de *TagDetection*.

```
// detect April tags (requires a gray scale image)
cv::cvtColor(frame, frame_gray, CV_BGR2GRAY);
detections = m_tagDetector->extractTags(frame_gray);
```

Entrando al detalle de la función *extractTags* vemos que primero se aplica un filtro Gaussianno paso bajo, para facilitar la detección de contornos de los marcadores, después se calcula el gradiente local de cada pixel de la imagen para aplicar un algoritmo voraz que agrupe píxeles con gradiente similar formando segmentos. Por cada cuatro segmentos que formen una forma cerrada se considera que se ha detectado un cuadrado (que puede o no ser un marcador). Por último se intenta decodificar el contenido de cada cuadrado analizando si los bits que se leen tienen sentido y en caso afirmativo se considera que se ha dado con la detección de un marcador. En la imagen 4.4 podemos ver el resultado positivo de la detección de dos marcadores en la escena en la que se realiza uno de los experimentos del componente demoAR.

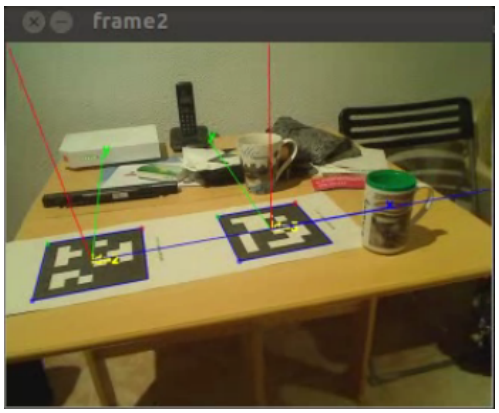


Figura 4.4: Detección de marcadores con AprilTags

Las detecciones que nos devuelve ApilTags contienen el valor de los cuatro píxeles del la imagen que se corresponden con las esquinas, el valor de su identificador y los píxeles de su centro, entre otros datos. En nuestro caso, nos quedaremos con los píxeles de las cuatro

esquinas para usarlos en las llamadas a ArUco, y con el identificador para recuperar la información que conocemos sobre la baliza (en especial su posición 3D en el mapa).

4.2.2. Estimación de la posición 3D de la cámara

Por cada detección obtenida podemos calcular una estimación de la posición 3D de la cámara, si tenemos en cuenta que se cumple el modelo de cámara *pinhole*, el cual podemos ver en la Figura 4.5, en una versión simplificada (con el plano imagen frente al foco de la cámara y no detrás como realmente estaría en una cámara real).

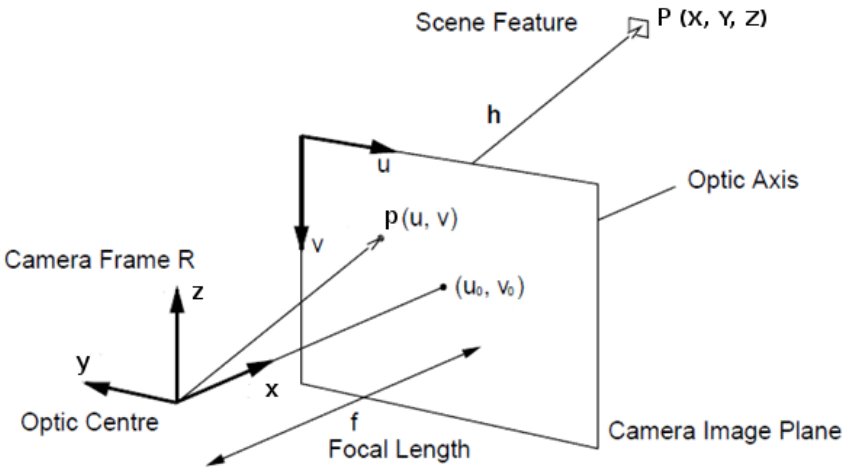


Figura 4.5: Modelo de cámara *pinhole*

Este modelo representa la proyección de un punto 3D sobre el plano imagen de una cámara mediante la ecuación 4.1, donde P es el punto 3D, p su proyección sobre la imagen y las matrices K y RT los parámetros intrínsecos y extrínsecos de la cámara respectivamente. La matriz K esta formada por la distancia focal y el centro óptico de la cámara, mientras que la matriz RT es la transformación (rotación y traslación) entre el origen de referencia y la posición de la cámara.

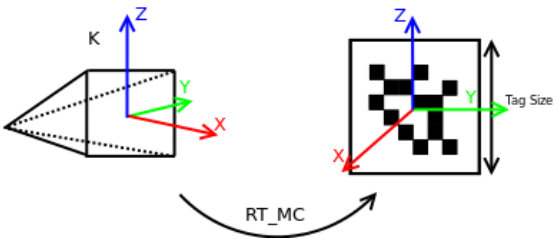
$$p = K \cdot RT \cdot P \tag{4.1}$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{4.2}$$

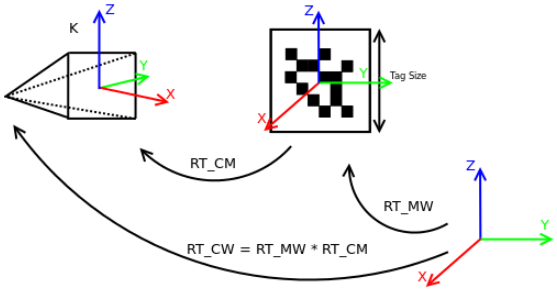
Resolviendo esta ecuación se pueden obtener los parámetros extrínsecos de la cámara siempre que se conozcan los intrísecos, los cuales se pueden obtener calibrando la cámara. En nuestro caso calibramos la cámara con una aplicación de la librería OpenCV que genera un fichero XML que contine la matriz K y la matriz de distorsión de la cámara. Para resolver la ecuación creamos una instancia de un *Marker* de ArUco pasando las esquinas detectadas. Esta clase de ArUco, además de las cuatro esquinas y del identificador del marcador también tiene como atributos los vectores de su rotación y traslación respecto de la cámara. Al crear la instancia estos vectores están vacíos por defecto, pero podemos llamar al método *calculateExtrinsics* para que ArUco calcule su valor. Como parámetros de entrada se necesitan el tamaño del marcador, los parámetros de la cámara (matriz de intrínsecos y de distorsión) y un *booleano* que indica si el *upvector* con el que estamos trabajando es el eje Y o Z.

```
std::vector<cv::Point2f> realCorners;

// Create the Marker instance and calculate extrinsics
TheMarkers[i] = aruco::Marker(realCorners, detections[i].id);
TheMarkers[i].calculateExtrinsics(TheMarkerSize,
    TheCameraParameters.CameraMatrix,
    TheCameraParameters.Distorsion, false);

// Transform the Rvec vector into a rotation matrix
cv::Mat R_MC; //R marker->cam
cv::Rodrigues(TheMarkers[i].Rvec, R_MC);
RT_MC(0,3)=TheMarkers[i].Tvec.at<float>(0,0);
RT_MC(1,3)=TheMarkers[i].Tvec.at<float>(1,0);
RT_MC(2,3)=TheMarkers[i].Tvec.at<float>(2,0);
```

Dentro de este método ArUco llama a la función *solvePnP* de OpenCV para resolver la ecuación 4.2 y obtener los parámetros extrínsecos de la cámara para finalmente guardar la traslación y rotación del marcador respecto de la cámara, lo que en la Figura 4.6a hemos denominado RT_MC.



(a) Matriz obtenida con el método *calculateExtrinsics*



(b) Calculo de la matriz RT_CW

$$RT_{CW} = RT_{MW} * RT_{CW}$$

Con esta información y teniendo en cuenta que conocemos la posición del marcador en nuestro sistema de referencia podemos calcular la matriz de transformación de la cámara respecto de nuestro origen de referencia (llamada RT_CW) y por tanto su posición 3D, como se puede ver en la Figura 4.6b. Para ello primero tenemos que calcular la inversa de la matriz RT_MC para obtener la matriz que transforma el centro del marcador en el centro de la cámara, después recuperamos la posición y orientación del marcador en el mundo (que han sido cargadas desde el fichero de configuración al arrancar el programa), pasamos esta información a formato matriz de transformación (matriz RT_MW) y operamos matrices para obtener la estimación de la posición 3D de la cámara.

```
Eigen::Matrix4d RT_CM;
// Calculate the inverse
RT_CM = RT_MC.inverse().eval();
RT_CW = RT_MW * RT_CM;

// Set the estimation value
```

```
totalWeight += tag->setEstimation(RT_CW);
detectedTags.push_back(tag);
```

La clase *Tag* donde guardamos la estimación contiene también la posición real del marcador así como una valoración de la calidad de la estimación que emplearemos en el siguiente apartado para la fusión de estimaciones. A este valor de calidad le llamamos *peso* de la estimación y se devuelve en el método que guarda la estimación para calcular la suma de los pesos y poder hacer una media ponderada de todas las detecciones que se van almacenando en el vector *detectedTags*.

4.2.3. Fusión de estimaciones

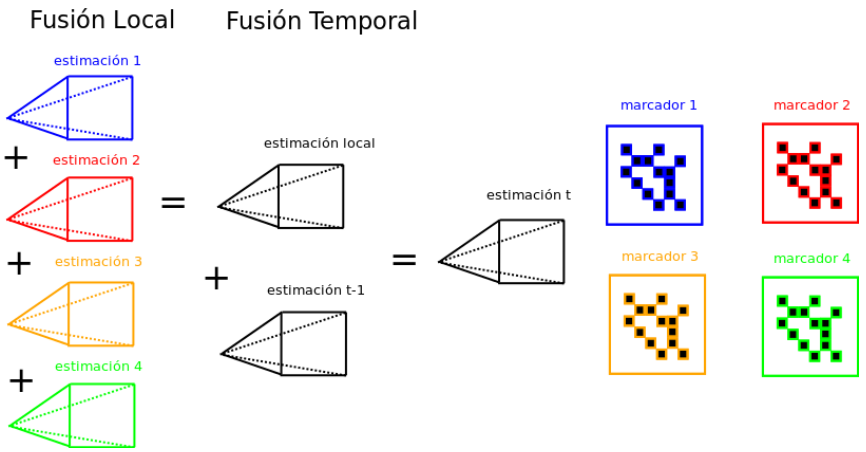


Figura 4.6: Fusión de estimaciones

Una vez procesadas cada una de las detecciones tenemos como resultado un vector con todas las estimaciones posibles, que podemos fusionar en una estimación final para la posición 3D de la cámara que *renderiza* la escena (Figura 4.6). Cómo método para fusionar las diferentes estimaciones emplearemos una media ponderada ya que la calidad de la estimación no es la misma para cada marcador detectado. Como ya hemos comentado, al guardar cada estimación se calcula un valor del *peso* de la misma para comprarla con las demás, en nuestro caso el criterio para determinar la calidad de una estimación será la distancia a la que se encuentra la cámara del marcador. Calculamos la distancia entre

la posición real del marcador y la cámara (la estimación obtenida para el marcador en concreto) y asignamos un valor de *peso* en función de unos rangos definidos, según podemos ver en la tabla 4.1.

Distancia (m)	**Peso**
<0.3	15
0.3 - 0.4	14
0.4 - 0.5	13
0.5 - 0.6	12
0.6 - 0.7	11
0.7 - 0.8	10
0.8 - 0.9	9
0.9 - 1	8
1 - 1.1	7
1.1 - 1.2	6
1.2 - 1.3	5
1.3 - 2	3
2 - 4	1
>4	0

Cuadro 4.1: Criterio para asignar un valor de peso

Estos rangos de valores se deciden a partir de la validación experimental del algoritmo de autolocalización que veremos en el capítulo de experimentos 5.3. Se asigna un peso muy bajo a los marcadores que estén muy alejados y dentro del rango de distancias en el que el error es aceptable para esta aplicación (por debajo de los 2 metros), se reparten los pesos de forma progresiva intentando que haya variedad (en intervalos de 10 cm) ya que en la escena que se emplee no habrá mucha diferencia de distancia ente los marcadores.

Como estamos trabajando con una posición 3D, por un lado tendremos que calcular una media ponderada para la posición y por otro calcular la orientación. La media de cada componente (x,y,z) de la posición es trivial y se calcula haciendo la suma de todos los elementos multiplicando cada uno por el *ratio* entre su peso y el peso total (suma de todos

los pesos), aplicando las fórmulas 4.3 y 4.4. Para la orientación trabajaremos separando la posible rotación de la cámara en componentes *roll*, *pitch* y *yaw* (rotación en grados sobre cada uno de los ejes de coordenadas) y además tendremos que tener especial cuidado al calcular la media para no obtener resultados incorrectos debido a discontinuidades (por ejemplo obtener que la media numérica entre 1º y 359º es 180º, justo lo contrario de la media real: 0º). Una forma de salvar las discontinuidades es trabajar con alguna función trigonométrica que sea continua, como las funciones seno o coseno. El problema de estas funciones es que, tras aplicarlas y calcular la media, si queremos aplicar la inversa de la función podemos obtener dos posibles valores para el ángulo, para solventarlo se calculará la media del seno y del coseno y se recuperará el valor del angulo con la función arcotangente, según la fórmula 4.5.

$$ratio_i = \frac{peso_i}{peso_{total}} \tag{4.3}$$

$$[x, y, z]_{fusion} = \sum ([x_i, y_i, z_i] * ratio_i) \tag{4.4}$$

$$\alpha_{fusion} = atan\left(\frac{\sum(sin(\alpha_i) * ratio_i)}{\sum(cos(\alpha_i) * ratio_i)}\right) \tag{4.5}$$

Además de las distintas estimaciones obtenidas de todos marcadores detectados también disponemos información de estimaciones anteriores, es decir de las fusiones instantáneas obtenidas en los fotogramas pasados. Como el movimiento de la cámara es suave, es interesante aprovechar esta información y aplicar también una fusión temporal entre la estimación actual y la de la imagen anterior. De este modo se evita que una mala estimación en un determinado instante provoque un defecto visible en la realidad aumentada (como que los objetos ficticios den un pequeño salto). En este caso se emplea siempre una ponderación de un 20 % para la nueva estimación.

Finalmente ya disponemos, en nuestra memoria compartida, de una posición 3D desde la que podemos *renderizar* la escena ficticia para que se vea aproximadamente desde el mismo punto de vista que la escena real.

4.3. Simulación de efectos físicos

El siguiente módulo es el encargado de simular los efectos de física que ocurren en la escena, en concreto se controlan las fuerzas generadas por colisiones y el efecto de la gravedad. En la escena disponemos de varios objetos (el robot, la pelota, un cilindro y dos superficies planas, una para la mesa y otra para el suelo) y cada uno de ellos tiene diferentes propiedades que Bullet tendrá en cuenta a la hora de calcular su posición. La pelota es un elemento dinámico (*Dynamic Rigid Body*) y su movimiento es totalmente simulado por Bullet, es decir en cada instante será el simulador de física el que indique su posición dentro de la escena. El cilindro, el cuál se emplea para la interacción con un elemento real que tenga esta forma (como por ejemplo una taza o un vaso), y las dos superficies planas son elementos estáticos (*Static Body*) y por tanto Bullet los tiene en cuenta para las colisiones pero nunca variará su posición. Por último el robot es un objeto *kinematico* (Kinematic Rigid Body) y es una mezcla de los dos anteriores, Bullet lo tiene en cuenta para las colisiones, sabe que se puede mover (actulizaremos su posición en cada paso de simulación), pero nunca aplicará una fuerza sobre él ya que el control de su movimiento lo tiene la aplicación, en nuestro caso el módulo de Ogre que simula la escena del robot caminado hacia la pelota.

La interacción de este programa con Bullet se puede dividir en dos pasos, uno inicial en el que se crea una escena de Bullet y se configuran los tres objetos que queremos simular y otro paso, que será iterativo, en el que se indique a Bullet que debe dar un paso de simulación del tiempo transcurrido y en el que se consulte la posición de la pelota.

4.3.1. Creación de la escena

Antes de ver la configuración de nuestra escena puede ser interesante conocer algunas de las clases de Bullet que permiten definir una escena y los elementos que ésta contiene. Para empezar, en función de la rigidez de los objetos disponemos de dos tipos de mundos, *btDiscreetDynamicsWorld* para cuerpos rígidos y *btSoftRigidDynamicsWorld* para cuerpos blandos. Además para la creación de ambos tipos de mundo se necesita instanciar una serie de objetos:

- *broadphase*: algoritmo que devuelve pares de objetos en colisión o a punto de colisionar
- *collisionConfiguration*: configuración a bajo nivel de parámetros para la detección de colisiones, como la cantidad de memoria empleada
- *dispatcher*: algoritmo que maneja las colisiones procesando los datos proporcionados por el *broadphase*
- *solver*: aplica todos los efectos de física sobre cada objeto.

Para nuestra escena utilizaremos un *btDiscreetDynamicsWorld* ya que sólo contiene cuerpos rígidos, una configuración de uso general para el resto de parámetros y una vez creada la instancia del mundo asignaremos el valor de la aceleración de la gravedad.

```
broadphase = new btDbvtBroadphase();
collisionConfiguration = new btDefaultCollisionConfiguration();
dispatcher = new btCollisionDispatcher(collisionConfiguration);
solver = new btSequentialImpulseConstraintSolver;

dynamicsWorld = new btDiscreteDynamicsWorld(dispatcher,
           broadphase,solver,collisionConfiguration);
dynamicsWorld->setGravity(btVector3(0, 0, -10));
```

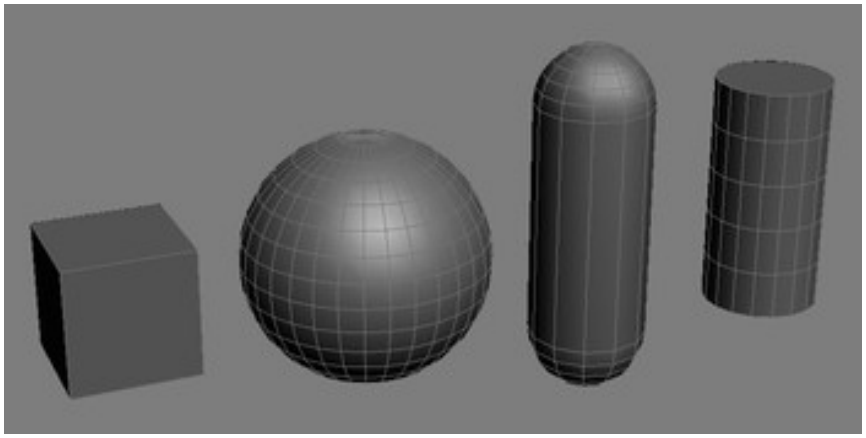


Figura 4.7: Formas primitivas de Bullet

Una vez creado el mundo de Bullet sólo falta crear y añadir los distintos elementos (*btRigidBody*) que tenemos. Por cada elemento tendremos que definir su forma, el valor de su masa, su posición inicial y su inercia. Para la forma (*Collision Shape*) Bullet ofrece una serie de primitivas (esfera, caja, cilindro, etc) que además se pueden combinar para obtener formas más complejas, también se pueden definir *Collision Shapes* a partir de mallas de puntos, útiles para definir por ejemplo una superficie a partir de un mapa de alturas, pero se recomienda el uso de las formas primitivas (Figura 4.7) por tener un mejor rendimiento en la detección de colisiones. El resto de parámetros no requieren de mayor explicación, pero es importante comentar que un objeto con masa = 0 se considera de masa infinita y por tanto inamovible (será el caso de nuestro cilindro y las dos superficies planas). Además, si un objeto es *kinemático* como el robot, se debe llamar a un método para configurar el cuerpo antes de añadirlo al mundo. Podemos ver a continuación un ejemplo de cómo se crea el cuerpo rígido del robot.

```
// Create the robot collition shape using a capsule shape
btCapsuleShape* robotShape = new btCapsuleShape(0.002,0.09);

// Initial motion state
btDefaultMotionState* robotMotionState = new btDefaultMotionState(
          btTransform(initialOrientation, initialPosition));
```

```
// Calculate the initial inertia
robotShape->calculateLocalInertia(mass,robotInertia);
btRigidBody::btRigidBodyConstructionInfo robotRigidBodyCI(mass,
          robotMotionState,robotShape,robotInertia);
robotRigidBody = new btRigidBody(robotRigidBodyCI);

// The robot is kinematic
robotRigidBody->setCollisionFlags(robotRigidBody->getCollisionFlags()
                       | btCollisionObject::CF_KINEMATIC_OBJECT);
robotRigidBody->setActivationState(DISABLE_DEACTIVATION);

// Adding the robot to the Bullet world
dynamicsWorld->addRigidBody(robotRigidBody);
```

En este caso se ha elegido una forma de cápsula para para el robot, parecida a la que podemos ver en la Figura 4.8 . Creando el resto de objetos de nuestra escena de forma similar terminamos de inicializar la simulación de Bullet, dejando el mundo de físicas listo para la actualización de la escena en cada ciclo de refresco.

4.3.2. Actualización de la escena

Cada vez que se cumpla el periodo de tiempo establecido para refrescar la escena, se emplearan los objetos instanciados durante la inicialización (mundo de física y cada cuerpo rígido) para, primero avanzar la simulación el tiempo que haya transcurrido y después obtener o informar la posición de los objetos en el instante actual. Para dar un paso de simulación basta con llamar al método *stepSimulation* indicando el tiempo que se quiere avanzar la simulación en segundos, en nuestro caso la frecuencia de refresco es de 60Hz tanto para la simulación de físicas como para el renderizado de los gráficos. Al llamar a este método se actualizará la posición (*Motion State*) de todos los objetos dinámicos de la escena, en nuestro caso la pelota. Para obtener los valores del nuevo estado del objeto podemos llamar al método *getMotionState* que devuelve la transformación del cuerpo

Figura 4.8: *CapsuleShape* para modelar un humanoide

(rotación y traslación).

```
// Step the simulation
dynamicsWorld->stepSimulation(totalDiff/1000000);
...
...
// Get the new motion state
sphereRigidBody->getMotionState()->getWorldTransform(trans);
```

De forma similar se puede actualizar la posición del robot guardando el nuevo *Motion State* de este objeto, para que Bullet lo tenga en cuenta en el próximo paso de simulación

4.4. Generación de las imágenes enriquecidas

Cómo módulo que gestiona la interfaz gráfica se ha implementado una aplicación Ogre que se encarga de crear una ventana, cargar la configuración y los recursos necesarios, llamar al método que crea la escena, asignar un *listener* que recibe un evento cada vez que se va a generar un nuevo fotograma y pintar la imagen.

Antes de entrar al detalle de la creación y actualización de nuestra escena, veamos las clases fundamentales que se necesitan para configurar un mundo en Ogre. El control de la escena lo tendrá una instancia de la clase *SceneManager* que se encarga del seguimiento de todos los elementos de la escena, tanto objetos, como la cámara, o la iluminación. Dentro de la escena existen nodos (*SceneNode*) que contienen información sobre la posición, orientación y tamaño de un objeto, el *SceneManager* crea un nodo raíz que sirve de referencia absoluta y el resto de nodos pueden ser hijos del nodo raíz o de otros existentes, pero hay que tener en cuenta que su posición y orientación será relativa a su nodo padre. Un *SceneNode* por sí solo no sería *renderizado*, es necesario que esté asociado a un objeto con una apariencia. Si la apariencia viene dada por un malla de puntos que ha sido cargada como un recurso podemos utilizar la clase *Entity*, si por el contrario vamos a generar nosotros la forma empleando formas simples (como triángulos) podemos usar la clase *ManualObject*. Podemos ver un ejemplo de jerarquía de elementos de Ogre en la Figura 4.9.

4.4.1. Creando la escena

Veamos ahora la implementación del método *createScene* que configura nuestra escena particular. Antes de invocar a este método ya se ha instanciado un *SceneManager* en el atributo mSceneMgr de nuestra aplicación que utilizaremos para indicar la configuración de luz y para ir añadiendo los diferentes objetos que componen la escena, el robot y la pelota, ya que el resto de elementos que vimos en la escena de Bullet pertenecerán a la escena real.

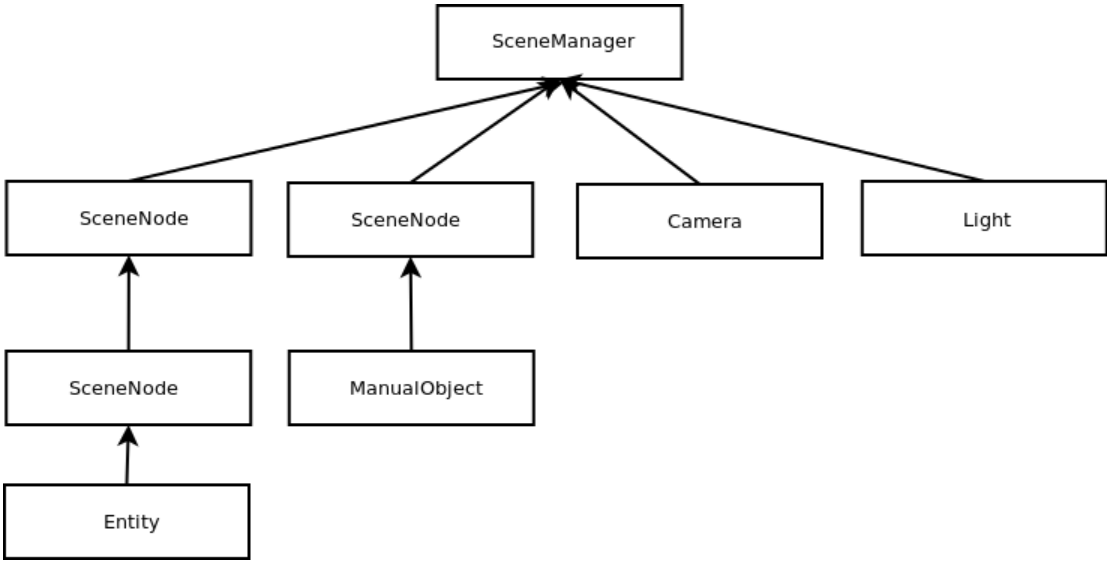


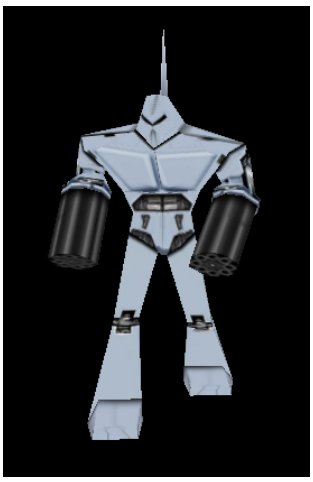
Figura 4.9: Elementos de una escena de Ogre

Los pasos a seguir para definir un objeto son, primero crear una *Entity* indicando el nombre de un recurso .mesh que contenga la información de la malla de puntos o bien crear un *ManualObject* y generar una forma con puntos y líneas (de la misma forma que se haría en OpenGL). Después se crea un nuevo nodo como hijo de alguno ya existente, se indica su posición, orientación y escalado y finalmente se asocia un *Entity* o *ManualObject* para tener el objeto listo para ser *renderizado*. Veamos el ejemplo de la definición del robot:

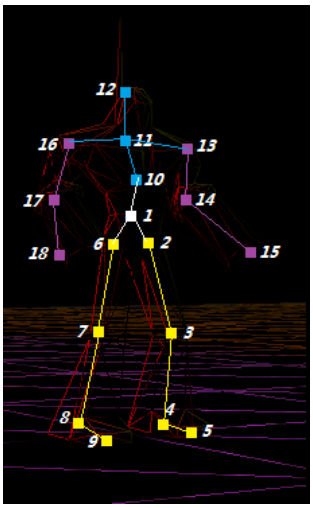
```
// Create the entity
mRobotEnt = mSceneMgr->createEntity("Robot", "robot.mesh");

// Create the scene node
mRobotNode = mSceneMgr->getRootSceneNode()->createChildSceneNode("SphereNode",
                      initialPosition, initialOrientation);

// Attach the entity to the node
mRobotNode->attachObject(mRobotEnt);
```

En el caso del robot, al crear su *Entity* no sólo se carga su malla de puntos si no también su *esqueleto*, es decir las diferentes partes que componen su cuerpo y que permiten articularlo. Con esta información podremos pedir al *Entity* que genere una animación del

personaje (*AnimationState*) para que por ejemplo este mueva las piernas como si caminara. Se añade la pelota a la escena de la misma forma y además se añaden los ejes de coordenadas en el origen para tenerlo como referencia.



(a) Textura del robot



(b) Esqueleto del robot

Puede parecer que ya está lista nuestra escena, pero aún falta lo más importante para que la aplicación sea de realidad aumentada, falta definir una superficie que se situe siempre en el fondo y que muestre la imagen real obtenida por la cámara usando una textura dinámica.

```
// Create the texture
texture = Ogre::TextureManager::getSingleton().createManual(
"DynamicTexture", 640, 480,
Ogre::TU_DYNAMIC_WRITE_ONLY_DISCARDABLE, ...);

// Create a material using the texture
Ogre::MaterialPtr material = Ogre::MaterialManager::getSingleton().create(
"DynamicTextureMaterial",
Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);

// Create a background rectangle covering the whole screen
Ogre::Rectangle2D* rect = new Ogre::Rectangle2D(true);
rect->setCorners(-1.0, 1.0, 1.0, -1.0);
rect->setMaterial("DynamicTextureMaterial");
```

```
// Render the background before everything else
rect->setRenderQueueGroup(Ogre::RENDER_QUEUE_BACKGROUND);

// Attach background to the scene
Ogre::SceneNode *node;
node = mSceneMgr->getRootSceneNode()->createChildSceneNode("Background");
node->attachObject(rect);
```

Primero creamos la textura indicando que tendrá el tamaño de la imagen y que será dinámica, después creamos un material formado por la textura y finalmente creamos un rectángulo a partir del nuevo material, que ocupa toda la pantalla e indicamos que se debe *renderizar* como fondo (siendo el primero en la cola de *renderizado*). Para que se tenga en cuenta en nuestra escena creamos un nuevo nodo como hijo del nodo raíz y asociado al rectángulo que pintará la imagen de cámara en el fondo.

Con estos pasos logramos configurar nuestra escena simulada para la demostración de realidad aumentada, podemos ver cómo queda la escena con todos sus elementos en la Figura 4.10

4.4.2. Actualizando la escena

Para actualizar los elementos que forman nuestra escena podemos utilizar el método *frameRenderingQueued* al que se envía un evento cada vez que se *renderiza* un nuevo *frame* de la ventana. Como tareas a realizar en cada ciclo de refresco tenemos que actualizar la posición 3D de la cámara con la última estimación, actualizar la posición de la pelota con la última situación que haya simulado Bullet, aplicar la lógica que hace que el robot camine hacia la pelota y por último refrescar la textura dinámica del fondo con la última imagen obtenida de la cámara.

Para actualizar la posición 3D de la cámara y la pelota tenemos que acceder a los atributos que contienen el *SceneNode* de la pelota y la instancia de la cámara y llamar a

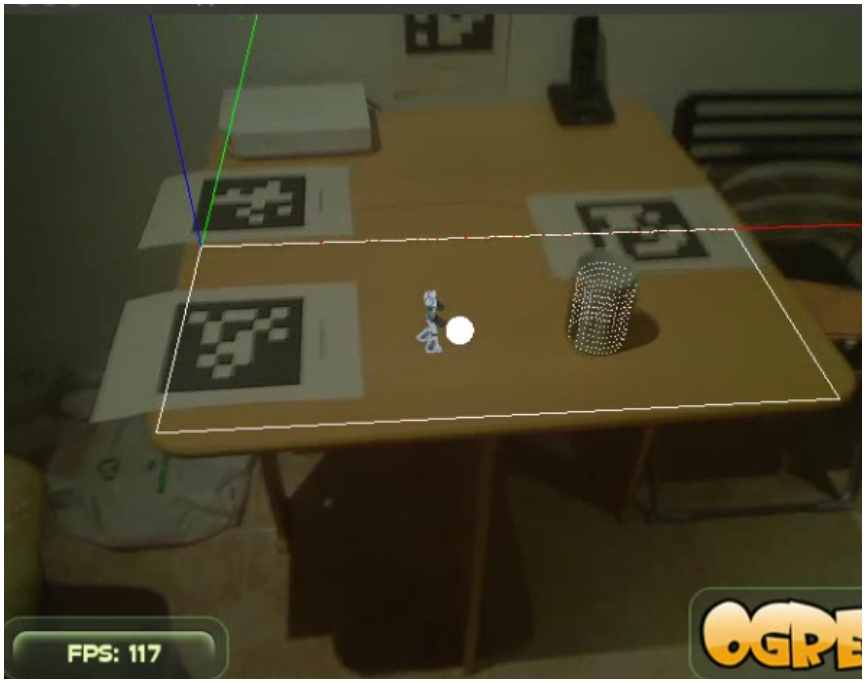


Figura 4.10: Escena virtual para el componente demoAR

sus métodos *setPosition* y *setOrientation*. Veamos el ejemplo de la pelota.

```
// update the shpere pose3D
mSphereNode->setPosition(newPosition);
mSphereNode->setOrientation(newOrientation);
```

Para la lógica del robot, se calcula la distancia que ha avanzado en el tiempo transcurrido (que se recibe en el envento recibido), se obtiene su nueva posición teniendo en cuenta que avanza hacia la pelota y finalmente se comprueba si ya ha alcanzado la pelota para cambiar su animación a estado de reposo ("*Idle*") o para informar a la animación del tiempo transcurrido para que se actualice.

```
frameRenderingQueued(const Ogre::FrameEvent &evt){
    Ogre::Real move = mWalkSpeed * evt.timeSinceLastFrame;
    mDistance -= move;
    if (mDistance <= 0.0f){
        mNode->setPosition(mDestination);
        mAnimationState = mEntity->getAnimationState("Idle");
    }else {
```

```
        // move the node
        mNode->translate(mDirection * move);
        // update the animation
        mAnimationState->addTime(evt.timeSinceLastFrame);
    }
}
```

Por último, para refrescar el fondo de la escena se recupera el buffer de bits que contiene la información de la textura y se recorre actualizando cada pixel con el valor de la nueva imagen.

Capítulo 5

Experimentos

En este capítulo se hablará de las pruebas que se han realizado sobre el componente desarrollado en este proyecto. Primero veremos una muestra de la ejecución del componente demoAR tanto en entorno real como simulado, para después analizar con una serie de experimentos los diferentes módulos que forman el programa: el algoritmo de localización, el motor de físicas y el motor gráfico.

5.1. Ejecución del componente demoAR con cámara real

Realizaremos una prueba del componente demoAR para observar su ejecución con un flujo de imágenes reales. Para ello lo primero es preparar el escenario sobre el que se quiere reflejar la realidad aumentada. Utilizaremos una mesa sobre la que situar los elementos ficticios y una taza para interactuar con ellos, además para esta prueba se emplearán cuatro marcadores de AprilTags (tres de ellos sobre la mesa y uno en una pared estando perpendicular al resto). En la Figura 5.1 podemos ver la disposición de esta escena.

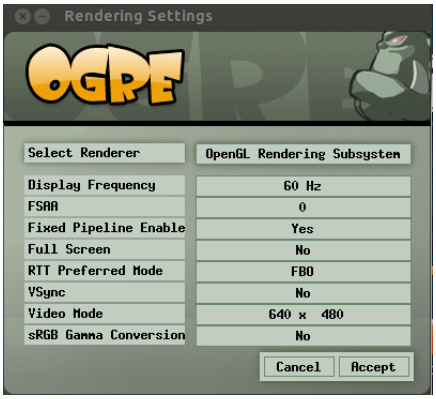
Antes de ejecutar el programa se debe rellenar el fichero de configuración con los datos necesarios para la prueba, en este caso la posición y orientación de cada marcador (identificado por su número de ID), la posición inicial del robot, la pelota y el cilindro (que debe

Figura 5.1: Escena para la prueba del componente demoAR

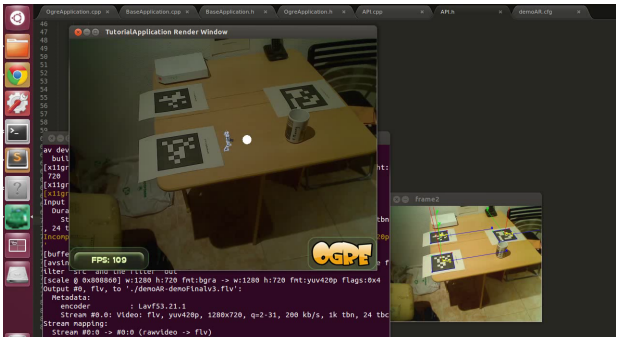
coincidir con la taza), las dimensiones de la mesa y por último la dirección IP y puerto del componente ICE que va a proporcionar las imágenes. Finalmente conectamos la cámara por USB al ordenador y lanzamos el componente *cameraserver* para que empiece a servir el flujo de imágenes.

Lanzamos el componente demoAR y el módulo de GUI abre una ventana para configurar el renderizado de Ogre (Figura 5.2a). Seleccionamos que queremos renderizar empleando OpenGL, con una frecuencia de 60 Hz y con una resolución de 640x480. A continuación se abre la ventana que muestra la aplicación de realidad aumentada y una ventana auxiliar que muestra los marcadores detectados en cada momento, como se puede ver en la Figura 5.2b. Tomamos la cámara, enfocamos la escena e inmediatamente se observan el robot y la pelota sobre la mesa. Además, la ventana auxiliar nos muestra la detección de los cuatro marcadores. Al realizar movimientos suaves con la cámara se puede visualizar el escenario desde diferentes posiciones y ángulos comprobando como los elementos ficticios se siguen viendo correctamente desde esas nuevas posiciones, sin producirse saltos ni parpadeos.

Continuamente se simula la misma situación en la que el robot, partiendo de la posición inicial que hemos configurado, camina hacia la pelota hasta que la golpea. Según la trayectoria que tome la pelota ésta puede rebotar contra la taza o seguir un camino recto,



(a)



(b)

Figura 5.2: Ventanas del componente demoAR

pero al alcanzar el borde de la mesa la bola se precipita hacia el suelo, rebota contra este y sigue rodando hasta que se reinicie la escena.

A continuación podemos ver secuencias de imágenes que muestran estos efectos:

Figura 5.3: Robot golpeando la pelota

Figura 5.4: Pelota rebotando en la taza

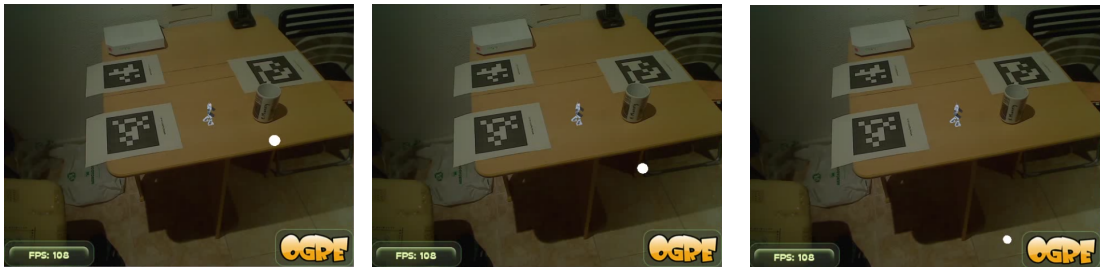
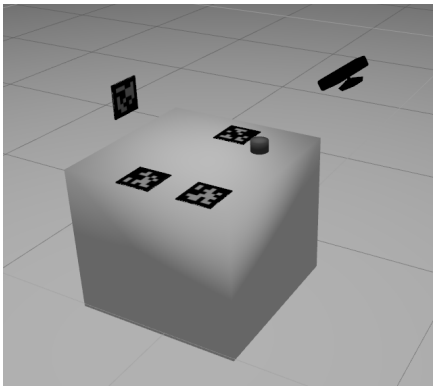


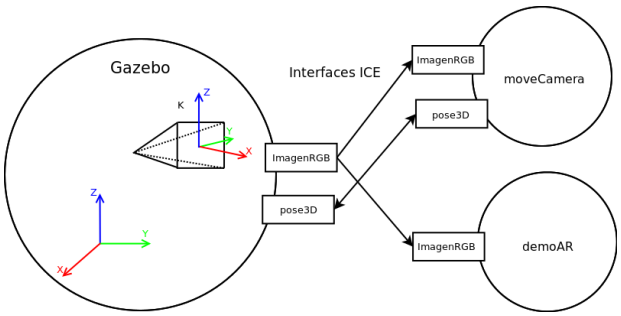
Figura 5.5: Pelota en el borde de la mesa

5.2. Ejecución del componente demoAR con cámara simulada

A continuación ejecutaremos del componente alimentado por un flujo de imágenes de un cámara simulada, para comprobar que el componente no depende de la cámara que emite las imágenes. Para ello disponemos de un mundo de Gazebo que replica la escena anterior para que el experimento se realice en el mismo escenario, el plugin *flyingCamera* para la cámara simulada y el componente *moveCamera* para operar la cámara. En este caso la mesa se modelará con una caja del mismo tamaño que la real, mientras que la taza (el elemento real que simulamos en Bullet) se modela con un cilindro en Gazebo. Podemos ver una captura de este escenario simulado en la Figura 5.6a.



(a)



(b)

Figura 5.6: Experimento con Gazebo

Antes de ejecutar debemos modificar el fichero de configuración para indicar la conexión

ICE de la cámara simulada, cambiar los parámetros de calibración de la cámara y lanzar el mundo de Gazebo y el componente *moveCamera*. Podemos ver un esquema con los interfaces ICE que intervienen en la Figura 5.6b.

Ejecutamos el componente demoAR y observamos un comportamiento similar al del entorno real. A continuación podemos ver unas capturas de la ejecución:

Figura 5.7: Pelota rebotando en el cilindro

Figura 5.8: Pelota cayendo de la mesa

5.3. Estudio de precisión del módulo de autolocalización

A continuación se describen distintas pruebas que se han realizado sobre el módulo de autolocalización. Para el experimento se usa de nuevo el simulador Gazebo para tener la posición 3D verdadera de la cámara y poder conocer el error, y el componente *moveCamera* para operar la cámara variando su posición y orientación. En cada prueba se fijarán algunos de los parámetros que definen la ubicación de la cámara (posición en coordenadas x,y,z y orientación como roll,pitch y yaw) variando otros para ver cómo afectan al error de la estimación. Para el experimento se colocan los marcadores en el suelo y la cámara mirará

hacia abajo, podemos ver una captura de la escena en Gazebo y una captura del componente *moveCamera* durante una de las pruebas en la Figura 5.9.

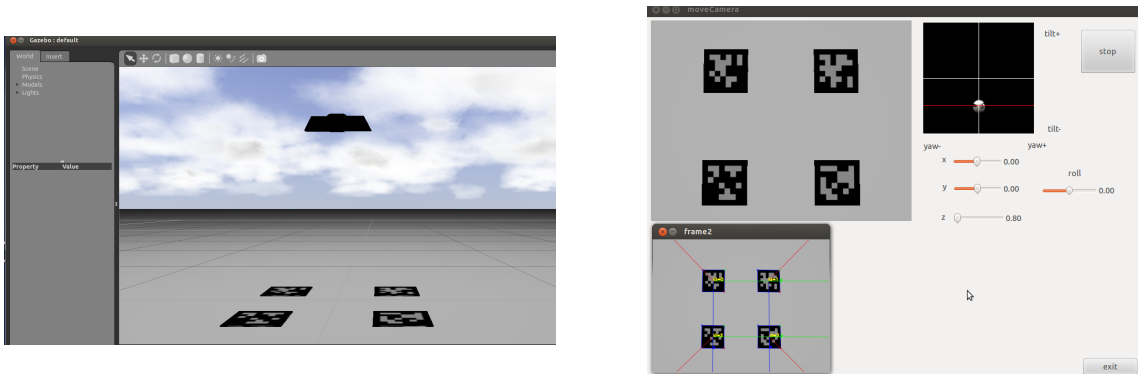


Figura 5.9: Experimentos con Gazebo

Pruebas de excentricidad

Para comenzar realizaremos una prueba sin desplazar ni girar la cámara, simplemente variaremos la posición de un marcador para que la cámara lo capte en distintos puntos de la imagen y estudiar si lo cerca o lejos que está un marcador del centro de la imagen puede afectar a la estimación. Para la pruebas se lanzan nueve ejecuciones modificando en cada una la posición del marcador de modo que empezamos situándolo en la esquina superior izquierda de la imagen, después en el lado superior y así hasta acabar en la esquina inferior derecha. Tiene sentido pensar que la estimación se comportará de forma similar cuando el marcador está en las cuatro esquinas y también cuando está en uno de los laterales de la imagen, pero comprobaremos nueve posiciones para estar seguros. Tras la ejecución de la prueba se obtiene los resultados de la Figura 5.10.

Primero comentar que estudiaremos por un lado el error en Z (en la profundidad) y por otro el combinado en X e Y (error lateral). Tras una primera observación a las gráficas sorprende ver que el error en XY es mayor cuando el marcador se encuentra en uno de los laterales que cuando está en una esquina, cuando la distancia al centro de la imagen es mayor y cabe pensar que se produciría un mayor error. También se observa que en algunos

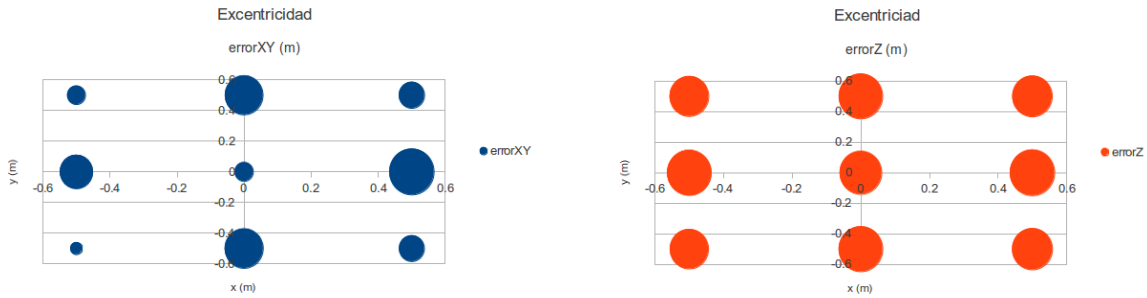


Figura 5.10: Prueba de excentricidad

caso el error obtenido al situar el marcador en una esquina es igual o incluso menor a cuando el marcador se halla en el centro (que a priori parece una mejor posición). Como conclusión de estos hechos se puede pensar que cuando el marcador está desplazado en dos ejes (casos de las esquinas) hay una mayor referencia visual que cuando el marcador es desplazado en un solo eje y por tanto se obtiene una mejor estimación. En cuanto a los valores del error, el combinado en XY toma valores entre 3 milímetros en el mejor de los casos y 3 centímetros en el peor, mientras que el error en Z, aún dependiendo menos de la posición del marcador varía entre 10 y 13 cm.

Pruebas de número de marcadores

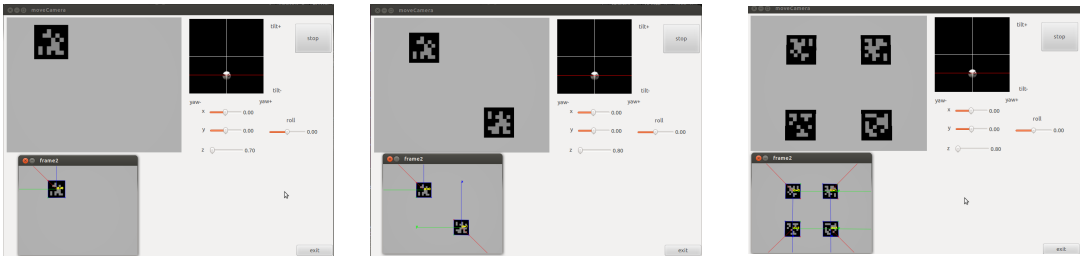


Figura 5.11: Prueba variando el número de marcadores

A continuación probaremos cómo se comporta la estimación al variar el número de marcadores que se detectan en la imagen. Para ello se probará con uno, dos y cuatro

marcadores, y para que la prueba sea lo más justa posible siempre se situarán en una esquina (para tener las mismas condiciones de excentricidad), como se puede ver en la Figura 5.11. El experimento se realiza manteniendo todos los parámetros fijos excepto el valor de z que irá aumentado (alejándonos de los marcadores) para ver como afecta la variación de la distancia en los tres casos. Los resultados se pueden ver en la Figura 5.12

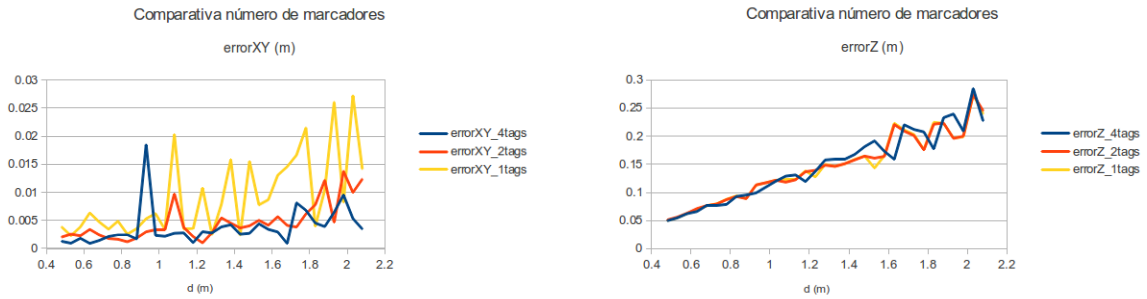


Figura 5.12: Comparativa entre distinto número de marcadores

De estas dos gráficas podemos obtener varias conclusiones. Primero se comprueba cómo el error en XY disminuye al emplear un mayor número de marcadores. Además, se observa que existen picos esporádicos causados por una mala estimación en un instante puntual que pueden producir un salto en la aplicación de realidad aumentada. Al trabajar con una cámara real y ser muy poco probable obtener el mismo fotograma en dos instantes seguidos, esto picos quedarán atenuados con la fusión temporal. Finalmente, se observa que cuando todos los marcadores están situados en el mismo plano, el error en profundidad no depende del número de marcadores.

Prueba de marcadores en distinto plano

Partiendo de la última conclusión parece interesante probar qué ocurre con el error en profundidad cuando los marcadores están en distinto plano. Para ello se lanza un nuevo experimento en el que cuatro marcadores se sitúan en el suelo y uno se halla perpendicular al resto. Veamos los resultados en la Figura 5.13

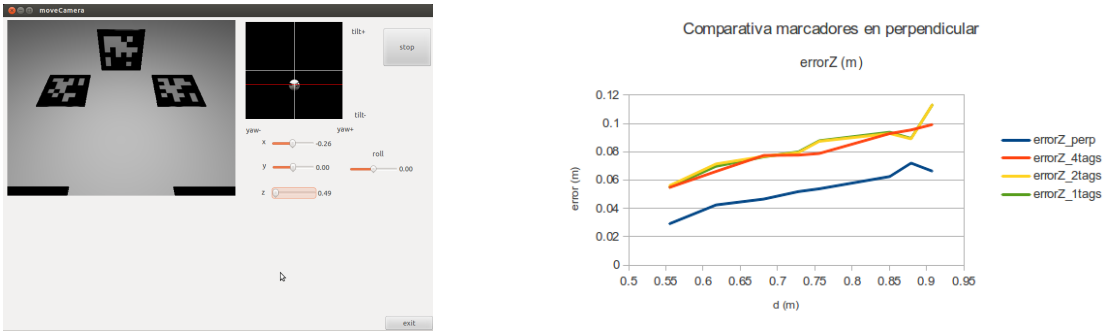


Figura 5.13: Prueba marcadores no coplanares

Observando la gráfica comprobamos cómo efectivamente al obtener información de marcadores situados en distintos planos se ve reducido el error de profundidad desplazándose la recta hacia abajo.

Pruebas de traslación

Ahora se probará a variar la posición de la cámara en tres ejecuciones diferentes, variando en cada una x, y o z, para estudiar cómo varía el error de la estimación al desplazar la cámara. Para la prueba se emplean cuatro marcadores situados en las esquinas. Podemos ver los resultados en las Figuras y .

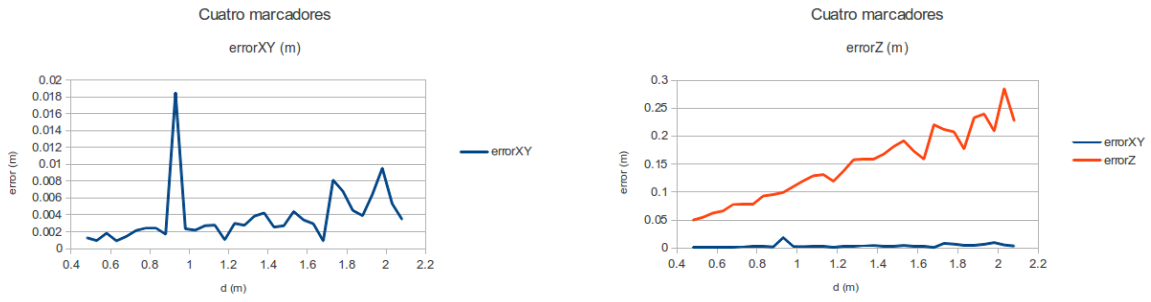


Figura 5.14: Resultado desplazamiento en Z

Si analizamos la variación de la altura (valor de z), podemos observar cómo el error XY es mucho menor que el que se produce en profundidad (errorZ). Por suerte a la hora de utilizar el algoritmo en la aplicación de realidad aumentada el error en XY es más molesto

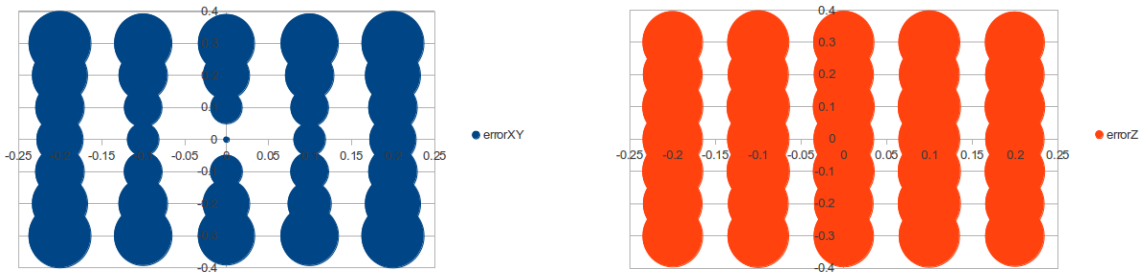


Figura 5.15: Resultado desplazamiento en x e y

que el de Z, ya que este último sólo afecta a que los elementos ficticios se vean un poco más grandes o más pequeños, mientras que el error en x e y puede hacer que el efecto de objetos reales y ficticios colisionando falle. Si nos fijamos en la variación de x e y, observamos cómo el errorXY aumenta al alejarnos de los cuatros marcadores (con un movimiento paralelo al plano XY), mientras que el error en profundidad (errorZ) no se ve afectado por este desplazamiento.

Pruebas de rotación

Para finalizar las pruebas sobre el algoritmo de autolocalización vamos a estudiar cómo se comporta el error frente a cambios en la orientación de la cámara. Al igual que en el experimento anterior emplearemos cuatro marcadores situados en las esquinas y se ejecutaran 3 pruebas variando en cada una el valor de uno de los ángulos: *roll*, *pitch* o *yaw*, de tal forma que en todo momento se estén detectando los cuatro marcadores. El resultado de los experimentos se puede ver en la Figura 5.16 donde se muestran tres gráficas una para cada ángulo.

En esta última prueba se comprueba que el error obtenido al variar la orientación de la cámara toma valores de entre 0º y 3º, y por tanto no es muy pronunciado.

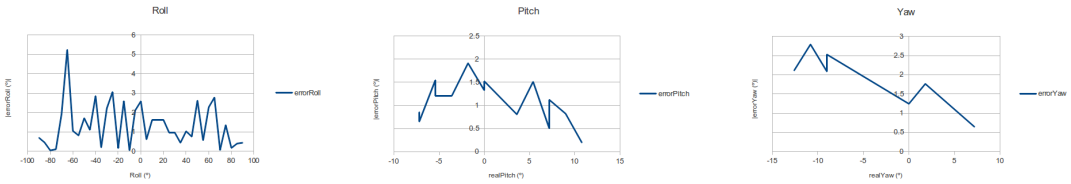


Figura 5.16: Prueba variando el *roll*, *pitch* y *yaw*

5.4. Experimento con Bullet

Con este experimento se quiere probar el motor de físicas Bullet para simular la caída de una pelota sobre un suelo gaussiano. Como en esta prueba nos queremos centrar en la simulación de físicas el renderizado de la imagen se realizará con un sencillo programa que utilice OpenGL. Para la prueba se crea el componente *skaterball* que configura una escena de Bullet con un pelota situada a cierta altura y un suelo con forma Gaussiana para experimentar con la creación de *Shapes* a partir de mallas de puntos.

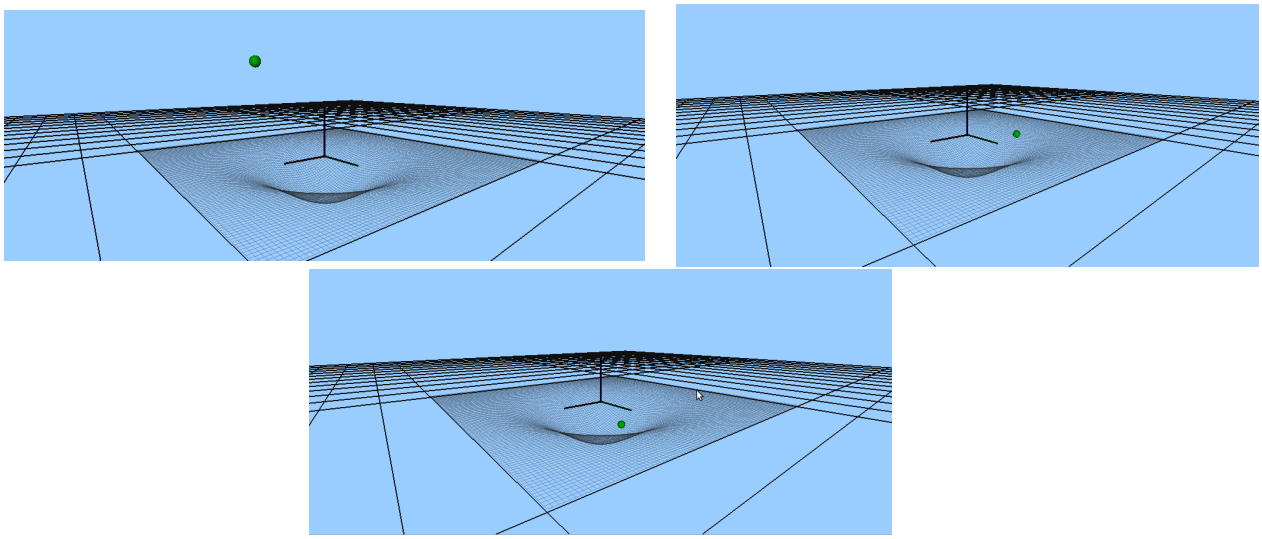


Figura 5.17: Pelota atrapada en la Gaussiana

Al ejecutar el componente se observa cómo la pelota se deja caer desde una cierta altura y va cayendo hasta alcanzar el suelo gaussiano donde choca y rueda por la superficie del suelo hasta alcanzar el borde opuesto. En este caso la pelota no alcanza la energía cinética suficiente para *escapar* del suelo Gaussiano y se queda oscilando entre un borde y

el opuesto hasta quedar completamente parada en el fondo, tal como haría una pelota real en un escenario similar. Podemos ver una secuencia de capturas de la prueba en la Figura 5.17.

Para comprobar cómo afecta un cambio en la escena al comportamiento de físicas, repetimos el experimento pero dejando caer la pelota desde una altura mayor. En este caso observamos cómo la pelota cae, sigue la trayectoria Gaussiana del suelo y al alcanzar el borde opuesto sí logra salir de la Gaussiana para seguir rodando hasta el final del suelo donde se precipita al vacío. Podemos ver la secuencia en la Figura 5.18.

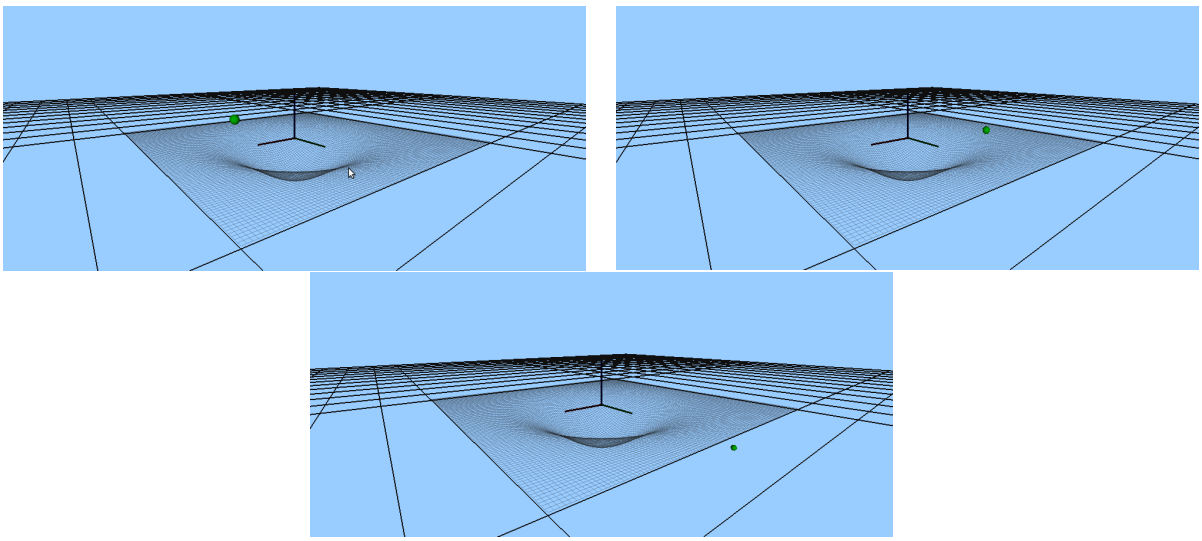


Figura 5.18: Pelota escapando de la Gaussiana

5.5. Experimento con Ogre

En este experimento se juega con Ogre para crear una escena virtual con un personaje cargado a partir de un fichero *.mesh* que se desplaza por el escenario entre tres puntos y estando animado para que parezca que anda. Además, se programa un lógica para que cada vez que se *renderiza* un fotograma se avance al robot en la dirección en la que se dirige y al alcanzar un punto se gire, mirando al nuevo destino y cambiando así su dirección.

En la Figura 5.19 se puede ver el resultado, un robot que se desplaza entre una serie

de puntos prefijados y cuando alcanza uno se gira para caminar hacia el siguiente.

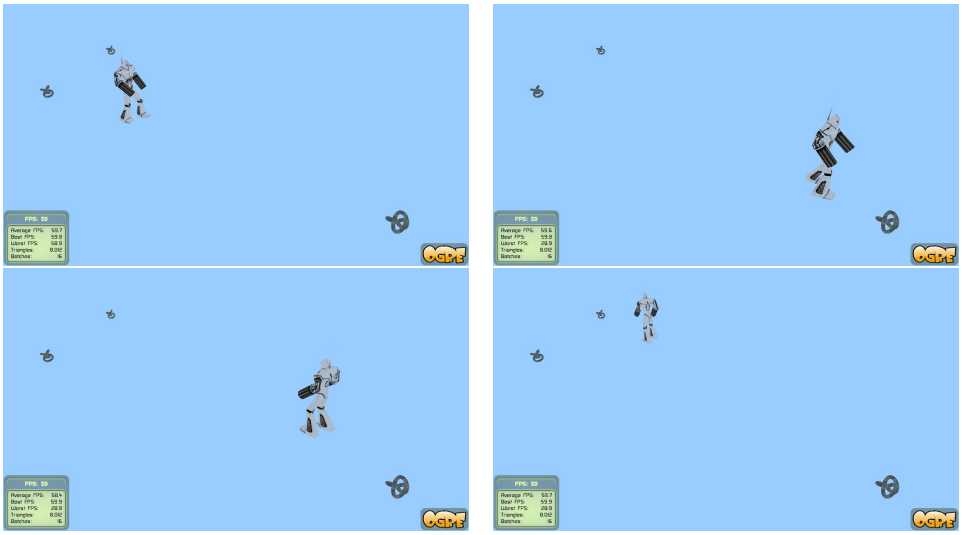


Figura 5.19: Robot caminando en un mundo Ogre

Capítulo 6

Conclusiones

Hasta ahora se han expuesto el contexto del trabajo, los objetivos que se han abordado en el mismo y finalmente el desarrollo que se ha elaborado para cumplirlos, así como su validación con varios experimentos. En este último capítulo se hablará de las conclusiones de este Proyecto Fin de Carrera y de los futuros trabajos que pueden surgir a partir del mismo.

6.1. Conclusiones

En el desarrollo final de este proyecto se ha logrado cumplir el objetivo principal que recordemos consistía en construir una aplicación de realidad aumentada realista que aunara diferentes técnologías. En concreto se ha conseguido crear una aplicación que emplea un algoritmo de autolocalización visual, un módulo de renderizado de gráficos, un módulo de simulación de efectos físicos y todo ello haciendo uso de una gran variedad de librerías, como se pudo ver en el capítulo de Infraestructura 3, logrando así obtener una aplicación muy heterogénea en cuanto a tecnologías.

Se exponen a continuación cada uno de los subobjetivos enunciados en el capítulo 2 para analizar cómo se han cumplido:

- *Desarrollo de un algoritmo de autolocalización visual*

  Se ha diseñado y programado un algoritmo de autolocalización visual basado en balizas, que como vimos en el capítulo de Desarrollo 4, se ha dividido en tres pasos. Primero se realiza un procesado de la imagen para buscar balizas, que recordemos son marcadores *AprilTags* de la familia 36h11. Después se calcula una estimación de la posición y orientación 3D de la cámara por cada detección, resolviendo para ello las ecuaciones de proyección con la función *solvePnP* de *OpenCV*. Por último, se aprovecha la información de cada detección realizando una fusión espacial, lo que mejora la robustez, y se aplica una fusión temporal para tener en cuenta la estimación del instante anterior y evitar cambios bruscos. Además, el algoritmo ha sido validado, como se vio en el capítulo de Experimentos 5, comprobando que estima la posición de la cámara con un error aceptable para su uso en aplicaciones de realidad aumentada.

- *Creación de una escena virtual con efectos gráficos y de físicas*

  La escena virtual diseñada para la realidad aumentada incluye una serie de efectos que hacen que ésta sea llamativa, como son objetos dinámicos y animados que interactúan con otros elementos simulando colisiones y efectos físicos como la gravedad. Para ello se han empleado dos potentes librerías de gráficos y físicas, *Ogre* y *Bullet*, que permiten un diseño de la escena orientado a objetos y poder abstraernos de otras tecnologías de más bajo nivel, como OpenGL, para centrarnos en la lógica de nuestra aplicación. En el capítulo de Desarrollo se describe en detalle el uso de estas dos librerías para que pueda servir de referencia a futuros proyectos que requieran emplear estas tecnologías.

- *Desarrollo de una aplicación de realidad aumentada*

  Finalmente se ha desarrollado una aplicación de realidad aumentada que puede ser ejecutada con un movimiento suave de cámara, que consigue una sensación de estabilidad de los objetos ficticios sobre los reales al mover la cámara y que además

muestra una interacción entre elementos virtuales y elementos reales (como es el caso de la pelota rebotando en la taza y cayendo del borde de la mesa al suelo). Para su desarrollo se han generado unas 2000 líneas de código de la cuales 500 forman parte del algoritmo de autolocalización, 1320 de la interfaz gráfica (simulación de gráficos y físicas) y el resto de diferentes módulos que dan soporte al programa. Además, en el capítulo de Experimentos se comprobó que el componente de realidad aumentada funciona independientemente del origen del flujo de imágenes, ya que se ha validado tanto con una cámara real como con una simulada.

Al igual que con los objetivos también se puede comprobar que los requisitos definidos en el capítulo 2 se cumplen:

- La solución desarrollada ha sido programada en lenguaje C++ y ha seguido la filosofía JdeRobot de componentes modulares que se pueden comunicar mediante las interfaces definidas en la plataforma. Por ejemplo, en el capítulo de experimentos 5.3 vimos cómo el componente se comunica con el simulador Gazebo mediante un plugin y cómo además interactúa con otros dos componentes JdeRobot, el *cameraServer* y el *moveCamera*.

- El componente desarrollado se ha probado a ejecutar sobre Ubuntu 12.04 tanto en arquitecturas de 32 como de 64 bits.

- Se ha logrado que la ejecución de la aplicación sea fluida, sin retardos ni parpadeos, de modo que el efecto de realidad aumentada sea realista y parezca que tanto elementos virtuales como reales forman parte del mismo escenario.

- Al ejecutar el componente se observa cómo los elementos virtuales se visualizan correctamente sobre el fondo real sin saltos ni temblores al desplazar la cámara suave-

mente lo que nos indica que el algoritmo de autolocalización cumple con el requisito de robustez necesario para esta aplicación.

- El componente permite modificar los parámetros que definen la escena, así como los datos de las balizas y de la comunicación ICE sin tener que recompilar el código mediante un fichero de configuración.

En el capítulo de Introducción se comentaba que una de las motivaciones de este proyecto es un videojuego de realidad aumentada desarrollado en el Proyecto Fin de Master de Alejandro Hernández. En este capítulo de conclusiones puede ser interesante comentar alguno de los aportes del componente *demoAR*, desarrollado en este proyecto, con respecto a la aplicación de Alejandro. En primer lugar se ha implementado un nuevo algoritmo de autolocalización visual, en este caso uno basado en balizas visuales. Además, se ha añadido un motor de físicas que aporta un mayor realismo ya que se puede simular que los personajes se caigan de la superficie por la que caminan, como por ejemplo de una mesa. Por último, se han simulado colisiones con elementos reales, lo que también aporta un mayor realismo y una realidad aumentada más llamativa.

6.2. Trabajos futuros

Para concluir este capítulo se presentan varias líneas de trabajo que pueden partir de este proyecto tanto para mejorar los resultados como para afrontar nuevos objetivos partiendo como base del trabajo realizado.

- *Optimización del código*

  Una posible mejora es la optimización del código desarrollado para que se pueda ejecutar en procesadores menos potentes que el utilizado en este proyecto y de esta manera llevar la aplicación que hemos visto funcionar en un PC a otro tipos de dispositivos como los móviles o las tablets.

- *Mejora del sistema de autolocalización visual*

  Otra línea de trabajo es la mejora del algoritmo de autolocalización que se ha desarrollado, mediante por ejemplo el uso de un filtro de Kalman en la fusión de estimaciones, o incluso la implementación de otro algoritmo de autolocalización más complejo que no dependa de balizas y que estime con menor error. En la introducción 1.2 de esta memoria se mencionaban diferentes técnicas de autolocalización como monoSLAM o PTAM que se podrían emplear para tener un módulo de localización que no requiera conocer previamente el escenario.

- *Desarrollo de otras aplicaciones*

  En este proyecto se ha buscado crear una aplicación de realidad aumentada que por un lado use el algoritmo de localización desarrollado, para probar su funcionamiento desde un punto de vista práctico, y por otro sea una demostración del potencial que ofrecen las librerías gráficas y de físicas empleadas en este trabajo para un campo en actual crecimiento como es el de la realidad aumentada. Un siguiente paso sería la elaboración de una aplicación que empleando las técnicas aquí expuestas cumpla con un fin algo más práctico (como alguna de las aplicaciones que se vieron en el capítulo de introducción 1.3) y que incluso se pueda llegar a comercializar.

- *Reconocimiento de los objetos reales para simulaciones físicas*

  La interacción entre objetos ficticios y reales que se ha logrado en este trabajo se basa en un conocimiento previo del escenario. Un interesante campo de avance sería que la aplicación fuera capaz de analizar la escena para formar una malla que recubra a los objetos reales o detectar los mismos, de tal forma que se pueda obtener información sobre la forma y posición de un objeto real para alimentar al simulador de físicas sin depender del conocimiento *a priori* del escenario y sus objetos.

Bibliografía

[1] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 29:2007, 2007.

[2] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. Svo: Fast semi-direct monocular visual odometry. *IEEE International Conference on Robotics and Automation (ICRA), Hong Kong*, 2014.

[3] Alejandro Hernández Cordero. Aplicación visual aplicada a la realidad aumentada. *Trabajo Fin de Máster, URJC*, 2014.

[4] G. Klein and D. Murray. Parallel tracking and mapping for small ar workspaces. In *Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on*, pages 225–234, 2007.

[5] José Alberto López Fernández. Localización de un robot con visión local. *Proyecto Fin de Carrera, URJC*, 2005.

[6] Luis Miguel López Ramos. Autolocalización en tiempo real mediante seguimiento visual monocular. *Proyecto Fin de Carrera, URJC*, 2010.

[7] Edwin Olson. AprilTag: A robust and flexible visual fiducial system. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3400–3407. IEEE, May 2011.

[8] Lawrence G. Roberts. Machine perception of three-dimensional solids. *Thesis (Ph. D.)–Massachusetts Institute of Technology, Dept. of Electrical Engineering*, 1963.

[9] José Manuel Villarán. Autolocalización de un robot industrial en interiores con balizas visuales. *Proyecto Fin de Carrera, URJC*, 2015.