Universidad Zaragoza
1542

Proyecto Fin de Carrera

Sistema para la integración de contenidos infográficos en sistemas de producción audiovisual estereoscópica

Autor: David Gascueña Ferré

Director: José Ramón Beltrán Blázquez

Ingeniería Informática

ESCUELA DE INGENIERÍA Y ARQUITECTURA

Septiembre de 2017

RESUMEN

Sistema para la integración de contenidos infográficos en sistemas de producción audiovisual estereoscópica

El objetivo de este proyecto fin de carrera es el diseño e implementación de un sistema capaz de integrar contenidos infográficos 2D/3D sobre contenido audiovisual capturado mediante técnicas estereoscópicas, tanto en tiempo real como en post-producción.

Para ello se realizará un estudio inicial del estado del arte sobre las técnicas de visión estereoscópica clásica, herramientas de realidad aumentada, motores gráficos y sistemas de renderizado.

Tras este análisis se procederá a la definición de un protocolo de comunicación para la adquisición de los parametros característicos de cámaras estereoscópicas (posicionamiento, convergencia, foco...).

A partir de aquí se realizará el diseño de la aplicación de acuerdo a los siguientes requisitos:

- Adquisición de los datos de cámara en base al protocolo definido, con el objetivo de calibrar y posicionar de forma relativa las cámaras virtuales.

- Sistema para la generación de contenido 3D y posicionamiento respecto a las cámaras virtuales.

- Integración de contenidos infográficos y señal de vídeo real para previsualización mediante técnicas estereoscópicas (anaglifo, dual, entrelazado...).

- Testeo y validación del sistema.

AGRADECIMIENTOS

A Zulema, por toda la paciencia que tuviste. Todo este proyecto es para ti.

A José Ramón Beltrán, por haber estado al otro lado de la puerta en muchos momentos.

A los profesores y compañeros de esta interminable carrera.

A mis padres, familia y amigos, por recordarme que no era ingeniero, pero también por saber que la valía no te la da un papel.

A la comunidad de Ogre3D porque todo lo que han creado es muy valioso. En particular a David Williams, Matt Williams, Trey Stout (QtOgre), Mathieu Le Ber (StereoManager). Y a muchos proyectos de código abierto que forman parte de este proyecto y de mi día a día.

Índice general

Capítulo 1

Introducción

Contenidos

Este primer capítulo trata de de dar una visión general del proyecto, sus motivaciones, objetivos y resultados. Además describe los contenidos de la presente memoria.

1.1. Ámbito del proyecto

El presente proyecto fin de carrera nació en 2010, fruto de una colaboración con 3+D Entertainment[2], una empresa de reciente creación que tenía por objeto la producción, grabación y post-producción, distribución y reproducción de contenido audiovisual utilizando tecnología estereoscópica 3D. Cabe destacar que el alumno no tenía relación contractual con la empresa más allá de la simple amistad con sus creadores.

Dicha empresa estaba desarrollando un bastidor estereoscópico motorizado que permitiría la grabación de contenido 3D. Se planteó la posibilidad de crear una aplicación capaz de adquirir las señales de vídeo y mezclarlas para conseguir una previsualización estereoscópica durante el rodaje.

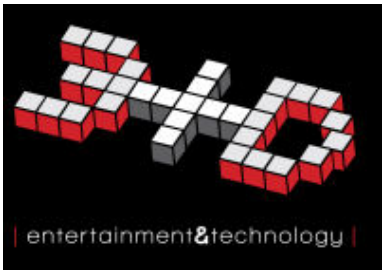


Figura 1.1: Logo de 3+D Ent.

Figura 1.2: Fabricación del bastidor VegasVision

1.2. Medios disponibles

Uno de los objetivos de la aplicación era que fuera capaz de comunicarse son un sistema de cámaras tanto para adquisición de vídeo como para el movimiento de éstas. A lo largo del proyecto se han utilizado diversos sistemas de cámaras.

VegasVision

Inicialmente se pretendía utilizar el sistema con el bastidor motorizado que se encontraba en desarrollo por 3+D Entreteinment. Este consta de dos bases sobre

raíles motorizados, donde se ubicarían dos cámaras Canon 5D MarkII. El sistema de motores está controlado por un sistema electrónico basado en Arduino y permite ajustar la separación y ángulo de las cámaras. El protocolo de comunicación por puerto serie con dicho Arduino es uno de los requisitos del presente proyecto y se encuentra especificado en el apartado 3.8.2 en la página 75.

Figura 1.3: Bastidor estereoscópico VegasVision

Figura 1.4: Caja de electrónica

Bastidor basado en webcams Logitech

En las primeras fases, debido a que el bastidor VegasVision estaba aún en desarrollo, se utilizaron dos webcams motorizadas Logitech Orbit AF, que además permitían reducir el coste de una instalación básica. Estas cámaras tienen la capacidad de rotar horizontal y verticalmente (*pan* y *tilt*), y obtienen imágenes con la suficiente resolución como para tratarlas eficazmente.

Para reducir la distancia entre lentes y facilitar la alineación de las cámaras se decidió prescindir de las bases, más anchas que las cabezas de la cámara, y que no contienen ningún elemento propio de la motorización. Así pues se elaboró un soporte de madera, que permitiría ajustar la separación de las cámaras.

Cámaras PlayStation-Eye

Pese a que el bastidor esta diseñado para utilizar cámaras de fotos de calidad profesional, fuera de las producciones audiovisuales que se realizaron se han utilizado dos cámaras PlayStation-Eye. Estás cámaras fueron diseñadas para su uso en juegos de realidad aumentada, con características muy interesantes para la visión por computador (son capaces de entregar hasta 120 imágenes por segundo) y se puede cambiar su distancia focal (zoom) de manera manual. Las cámaras están montadas sobre dos piezas impresas en 3D que anula los giros que permite

Figura 1.5: Webcam Logitech Orbit AF

la propia cámara para facilitar el ajuste estereoscópico ([24, Proyecto en thingiverse]). Ambos soportes están montados sobre un trípode flexible.

Figura 1.6: Bastidor con 2 PlayStation-Eye y piezas impresas en 3D

1.3. Objetivos del proyecto

Al inicio del proyecto se mantuvieron conversaciones con los miembros de 3+D entertainment con el fin de conocer las necesidades y recopilar una lista inicial de requisitos que la aplicación debía satisfacer.

En el ámbito de la producción audiovisual estereoscópica se requería un sistema de visualización en tiempo real de las imágenes captadas por el par de cámaras, con varios

Figura 1.7: Marcas fiduciales.

modos de mezclado para facilitar la labor del técnico de estereoscopía.

Otro de los objetivos prioritarios era reducir el tiempo necesario para ajustar del par de cámaras y de esta manera conseguir la convergencia deseada. Este proceso crucial en la preparación de cada plano resultó ser muy costoso en tiempo hasta conseguir un efecto 3D agradable para la vista. Consiste en orientar las cámaras de tal manera que converjan a un plano definido por el director de estereoscopía, proceso que establece qué elementos quedarán por delante o por detrás de la pantalla. Además un error en esta fase puede hacer que un plano sea inservible, ya que una excesiva convergencia puede ser imposible de corregir en postproducción.

Para ello se pensó utilizar procesos de visión por computador ya desarrollados, como es el seguimiento de marcas fiduciales (figura 1.7). Estas marcas, una vez analizadas, permiten al sistema conocer su posición y orientación en el espacio con respecto a la cámara. El alumno conocía la existencia de librerías de realidad aumentada, así que se decidió utilizar esta tecnología para detectar automáticamente el punto al que debían converger las cámaras. Puesto que el bastidor se diseñó para ser motorizado, se podría emplear esta posición para calcular el movimiento de los motores necesario para lograr que el plano de convergencia coincida con la marca. También se puede utilizar la información para conocer la distancia de la marca a las cámaras, extrayendo información que puede ser de utilidad para preparar el plano, como el paralaje en puntos determinados de la escena, detectando posibles divergencias en el resultado final.

También debería existir la posibilidad de mover los motores manualmente, ya sea actuando sobre los motores individualmente o de manera conjunta, modificando tanto la separación de las cámaras como su convergencia. Todo esto requería establecer un protocolo de comunicación con la electrónica. En este sentido, el alumno aportó la

información necesaria que permitiría escribir el código de los micro-controladores encargados del control de los motores.

El uso de estas marcas abría además un abanico de posibilidades en cuanto a integrar contenidos tridimensionales coherentes con el mundo real, por ejemplo, para crear contendidos infográficos en tiempo real para retransmisiones o previsualizar contenido 3D durante el rodaje de la escena, que posteriormente sería sustituido por contenido infográfico de mayor calidad en post-producción.

El sistema permitiría además ser utilizado en instalaciones interactivas de realidad aumentada donde el usuario fuera capaz de elegir el lugar de distintos objetos virtuales así como su comportamiento. A lo largo de la vida del proyecto, esta parte fue ganando peso, añadiendo requisitos cómo usabilidad, personalización del contenido asociado a las marcas en tiempo de ejecución, o integración de ruido para mejorar la coherencia final.

1.4. Resultado

El sistema implementado permite la adquisición de las señales de vídeo de una o varias cámaras, el análisis de estas para encontrar marcas fiduciales, el ajuste automatizado de la posición y orientación del par de cámaras hacia una de estas marcas y la generación de contenidos virtuales coherentes con el entorno.

Utilizado en el contexto de grabación de contenidos 3D permite al operador de cámara o realizador previsualizar el contenido adquirido con el virtual ya integrado y realizar los ajustes básicos de convergencia.

En colaboración con 3+D, se utilizó la aplicación como medio para calibrar y previsualizar las cámaras estereoscópicas en diversas producciones 3D, destacando el rodaje de *Road to Wacken 3D*, durante el verano de 2010.

Además constituye la base para el desarrollo de aplicaciones interactivas de realidad aumentada, ya sea en dos dimensiones o estereoscópicas, pudiendo utilizar cámaras auxiliares para aumentar la fiabilidad del sistema.

La aplicación resultante ha sido utilizada con éxito en diversos proyectos. Ha sido la base para varias instalaciones interactivas, destacando *Virtual Graffiti*, en Festival Internacional de Arte Urbano de Zaragoza ($6^{\underline{o}}$ asalto), un sistema interactivo de realidad aumentada con el que los asistentes pudieron pintar virtualmente un edificio de la ciudad.

Ha sido también utilizado en diversos talleres y actividades con niños, labor a la que se dedica actualmente el alumno.

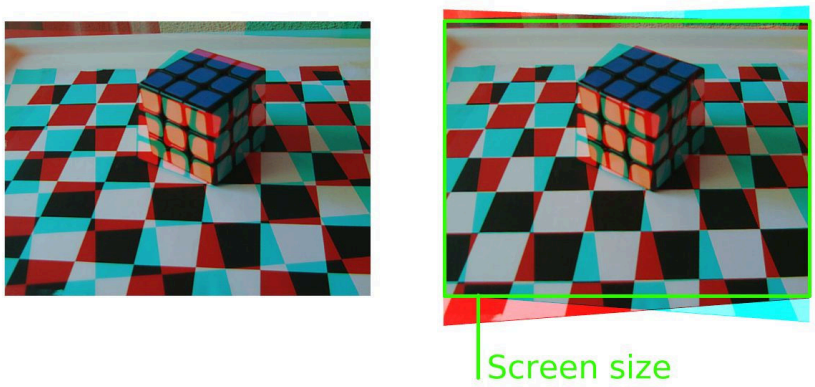
Figura 1.8: Virtual Graffiti

Figura 1.9: Road to Wacken 3D

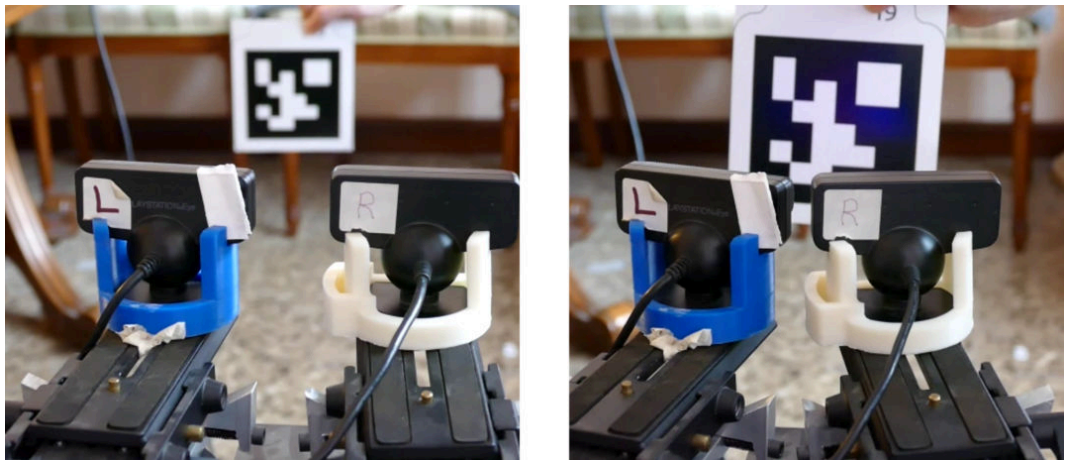
1.4.1. Listado de las características principales

- Adquisición de varias cámaras y mezclado en diversos modos estereoscópicos (apartado 3.2.1 en la página 53)

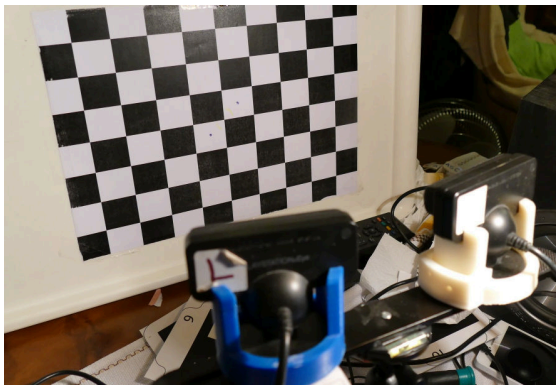
- Corrección manual de la estereoscopía: ajuste vertical u horizontal, corrección del efecto keystone o incorporación de márgenes laterales (apartado 3.5 en la página 63).



- Búsqueda de marcas fiduciales y sobreimpresión de contenido sintético sobre la imagen.
- Movimiento del bastidor motorizado y convergencia automática de las cámaras(apartado 3.8 en la página 74 ,apartado 3.1.2 en la página 42).



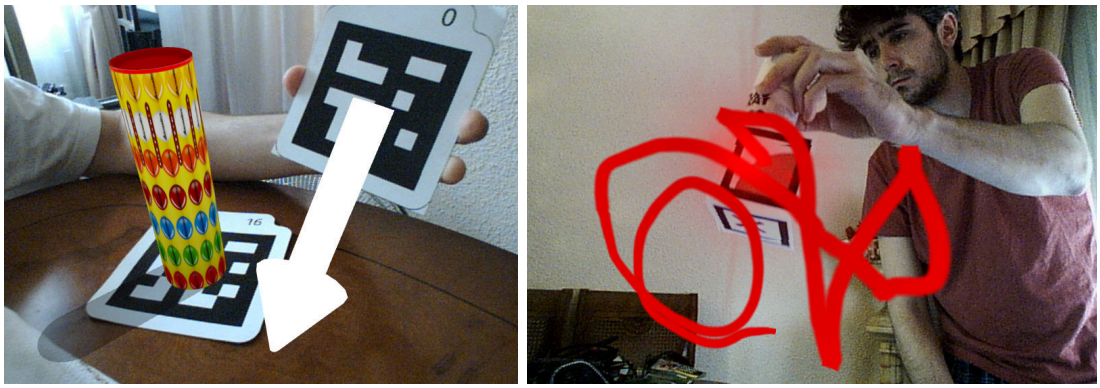
- Calibración para calcular la posición de las cámaras derecha o auxiliares con respecto a la cámara izquierda.



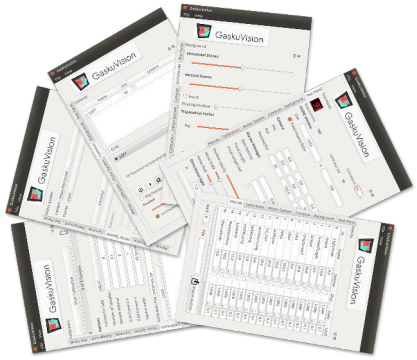
- Implementación de sombras y objetos virtuales que actúan como receptores de sombras para objetos reales.

- Integración de ruido sobre el contenido sintético para mejorar la coherencia final.

- Implementación de diversos tipos de marcas interactivas.



- Interfaz modular en ventana independiente.



1.5. Fases del proyecto

El proyecto ha pasado por diversas fases a lo largo de su desarrollo:

Estudio de las bases teóricas y elección de subsistemas

Estudio de la teoría de estereoscopía clásica y ampliación de los conocimientos sobre geometría proyectiva y generación de contenidos 3D. Además se hizo un estudio del estado del arte de diversas librerías necesarias, con el fin de elegir los paquetes de *software* a integrar en el sistema.

Implementación inicial orientada a la producción estereoscópica

En esta implementación inicial, orientada a las diversas producciones estereoscópicas en que colaboró el alumno, predominó la gestión de las imágenes, modos de mezclado y la comunicación con el bastidor.

Desarrollo del proyecto como framework para aplicaciónes interactivas de realidad aumentada

En esta fase primó más la parte de realidad aumentada, desarrollo de tipos de marcas que dotaran de posibilidades a las aplicaciones finales y rendimiento general de la aplicación.

1.6. Estructura de la memoria

Para la redacción de la memoria se ha empleado el sistema de composición de textos LaTeX [17]. Como editor se ha utilizado Kile y TexStudio[16], sobre Ubuntu 16.10.

Memoria principal

El presente documento se divide en los siguientes capítulos:

1. Introducción
Capítulo que ofrece una visión general del proyecto realizado.

2. Diseño y componentes externos
Expone el proceso de análisis de requisitos, diseño de la aplicación e integración de las librerías externas utilizadas.

3. Implementación y descripción del programa
Detalle de la implementación realizada y exposición del funcionamiento del programa.

4. Conclusiones y posibles mejoras
Capítulo dedicado a valorar el desarrollo y posibles aplicaciones del proyecto, así como de dar una perspectiva de hacia donde podría ampliarse.

Anexos

Adicionalmente se añaden diversos anexos, que por su temática más específica, no tienen cabida en la estructura principal:

A. Estereoscopía: teoría y estudio del arte
Anexo que sirve como aproximación a la estereoscopía clásica, los diversos métodos de grabación y reproducción, así como algunos problemas asociados a las técnicas estereoscópicas. Además se incluye información relativa a la teoría de geometría proyectiva relevante al proyecto.

B. Motor gráfico: comparativa y elección de Ogre3D
En este anexo se realiza una comparativa entre los diversos motores gráficos analizados, se exponen las razones de la elección de Ogre3D.

C. Entregables: compilación y configuración de la aplicación
En este anexo se recopila la información útil para compilar el proyecto y configurar y ejecutar las distintas aplicaciones.

Capítulo 2

Diseño y componentes externos

Contenidos

Expone el el establecimiento de los requisitos, diseño inicial y describe los componentes y librerías externas utilizadas.

2.1. Requisitos del sistema

En esta fase inicial, y sin el conocimiento necesario para abordar un análisis de requisitos más detallado, simplemente se trató de describir el funcionamiento deseado del sistema, así como establecer los elementos básicos que lo formarían.

- **R01:** Se debe priorizar el uso de herramientas y librerías de código abierto, y en la medida de lo posible gratuitas.
- **R02**: El objetivo es lograr un sistema multiplataforma, al menos soportando Windows y Linux.
- **R03**: El lenguaje debía ser C++ por familiaridad, cantidad de librerías disponibles y eficiencia.
- **R04**: El sistema será capaz de adquirir la señal de diversas cámaras.
- **R05**: Debe ser capaz de detectar diversas marcas fiduciales y deducir su posición y orientación con respecto a las cámaras.
- **R06**: Se debe poder modificar la posición de las cámaras reales, ya sea manualmente o por medio de alguna marca fiducial.
- **R07**: Debe ser posible sobreimpresionar contenidos 3D sobre la señal adquirida, y establecer su posición mediante marcas.
- **R08**: Debe implementarse un sistema de sombras sobre planos transparentes para integrarlos en la imagen final.
- **R09**: Se debe poder escoger entre distintos tipos de visualización tanto 2D como estereoscópica (anaglifo en sus distintos modos, entrelazado, paralelo, etc...)

2.2. Componentes de la aplicación

Tras concretar las funciones que debía satisfacer la aplicación desarrollada, se efectuó una comparativa de las librerías necesarias para cada funcionalidad. La aplicación debía hacer uso de un motor gráfico para mostrar el contenido y una librería de visión por

computador para el análisis de las imágenes. Posteriormente se decidió el uso de un framework para la interfaz que que complementara al motor gráfico. Para tener una visión completa de las partes de las que consta la aplicación se muestra su diagrama de componentes (figura 2.1).

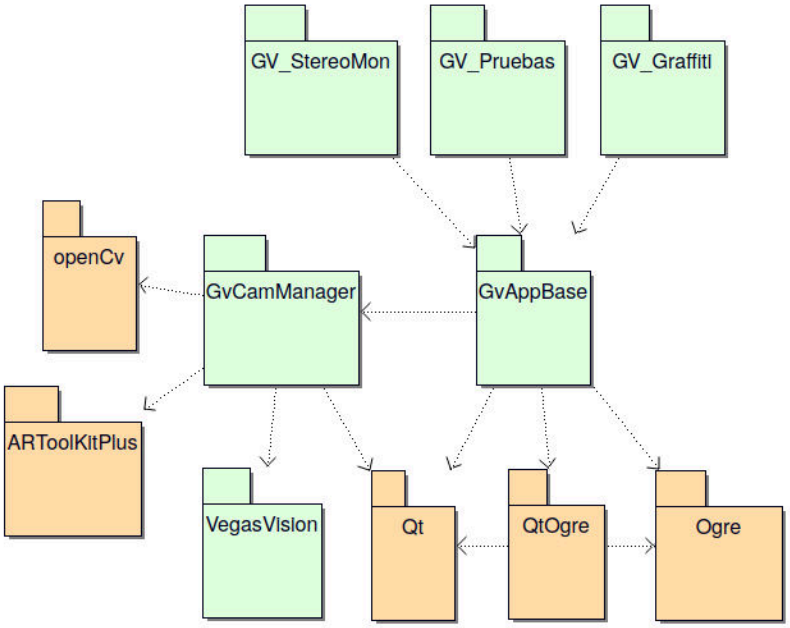


Figura 2.1: Diagrama de componentes

Se muestran en color verde los módulos implementados completamente por el alumno y en naranja las librerías externas utilizadas.

El módulo principal **GvAppBase** contiene todo lo necesario para crear una aplicación mínima. De este componente dependen las aplicaciones finales que se quieran implementar. Se entregan tres ejemplos:

- **Gv_StereoMon** sirve como herramienta en una prueba en una producción estereoscópica y no contiene ninguna funcionalidad relacionada con la realidad aumentada.
- **Gv_Graffiti** es una aplicación interactiva que permite dibujar trazos de colores sobre la imagen o sobre imágenes predefinidas.
- **Gv_Pruebas** contiene ejemplos del resto de funcionalidades que se pueden aplicar a las marcas.

Todas las aplicaciones dependen de cuatro librerías externas para su mínimo funcionamiento. **Ogre** es el motor gráfico encargado de presentar las imágenes obtenidas con el contenido virtual sobreimpresionado. **Qt** es la librería encargada de gestionar la interfaz y proporcionar una gran cantidad de tipos y clases. Para la integración de estas dos últimas librerías se contó con la ayuda de **QtOgre**, desarrollada por la comunidad, que facilita la integración de ambas. Para el análisis de los fotogramas se ha usado **ARToolKit**, y posteriormente se utilizó **openCV** para la calibración de las cámaras.

La aplicación consta de varios módulos en forma de librerías estáticas que conforman la base del sistema. Cada módulo implementa además los *widgets* o partes de la interfaz que gobiernan su comportamiento. Así podemos distinguir los siguientes subsistemas:

- **Módulo GvCamManager:** este módulo pone en funcionamiento las diversas cámaras, ejecuta el análisis de los fotogramas, realiza las conversiones entre tipos de datos (ARToolKit a Ogre3D) y permite al programa principal asociar objetos 3D a distintas marcas fiduciales. Además permite analizar los datos obtenidos para ordenar el movimiento de las cámaras para establecer la convergencia o calibrar cámaras auxiliares para mejorar la robustez del análisis de marcas.
- **Módulo SistemaMotores:** es la interfaz para manejar el hardware, permitiendo modificar la separación y la convergencia de las cámaras para su ajuste. Es el encargado de la comunicación con la electrónica por puerto serie.
- **Módulo QtOgre:** integra Ogre dentro de la estructura Qt.

2.3. Motor gráfico: Ogre3D

Esta sección expone el funcionamiento de la libraría gráfica Ogre3D. Se detallan los aspectos más generales que atañen al diseño de la aplicación. El apéndice B en la página 105 contiene la comparativa realizada entre las varias alternativas que se valoraron y una visión más detallada de la librería.

Uno de los pilares de la aplicación es obviamente el motor gráfico. Existen una amplia variedad de opciones a considerar. El principal requisito fue que debía ser de código abierto y multiplataforma. Ogre3D [19] es un motor de renderizado gráfico, con una interfaz orientada a objetos diseñada para ser independiente del subsistema gráfico subyacente (OpenGL, DirectX...). Como puntos a favor destacan:

- Licencia de código abierto MIT.

- Buen diseño con una amplia documentación disponible.
- Multiplataforma (soporte para Windows, Linux y Mac OSX).
- Lenguaje C++ (Bindings a Python disponibles).
- Potente declaración de materiales en archivos externos, que permiten mantener los recursos de la aplicación fuera del código.
- Uso de shaders (Cg, HLSL o GLSL), compositores de escena, sistemas de partículas, etc...
- Uso de plugins para añadir funcionalidades adicionales.
- Gran comunidad que facilita la resolución de dudas.

Una característica atractiva como desarrollador, es el buen diseño orientado a objetos que presenta, algo no demasiado común en motores gráficos, que suelen usar C para mejorar el rendimiento. Ogre3D hace un buen uso de conceptos como encapsulamiento, abstracción, polimorfismo, además de emplear diversos patrones de diseño (figura 2.2 en la página siguiente).

Una vez elegido el motor gráfico, el alumno realizó varias pruebas con el fin de familiarizarse con su arquitectura, los *scripts* que definen los materiales y el entorno de desarrollo.

Clases principales

Se puede ver una vista de las clases más importantes en la figura 2.2 en la página siguiente. En el nivel superior se encuentra la clase **Root** que sirve para inicializar el motor, configurarlo y ejecutar los bucles principales. A través de esta clase se instancian los componentes principales del motor, a los cuales se puede acceder en cualquier momento sin necesidad de mantener referencias ya que usan el patrón singleton.

El resto de clases se pueden englobar en tres grupos según su función: manejo de la escena, manejo de los recursos y el renderizado.

SceneManager Es la clase encargada de gestionar los contenidos de la escena (cámaras, objetos, luces y materiales) y se comunica con el sistema de renderizado para enviarle los elementos que deben ser mostrados en pantalla. Todos los elementos visibles que se muestran por pantalla son creados a través de esta clase. Cada objeto está asociado a una entidad (*Entity*), clase encargada de gestionar las animaciones y los materiales. Para ser visible, una entidad tiene que

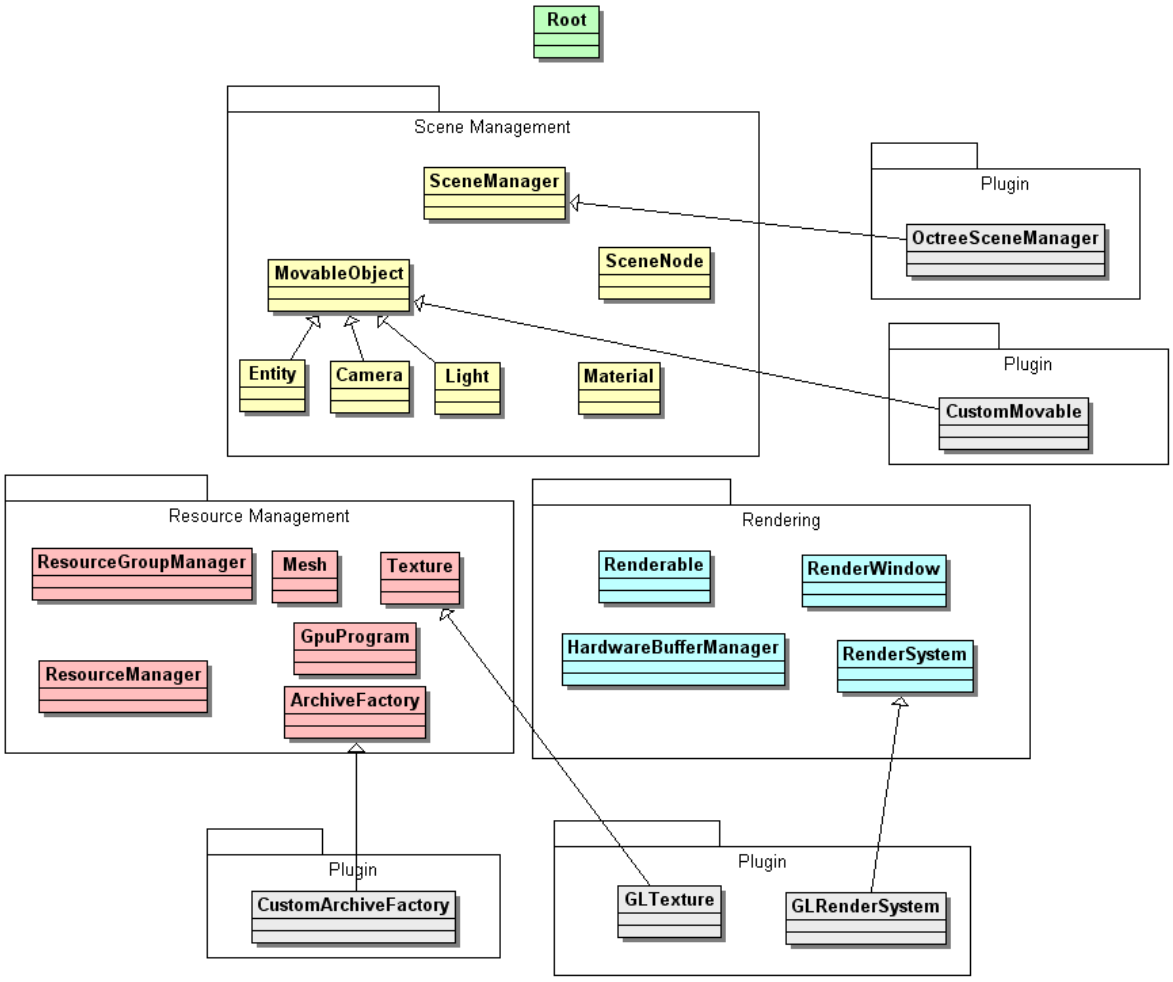


Figura 2.2: Diagrama UML de Ogre3D [19]

estar asociada a un nodo de la escena (*SceneNode*), que tiene una posición determinada en la escena, y a través de esta clase se interactúa para situar el objeto en cualquier lugar (figura 2.3 en la página siguiente). En el presente proyecto, a cada marca se le asocia a un nodo principal de Ogre, del que descuelgan el resto de nodos que contienen el objeto, receptores de sombras, información de depuración, etc...

ResourceGroupManager Esta clase tiene la función de cargar los grupos de recursos que utiliza la aplicación, de manera que se asegure que son cargados una sola vez y compartidos de manera eficiente por el resto del componentes del motor. Estos grupos pueden ser cargados y descargados de memoria a voluntad del programador.

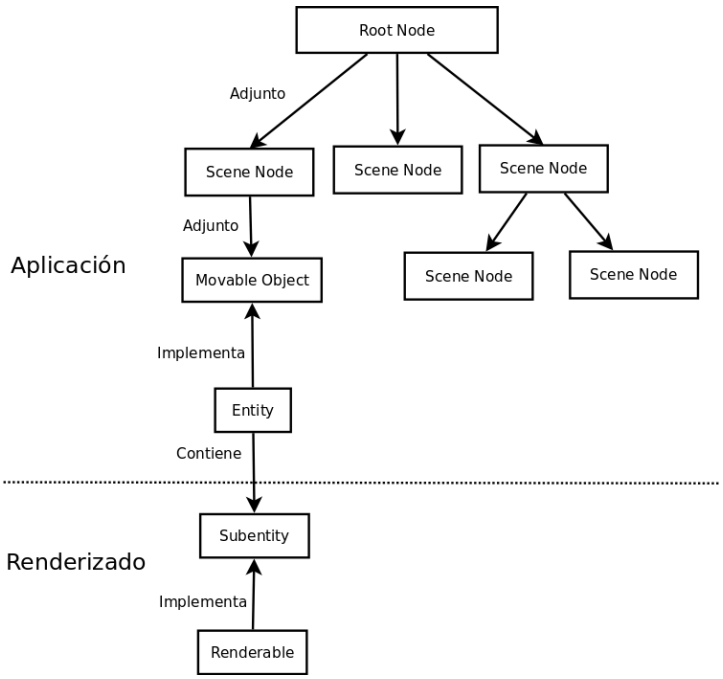


Figura 2.3: Esquema del grafo de escena de Ogre [13]

Trabaja con otras clases subyacentes que permiten manejar los distintos tipos de recursos como son las texturas (**TextureManager**) o las mallas (**MeshManager**). El recurso más importante que se gestiona en la aplicación son las texturas de la cámara, ya que deben ser accedidas por el módulo que gestiona las cámaras en escritura y por el gestor de fondos en lectura, siempre en exclusión mutua.

RenderManager Es una clase abstracta que define una interfaz común a las distintas API de renderizado (Direct3D, OpenGL...). El programador no suele tener la necesidad de acceder a ella una vez inicializada. Pese a que la renderización se realiza automáticamente para todos los objetos que forman parte del árbol de nodos de Ogre, se hace un uso avanzado en ciertas áreas de la aplicación, por ejemplo en los cambios de posición de los objetos (izquierda/derecha) cuando se desea un ajuste perfecto de estos en ambas vistas (apartado 3.2 en la página 51).

2.4. Librería Qt

Pese a que Ogre contiene una librería para la creación de interfaces gráficas básicas, pronto se decidió que ésta no era la adecuada para crear la interfaz que el programa requería. La iterfaz más utilizada de Ogre3D, **CEGUI** (figura 2.4 en la página siguiente),

está más orientada al diseño de interfaces para juegos, sobreimprimiendo los distintos elementos sobre la ventana principal, con un diseño que resultaba poco adecuado.

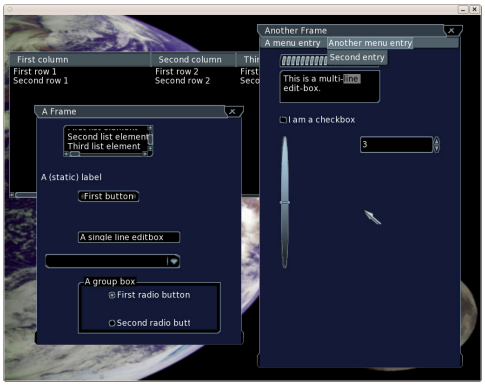


Figura 2.4: Interfaz CEGUI

Se decidió utilizar la librería Qt[21], con la que ya se había trabajado. Qt es un *framework* de desarrollo multiplataforma y de código abierto desarrollado por Nokia. Está escrita en C++, pero hace uso de un preprocesador (llamado *Meta Object Compiler,* o moc) que dota al lenguaje de un sistema de señales y slots, que facilita el diseño eficiente de la interfaz siguiendo el patrón modelo/vista.

Cada clase pues definir unos nuevos tipos de métodos llamados ranuras (*slots*) y señales (*signals*), como se muestra en la figura 2.5 en la página siguiente. La función de Qt connect asocia una señal a una ranura. A partir de ese momento, cada vez que se emita dicha señal se ejecutará la ranura asociada. Esta funcionalidad facilita la labor de diseño y reduce la interacción directa entre clases. Un ejemplo es la interacción entre las cámaras y el resto de la aplicación. Cuando la clase encargada del análisis de las imágenes, función que se realiza en un hilo independiente, termina de analizar un fotograma, emite una señal. Es la clase superior *GvCamManager*, la que decide qué efectos tendrá. Puede provocar la copia de la imagen a texturas que más tarde se renderizarán, la actualización de nodos asociados a las marcas analizadas, o el refresco del elemento de la interfaz que muestra los cuadros por segundo de dicha cámara.

Este proyecto utiliza Qt para una gran cantidad de tareas que van más allá de la propia interfaz:

- Entradas de ratón y teclado.
- Comunicación de las distintas clases a través de señales y slots.
- Gestión de hilos y acceso en exclusión mutua a datos compartidos.
- Lectura y escritura de archivos de configuración.

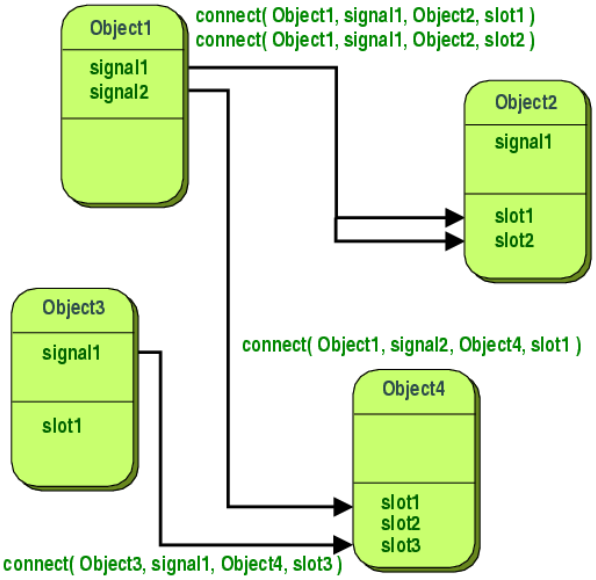


Figura 2.5: Mecanismo de señales (signals) y ranuras (slots)

- Estructuras de datos.
- Temporizadores.
- Gestión y comunicación por puerto serie.

2.5. Integración Qt-Ogre

Para la integración de Qt con Ogre se utilizó el framework QtOgre, un add-on para Ogre3D que facilita la inicialización de una ventana Ogre en el bucle principal de Qt. Implementa también métodos para cargar escenas completas de objetos, utilizando para ello archivos XML.

Entre los cambios implementados destacan:

Multimedia Timers: Implementación para Windows de *Multimedia Timers*, para mejorar la precisión de la tasa de refersco de la ventana de Ogre (apartado 2.5.1 en la página 26).

Pantalla de bienvenida: Se añadió el logo de la aplicación al inicio, ya que la carga se puede llegar a demorar varios segundos, según el número de cámaras.

Manejo de la relación de aspecto: Debido a que la imagen de la cámara no acepta cambios en la relación de aspecto, se tuvo que implementar el algoritmo que modifica la ventana de visualización (*viewport*) de Ogre, añadiendo bandas negras para adaptarlo a la resolución y relación de aspecto de la resolución de las cámaras físicas (figura 2.6);

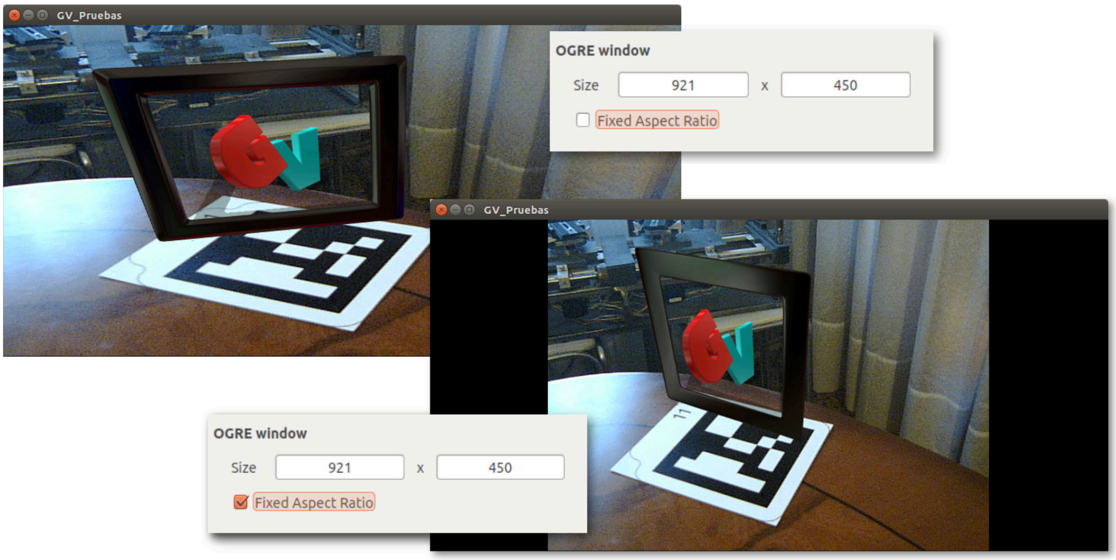


Figura 2.6: Opción para mantener la relación de aspecto

Unificación de los archivos de configuración: Todos los archivos de configuración se han unificado, de tal manera que cada aplicación final tenga su propio archivo con todos los parámetros necesarios.

Todos estos cambios se encuentran en el archivo */gaskuvision/patches/QtOgre.patch* del proyecto, del que se muestran a continuación las estadísticas.

Listing 2.1: Modificaciones del framework QtOgre

```
$ diffstat QtOgre.patch
include/Application.h  |   39 ++++++++-
include/OgreWidget.h   |   12 ++-
source/Application.cpp |  196 +++++++++++++++++++++++++++++++++++++-----------
source/OgreWidget.cpp  |  103 +++++++++++++++++++------
4 files changed, 274 insertions(+), 76 deletions(-)
```

2.5.1. Intervalo de refresco del bucle gráfico

Debido a que Ogre se encuentra embebido dentro de un *widget* de Qt no se refresca automáticamente, y es Qt el encargado de forzar la actualización del motor gráfico. Se utilizó inicialmente el tipo *QTimer* de la librería Qt, un reloj que permite realizar tareas periódicas. La aplicación debería permitir establecer el intervalo de refresco y con ello la tasa de fotogramas por segundo del motor gráfico. Cada vez que el temporizador llega a cero se da la orden de refrescar la escena gráfica. Tras la implementación inicial se efectuaron medidas de la tasa de refresco que se incluyen en la siguiente tabla (cuadro 2.1 en la página siguiente). Las columnas indican:

- El tiempo de refresco en milisegundos, que podemos variar desde la interfaz en tiempo de ejecución.
- El tiempo transcurrido entre bucles, en milisegundos, que debería coincidir con el tiempo introducido.
- Los fotogramas por segundo esperados (1/IntervaloEsperado)
- Los fotogramas por segundo resultantes, que deberían coincidir con los esperados.

Se puede observar que hasta 20 milisegundos, la aplicación funciona a la tasa de refresco esperada, exceptuando los intervalos más bajos (resaltados en amarillo), debido a que la aplicación se encuentra limitada por CPU.

Sin embargo, a partir de 21 milisegundos, la tasa de refresco obtenida no coincide con la tasa esperada (resaltado en rojo), excepto en unos valores concretos. La razón de este comportamiento es la granularidad o resolución de los relojes de sistema de Windows, que es de **15.625 ms**. Por tanto se obtienen tiempos transcurridos múltiplos de este valor (31, 46, 62, 78, ...). Esto lleva las tasas de refresco a unos valores discretos que difieren de los esperados, e impide a la aplicación funcionar a tasas típicas como 24fps, 25fps o 30fps.

La razón de que por debajo de 20 milisegundos la resolución sea de un milisegundo es la implementación interna de *QTimer*. Si se le pide un intervalo menor de 20ms, Qt usa relojes multimedia en vez de los relojes Win32 del sistema, lo que conlleva un aumento del gasto de CPU. Esto se nota en la caida de gasto de procesador al pasar de 20 a 21 ms. Aunque este comportamiento no esta documentado, se encontro esta línea en log de cambios de QT:

Listing 2.2: Qt Changelog

```
QEventDispatcherWin32 (internal class):
* Changed the threshold for using multimedia timers to 20ms (was 10ms).
```

Intervalo esperado	Tiempo transcurrido	FPS esperados	FPS medidos
0	1	-	815
1	2	1000	455
2	2	500	432
3	3	333	333
4	4	250	250
5	5	200	200
6	6	166	166
..	..	..	..
19	19	52	52
20	20	50	50
21	31 (CPU 0 %)	47	32
..	31	..	32
31	31	32	32
32	39	31	25
33	46	30	21
..	46	..	21
46	46	21	21
47	52	21	19
48	62	20	16
..	62	..	16
62	62	16	16
..	..	..	..

Cuadro 2.1: Medidas de tiempos del bucle gráfico

Para poder alcanzar cualquier tasa de refresco, se implementó el bucle de refresco de tal manera que se pudiera elegir entre el uso por defecto de *QTimer* o la implementación con un reloj multimedia (*mmsystem.h*), con mejor resolución, pero con más gasto de CPU. De esta manera, en el caso de querer una tasa de refresco concreta, basta con compilar la aplicación definiendo la variable **_MMTIMER**. Cabe observar que en Linux no existe este problema.

2.6. Visualización de la señal estereoscópica

Para la visualización estereoscópica se requería implementar un método capaz de renderizar la escena desde dos puntos de vista distintos, uno por cada cámara, y

posteriormente un sistema de mezclado con distintos modos de visualización. En los foros de Ogre se encontró el módulo *StereoManager*, que permitía ejecutar las aplicaciónes en modo estereoscópico, pero tenía limitaciones derivadas de las particularidades propias del presente proyecto.

- **Limitación 1**: Nuestra aplicación debía ser capaz de utilizar una imagen de fondo distinta para cada cámara. Esto requiere renderizar un plano distinto desde cada punto de vista.

- **Limitación 2**: Nuestras cámaras virtuales deben utilizar matrices de proyección personalizadas con el fin de incorporar la información de los parámetros intrínsecos calculados en la calibración de las cámaras físicas.

- **Limitación 3**: Era necesario incorporar nuevos modos de mezclado estereoscópico con tal de satisfacer los requisitos (apartado 3.2.1 en la página 53)

Para solventar estas limitaciones, y utilizando dicho código como base, se modificaron y añadieron los métodos necesarios.

Para solventar el punto 1, se buscó en la API de Ogre algúna funcionalidad que permitiera ocultar objetos para un determinado punto de vista, maneniéndolo en el resto. Esto se consigue utilizando máscaras de visibilidad, que permiten decidir qué objetos se muestran en cada punto de vista.

Con respecto a los puntos 2 y 3, se añadieron a la clase *StereoManager* los métodos necesarios para pasar las matrices de proyección propias de las cámaras, en lugar de usar los parámetros de cámara propios del motor. Además permite pasar una matriz de transformación entre las dos cámaras estéreo en lugar de utilizar un parámetro de separación y distancia al plano focal. Para soportar los nuevos modos de renderizado estereoscópico se añadieron los métodos necesarios, además de crear nuevos materiales y shaders. Todos los cambios se listan en el archivo *gaskuvision/patches/StereoManager.patch* del que se muestran a continuación las estadísticas.

Listing 2.3: Modificaciones del módulo StereoManager

```
$ diffstat StereoManager.patch
StereoManager.cpp      |  148 ++++++++++++++++++++++++++++++++++++++--------
StereoManager.h        |   35 +++++++++
Stereoscopy.cg         |   82 ++++++++++++++++++++++
Stereoscopy.compositor |  123 ++++++++++++++++++++++++++++++++---
Stereoscopy.material   |  166 ++++++++++++++++++++++++++++++++++++++++++++++++
5 files changed, 514 insertions(+), 40 deletions(-)
```

2.7. Seguimiento de marcas con ArToolKitPlus

La inclusión de contenido virtual en la imagen real implicaba que el posicionamiento de las cámaras virtuales en la escena virtual fuera coherente con la posición de las cámaras reales con respecto al entorno.

Se realizó un estudio del arte de las tecnologías actuales de seguimiento de marcas fiduciales. Existen una variedad de soluciones software, aunque muchas de ellas se basan en la librería ARToolKit, que en el momento de iniciar el proyecto, ya no se seguía desarrollando.

Limitando por el lenguaje utilizado (C++) y la licencia usada, se redujeron las alternativas disponibles a las siguientes:

- ARToolKit: versión original de la librería, escrita en C, que permite el seguimiento de marcas fiduciales usando técnicas de visión por computador.
- ARToolKitPlus: desarrollada a partir de la anterior, pero con un enfoque orientado a objetos (C++), multiplataforma, cumplía con los requisitos de la aplicación. Además ofrece mejoras de rendimiento y nuevas funcionalidades, como el uso de marcas codificadas BCH.

ARToolKitPlus es una librería software que puede ser utilizada para calcular la posición y orientación de una cámara en relación a marcas físicas en tiempo real. Esto permite el desarrollo de una gran variedad de aplicaciones de realidad aumentada. ARToolKitPlus es una versión extendida de ARToolKit, y desarrollada a partir del código de visión por computador de esta última que añade funcionalidades, pero rompe la compatibilidad debido a un nuevo diseño basado en clases.

Mejoras con respecto a ARToolKit (version 2.1.1):

- API orientada a objetos (parametrización en tiempo de compilación por medio de plantillas)
- Hasta 4096 marcas basadas en Id sin penalización de velocidad por el número de marcas utilizadas
- Nuevos formatos de imagen soportados (RGB565, Gray)
- Anchura variable del borde de las marcas
- Nuevos algoritmos de estimación de pose que mejoran la estabilidad del seguimiento
- Implementación del algoritmo Robust Planar Pose"por G. Schweighofer y A. Pinz

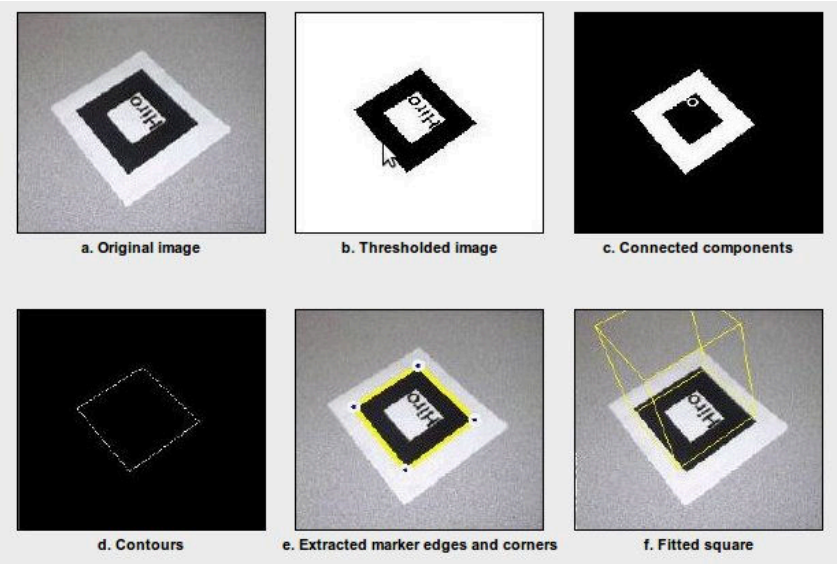


Figura 2.7: Proceso de identificación de marcas de ARToolKit

- Carga de archivos de la toolbox de MATLAB para calibración de camaras
- Umbralización automatica

El objetivo de la librería es la búsqueda de marcas cuadrangulares en una imagen (figura 2.7). Tras un proceso de binarización se buscan distintas rectas así como los puntos en los que interseccionan. De ahí se puede estimar qué conjunto de rectas corresponden a marcas reales. Una vez encontrada una marca se puede extraer la información interior que contiene, tomando muestras en distintas áreas el cuadrilátero interior. Esa información se utiliza para identificar a la marca con un número entero.

Existen dos tipos de marcas soportadas.

- La primera consiste en un código binario interno (**BCH**), similar a los códigos QR, que puede identificar la marca usando el código 6x6 de la marca. Es capaz de identificar 4096 marcas distintas.
- El segundo tipo se basa en **patrones** ya conocidos que son cargados en el inicio de aplicación. Esto permite crear marcas personalizadas en blanco y negro o en color y asociarla a una marca. Como característica adicional a la librería se ha implementado la capacidad de guardar una marca presentada ante la cámara.

El proyecto extiende todas las características de ARToolKit y las pone a disposición de la aplicación final. Todos los parámetros que se pueden personalizar son accesibles desde los archivos de configuración.

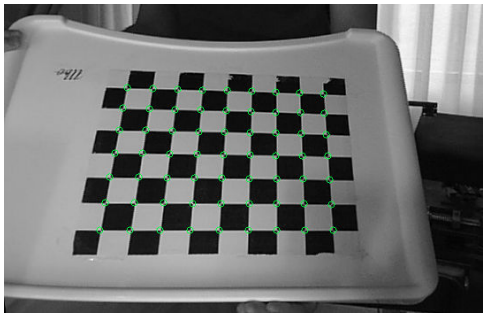


Figura 2.8: Cuadrícula de calibración.

2.8. Calibración con Open CV

Para conocer la posición relativa de las distintas cámaras con respecto a la cámara izquierda, que se toma como referencia, se utiliza openCV a través del módulo Qt openCvStereoVision [23]. OpenCV es una librería de visión por computador de código abierto. Contiene cientos de algoritmos que pueden ser usados para multitud de aplicaciones: reconocimiento facial, identificación de objetos, tratamiento de imágenes, etc.

El primer paso es adquirir suficientes tomas (entre 10 y 20 por cámara) de una carta de calibración consistente en una cuadrícula (figura 2.8). Tras ello somos capaces de extraer todos los vértices interiores (función *cvFindChessboardCorners*) y calcular su posición en pantalla con precisión de sub-pixel (*cvFindCornerSubPix*). Finalmente la función *cvStereoCalibrate* es capaz de estimar la transformación entre dos cámaras, fijas una con respecto a la otra.

Como resultado de esta función obtenemos la matriz esencial, una matriz de rotación y un vector de traslación, que nos permite situar las diversas cámaras virtuales que verán la escena 3D desde el punto de vista correspondiente a la cámara real (figura 2.9 en la página siguiente).

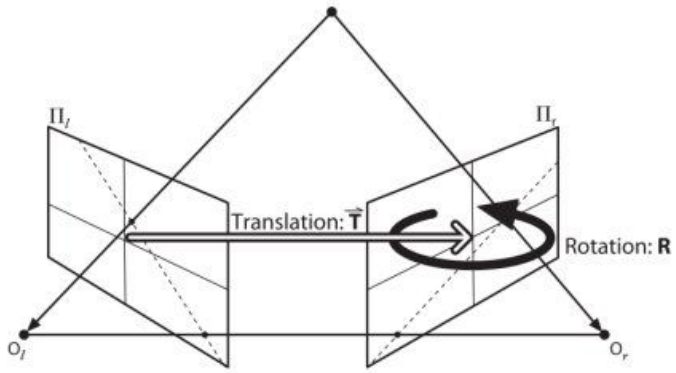


Figura 2.9: Pose de la cámara derecha a partir de R y t

2.9. Adquisición de las imágenes de cámara

La aplicación debía ser capaz de recibir la señal capturada por una o varias cámaras. La idea inicial fue utilizar el SDK de Canon[8], que permite el control de las funciones de las cámaras **Canon 5D MarkII**. Pero debido a sus restricciones en cuanto licencia y plataforma de desarrollo, se descartó.

Finalmente se optó por usar capturadoras USB de vídeo que permiten adquirir las imágenes de cámara de la misma manera que una cámara web. A lo largo del desarrollo se ha añadido soporte para diversas librerías, dependiendo entre otras cosas, del sistema operativo:

- **VideoInput** [27]: una librería para Windows que permite adquirir de manera sencilla las imágenes de varias cámaras o capturadoras. Consiste en un *wrapper* sobre *DirectShow*. Permite además controlar distintas propiedades de éstas, tanto de imagen (brillo, contraste, saturación, etc...) como de movimiento (pan, tilt, zoom...), mediante las propiedades definidas en la API de DirectShow. Una característica importante es la posibilidad de realizar la captura mediante una llamada bloqueante o no bloqueante.

- **CL Eye (Code Laboratories)** [4]: Una librería de código privativo, pero la única que permitía adquirir la imagen de las cámaras PS Eye a 120 fotogramas por segundo.

- **ARToolKit**: A través de la librería de video de ARToolKit, que permite utilizar tanto Video4Linux2 como GStreamer, las dos librerías de vídeo principales en Linux.

Para el control de los parámetros de cámara (brillo, contraste, etc...) se utiliza el programa **GTK+ UVC Viewer** [12], un programa de Linux. Se ejecuta desde la aplicación como proceso externo, utilizando la opción - *-control_panel* que permite utilizar solo las características de control de cámara (figura 2.10).

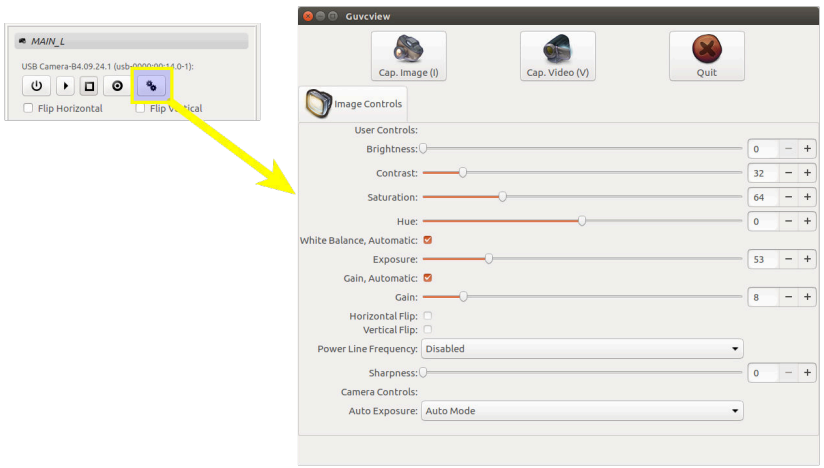


Figura 2.10: Acceso e interfaz de Guvcview

Capítulo 3

Implementación y descripción del programa

Contenidos

En este capítulo se detallan todas las características implementadas, agrupadas en los distintos módulos que forman parte de la solución del proyecto.

3.1. GvCommonLogic: la clase principal

La clase **GvCommonLogic** contiene gran parte de la lógica del programa principal. Está diseñado de manera que funcione como un *framework* y cada aplicación que se desarrolle derive de esta clase. La clase **GvLogic**, derivada de **GvCommonLogic**, contiene las características particulares que requiera cada aplicación. **GvCommonLogic** a su vez, deriva de la clase GameLogic (del módulo QtOgre, que tan sólo declara las funciones virtuales que necesitan ser implementadas para que el bucle principal de Qt pueda llamarlas. Se puede observar una vista general del diagrama de clases en la figura 3.1.

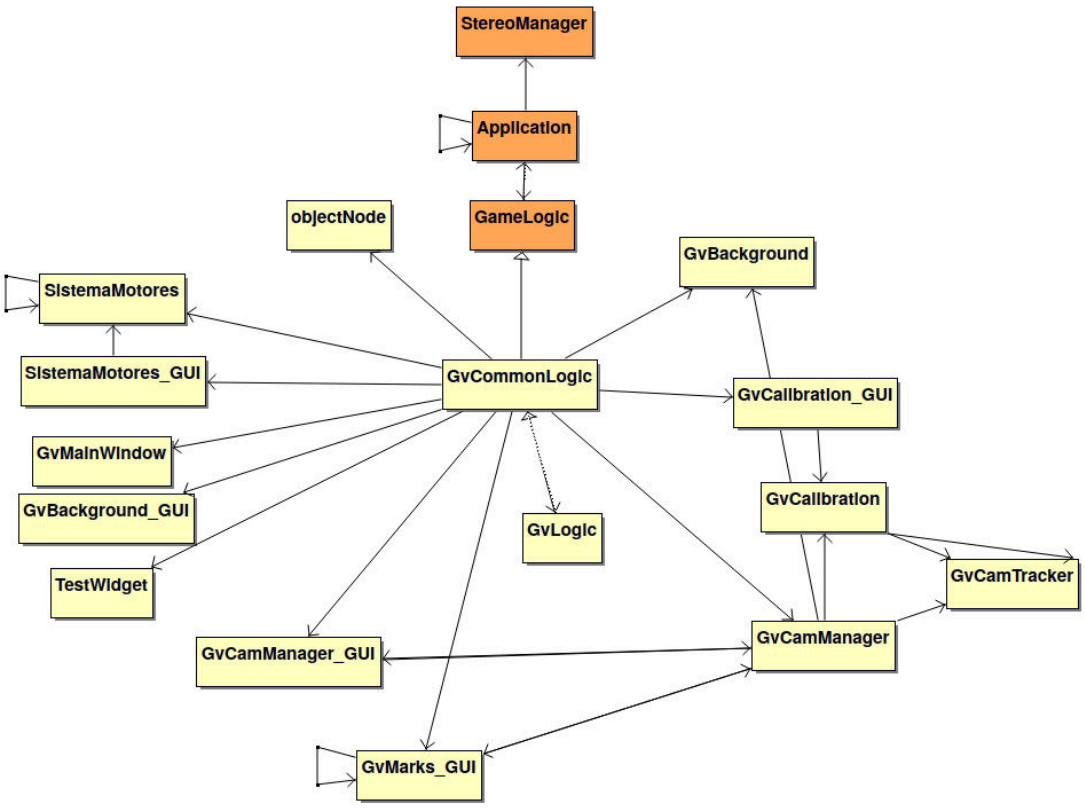


Figura 3.1: Vista general del diseño.

GvCommonLogic es la encargada de instanciar todas las ventanas que forman la

interfaz, e inicializar todos los módulos requeridos. Toda la lógica propia del programa, como qué marcas serán detectadas, objetos que se mostrarán, etc... está contenida en GvLogic.

3.1.1. Aplicación principal

Al iniciar la aplicación se muestra el logo en una pantalla de bienvenida mientras se cargan todos los archivos necesarios y se configuran las cámaras. Este proceso suele durar unos segundos (figura 3.2).

Figura 3.2: Logo de bienvenida

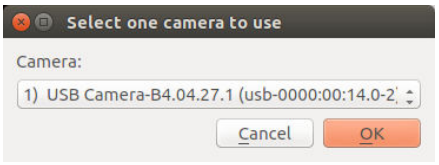


Figura 3.3: Diálogo de selección de cámara.

Si las cámaras no están configuradas por defecto en los archivos de configuración aparecerá una ventana donde podemos seleccionar la cámara principal (figura 3.3). Con esto podemos utilizar la aplicación en su versión monoscópica. Si queremos configurar más de una cámara será necesario definirlas en dicho archivo (apéndice C.3 en la página 114).

Si hemos activado la opción "**ShowConfigDialog**"se mostrará la ventana de configuración al iniciar la aplicación (figura 3.4 en la página siguiente). Si no, siempre podemos acceder a través de la barra de menús. Este menú forma parte del proyecto QtOgre, al que se le ha añadido la opción de poder mostrar la ventana al iniciar la aplicación. Podemos seleccionar el motor gráfico a utilizar (directX o openGL), la resolución de la pantalla, y otras opciones gráficas como el filtro antialiasing o la sincronización vertical.

Se lanza finalmente la aplicación que consta de dos ventanas principales. Una ventana dónde aparecerá la imagen de cámara, y una ventana de opciones, con distintos aspectos de la aplicación qué podemos configurar. Se decidió realizar un diseño a dos ventanas ya que suele ser habitual el uso de la ventana gráfica a pantalla completa, ya que es requerido por los proyectores para poder activar la opción estereoscópica. De esta manera podemos cerrar o mostrar la ventana de opciones en cualquier momento pulsando la tecla **Tabulación**. Por defecto la aplicación viene configurada conciertos atajos de teclado:

- **Tabulación:** Muestra u oculta la ventana de opciones.

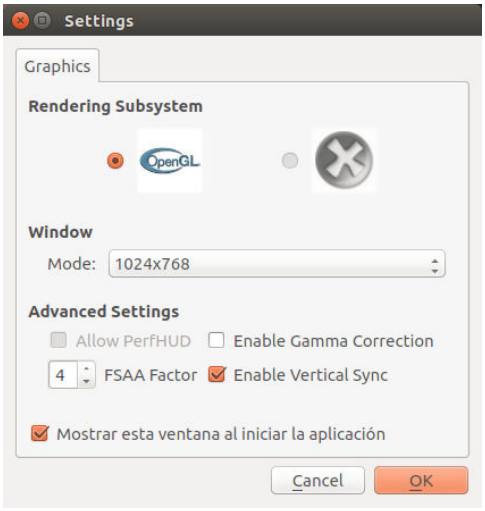


Figura 3.4: Ventana de configuración.

- **W, A, S, D, Q, E, Z y ratón:** Modo navegación: permite mover la cámara virtual en distintas direcciones. Es util para depurar. La tecla Z coloca la cámara en su posición inicial.
- **X:** Modo depuración: Muestra los planos receptores de sombras y posición e información de las marcas.
- **G:** Activa la rejilla de la escena (figura 3.5 en la página siguiente).
- **P:** Muestra por línea de comandos estadísticas y tiempos de algunas funciones críticas que permiten medir el rendimiento de la aplicación .
- **M:** Activa el modo espejo invirtiendo en el eje horizontal .
- **F1 - F7:** Permiten activar los distintos modos de visualización estereoscópica .
- **Inicio, fin, RePag, AvPag:** Si la aplicación se encuentra conectada al bastidor estas teclas mueven las cámaras físicas tanto en separación como en convergencia.

3.1.2. Tipos de objetos e inicialización

Una vez iniciada la aplicación podemos mostrar marcas a la cámara, y si éstas tienen objetos asociados, aparecerán sobre éstas. Para que una marca funcione debe estar registrada por la aplicación. Esto asocia un identificador de marca a uno de los

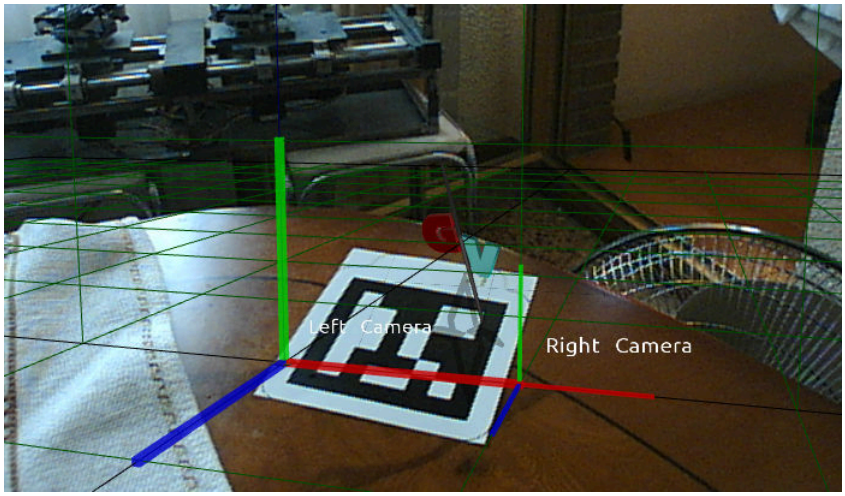


Figura 3.5: Escena vista desde un punto de vista modificado, con la rejilla activada.

tipos de objetos implementados, todos derivados de la clase **objectNode**. Cada tipo añadirá funciones distintas a la marca.

Para poder asociar un objeto a una marca éste tiene que estar presente en la carpeta de medios. Dependiendo del tipo de objeto constará de un archivo de malla (.mesh), archivos de materiales o texturas. Opcionalmente podemos añadirlo a la escena editando el archivo XML de escena de nuestro programa (.scene) (apéndice C en la página 113). Puede ser útil para aplicar transformaciones al objeto, cómo traslaciones o cambio de escala para conseguir el tamaño de deseado en la aplicación. Sin embargo la carga del objeto podemos hacerlo a través de código si lo deseamos. Podemos ver el ejempo de un objecto definido en el listado 3.1.

Listing 3.1: Definición de un objeto en el archivo XML de escena.

```
<node name="Lata">
        <scale x="0.101" y="0.114" z="0.101" />
        <rotation qw="0.707" qx="0" qy="-0.707" qz="0"/>
        <position x="0" y="0" z="0" />
        <entity meshFile="lata.mesh" name="Lata"/>
</node>
```

Finalmente hay que declarar el objeto y asociarlo a un número de marca. Esto se realiza ya en el código de la aplicación, instanciando la clase **objectNode** o alguna de sus clases derivadas. El objeto creado se inserta en una lista que contiene todas las marcas, y que será accedida desde distintas partes de la aplicación.

Se han implementado varios tipos de nodos que sirven de ejemplo para futuras implementaciones (figura 3.6 en la página siguiente).

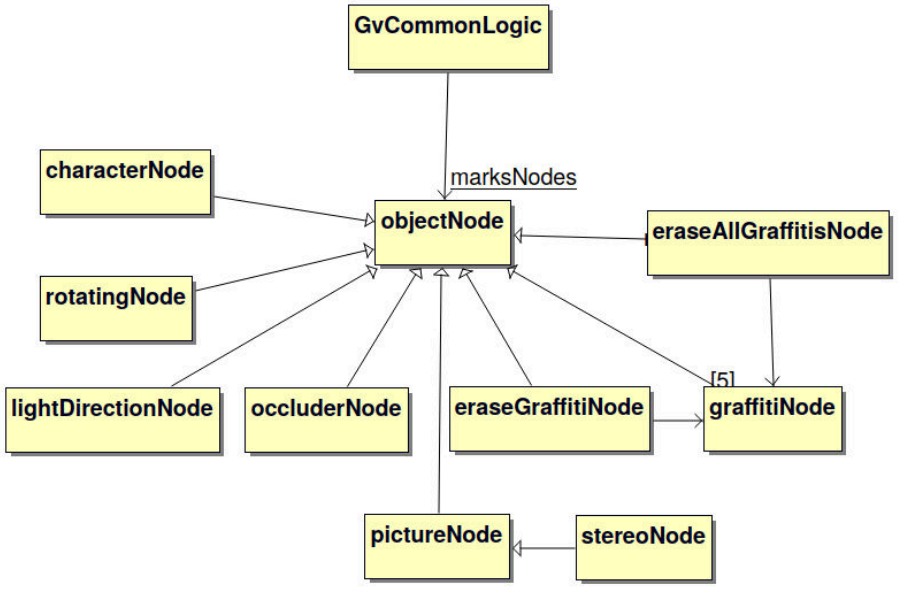


Figura 3.6: Diagrama de clases de los tipos de objetos implementados.

La clase base ObjectNode

Es el tipo de objeto base que implementa el comportamiento básico de una marca: aparecer y desaparecer en la escena de forma gradual, posicionamiento, creación del plano receptor de sombras, etc... . El objecto se instancia pasando como parámetros: el nombre del objeto, que debe coincidir con uno de los declarados en el archivo de escena, el número de la marca al que se va a asociar y el tamaño de la marca física, que definirá el ancho del objeto receptor de las sombras. Por ejemplo, la instrucción `marksNodes.insert (1, new objectNode("Robot", 1, 0.10))` asocia el modelo Robot definido en el archivo de escena, a la marca número 1, con un receptor de sombras de 10 cm.

Para desarrollor otro tipo de objetos con distintas características, se pueden derivar nuevas clases a partir de esta, redefiniendo las clases virtuales que se muestran en el listado 3.2.

Listing 3.2: Métodos virutales de la clase objectNode.

```
// es llamada cada vez que se detecta la marca asociada
virtual void locationChanged(Ogre::Matrix4 trans);
//es llamada cuando se activa el modo depuracion
virtual void toggleDebug(bool toggle);
//es llamada una vez por fotograma, para poder actualizar animaciones, effectos, etc.
virtual void updateMovement (float timeElapsedInSeconds);
//es llamada cuando el objeto desaparece, para poder implementar nuevos efectos
virtual void fadingStep();
```

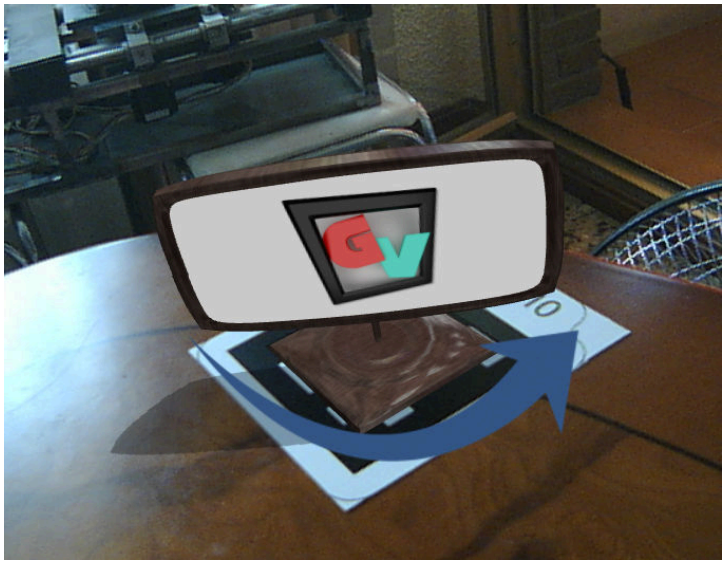


Figura 3.7: Panel giratorio.

RotatingNode: objetos que rotan sobre la marca

Permite definir un objeto como base, que permanecerá estático con respecto a la marca, y otro objeto que rotará con respecto al primero sobre un eje definido por nosotros. En el ejemplo de la figura 3.7, la base queda estática y el letrero luminoso da vueltas sobre ésta.

En la creación del objeto, deberemos pasar además de los parámetros nombre, número de marca y ancho de la base, el nombre del objeto que rota, la velocidad de rotación y el eje de giro (`rotatingNode(Ogre::String mNodeName, int markNumber, float shadowRecWidth, Ogre::String rotNodeName,float rotSpeed, Ogre::Vector3 rotAxis)`).

StereoNode: marca de convergencia del bastidor

Este tipo de marca obtiene la cantidad de paralaje entre la cámara izquierda y la cámara derecha y comunica al módulo *SistemaMotores* el valor calculado para conseguir la convergencia al punto exacto donde se encuentra la marca (figura 3.8 en la página siguiente). Sobre la marca se sobreimpresiona un *widget* que permite activar o desactivar el movimiento. Utilizando la rotación, situamos el indicador rojo en la parte superior, indicando así que se desea converger (figura 3.9 en la página siguiente).

Una vez establecida la convergencia, podemos adelantar o retrasar la marca para comprobar el paralaje en distintos puntos de la escena. Esta cantidad representa una distancia en centímetros teniendo en cuenta la dimensión de la pantalla, parámetro que

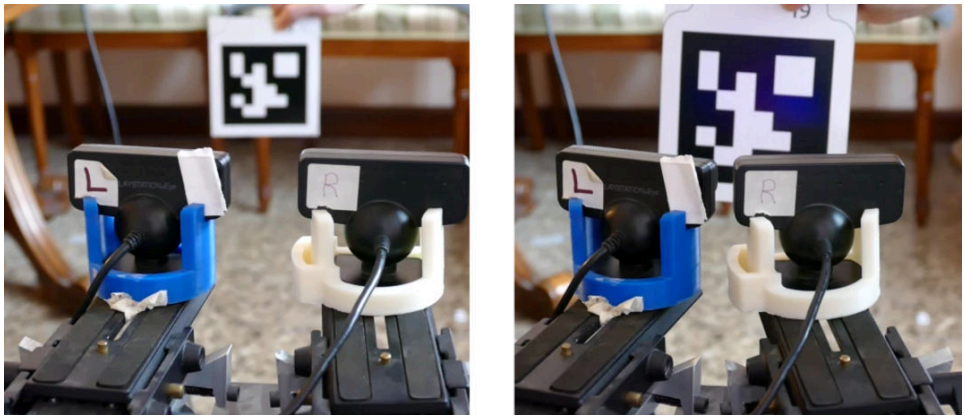


Figura 3.8: Convergencia automática de las cámaras.

Figura 3.9: Node de ajuste del bastidor.

se encuentra en los archivos de configuración. El valor cambiará al color rojo cuando se considere que el paralaje excede los valores de seguridad. Para este cálculo se utiliza también el parámetro de distancia del espectador a la pantalla. Consideramos que una marca está dentro de la zona de confort estereoscópico, cuando no sobresale de la pantalla más del 20 % de la distancia entre el espectador y la pantalla (paralaje negativo) y no se aleja de ésta más del 80 % de dicha distancia hacia dentro de la pantalla ([25]).

OccluderNode: objetos reales que ocluyen a objetos virtuales.

Se ha implementado un tipo especial de marcas, a cuyo objeto asociado se le aplica el mismo material receptor de sombras. Si el modelo 3D representa fielmente a un objeto real, podemos reemplazar la marca por el objeto, haciendo de esta manera que dicho objeto reciba las sombras virtuales de otros modelos 3D, incluso ocluyendo a estos.

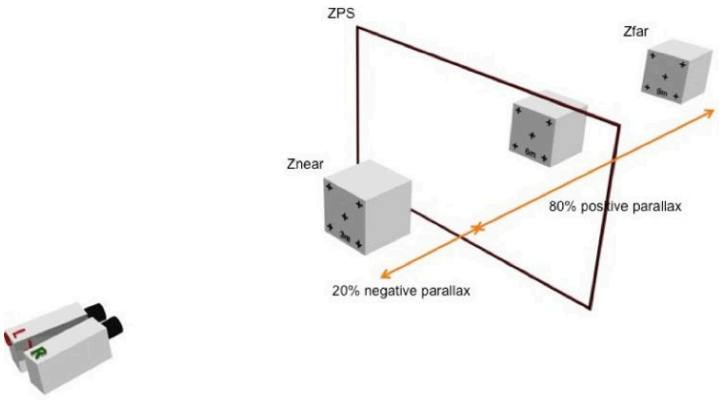
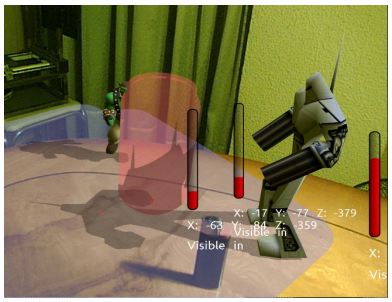
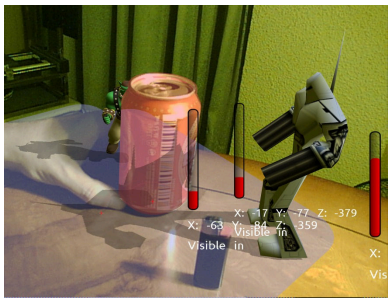


Figura 3.10: Zona de confort estereoscópica.

En las siguientes figuras vemos como se prepara una escena con tres marcas (subfigura 3.15(a)), una de las cuales está asociada a un modelo de una lata de cerveza. El modo de depuración nos permite situar entonces la lata real en la posición exacta, pudiendo opcionalmente quitar la marca ya que está configurada como permanente (subfigura 3.11(b)). El resultado final es una lata real que recibe sombras virtuales y ocluye objetos virtuales (subfigura 3.15(b)). Esta técnica se puede utilizar también para definir planos reales como mesas, paredes, suelos, etc... y que reciban sombras.



(a) Tres marcas, una de ellas de tipo *occluderNode*



(b) Ajustando el objeto real

(c) Resultado final

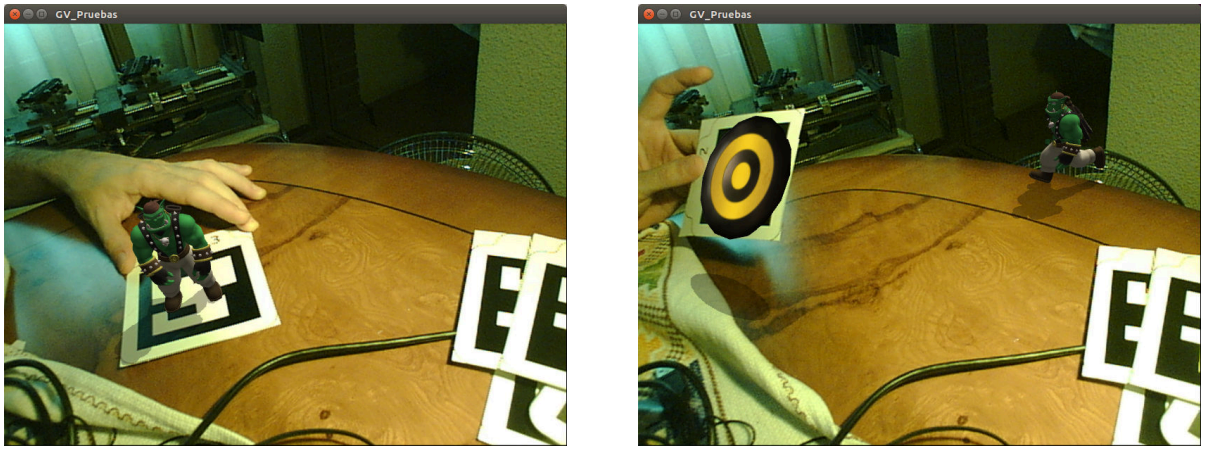
Figura 3.11: Objetos reales ocluyendo objetos virtuales

.

CharacterNode: ejemplo de interacción con un personaje

Este tipo de nodo es un ejemplo de interacción entre dos marcas. La marca principal está asociada a un personaje, que una vez presentado en la escena define el plano sobre el que se moverá. Una segunda marca sirve para definir el punto hacia el que queremos

mover el personaje. La asociación entre las dos marcas se realiza llamando a la función *setWalkToNode*. En la implementación de la función virtual *updateMovement*, se calcula él punto que resulta de la proyección del nodo objetivo sobre el plano de movimiento del personaje. En cada *fotograma*, este se moverá en la dirección del objetivo.



(a) Situamos al personaje, definiendo así el plano sobre el que se moverá

(b) Al mostrar la marca objetivo, el personaje correrá hasta el punto perpendicular

Figura 3.12: Ejemplo de marca de personaje.

LightDirectionNode: modificación de la dirección de la luz direccional

El tipo de nodo LightDirectionNode permite establecer la dirección de la luz principal definida en la escena. Presentando su marca asociada a la cámara podemos ajustar esta dirección, y por tanto la dirección de las sombras de todos los objetos para que coincida con las sombras que reciben los objetos reales, y así mejorar la coherencia de la imagen resultante (figura 3.13 en la página siguiente).

Tipos de nodos de GvGraffiti

Para la aplicación de demostración GvGraffiti se implementaron varios tipos de marcas que permiten pintar sobre la cámara o sobre un fondo predefinido trazos de colores simulando un graffiti. La aplicación utiliza unos botes que representan *sprays* de graffiti de distintos colores (figura 3.14 en la página siguiente). Cada uno contiene tres marcas, de tal manera que según la posición del bote con respecto a la cámara se comportará de distinta manera. Dispuesto de perfil, solo se nos mostrará un indicador para saber en qué posición vamos a pintar. Cuando giramos el bote hacia la pantalla

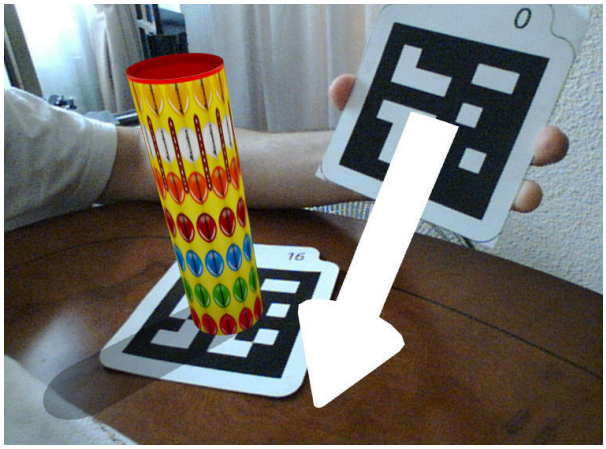


Figura 3.13: Modificación de la dirección de la luz principal.

Figura 3.14: Botes con marcas diseñadas para la aplicación GvGraffiti.

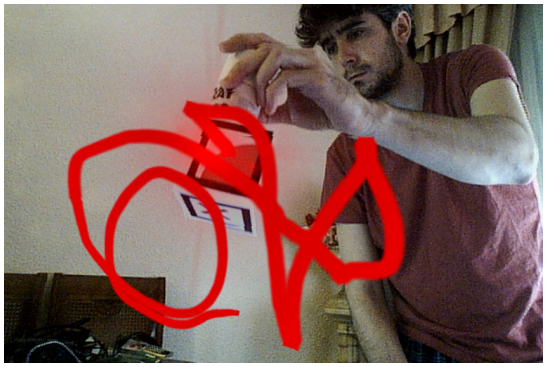
empezamos a pintar un trazo del color seleccionado. Si mostramos la base del bote borramos los trazos que hemos pintado.

Los cuatro tipos de marcas implementadas son las siguientes:

- El tipo **graffitiNode** permite asociar una marca a un color pasado como parámetro. El movimiento de la marca frente a la cámara crea un trazo continuo cuyo grosor depende del ángulo de la marca con respecto a la cámara.
- El tipo de marca **billboardNode** muestra un indicador de la marca sobre la pantalla. Esto es necesario cuando se ejecuta la aplicación con una imagen de fondo diferente de la cámara, para saber dónde empezaremos a pintar al girar el bote.
- El tipo de marca **eraseGraffitiNode** borra los trazos del color que estamos

utilizando.

- La marca del tipo **eraseAllGraffitisNode** borra todos los trazos de todos los colores.



(a) Uso de la marca de pintado sobre la cámara.

(b) GvGraffiti con fondo predefinido.

Figura 3.15: Aplicación interactiva GVGraffiti.

Marcas de imágenes

El tipo de marca **pictureNode** permite asociar una imagen a una marca. A diferencia de una marca de objeto, ésta no tiene que tener un objeto ya definido en la escena. El nombre que pasamos como parámetro indicará la imagen a utilizar, qué habrá sido almacenada en la carpeta de recursos de la aplicación. Adicionalmente podemos pasar un parámetro que representa la escala de la imagen con respecto a la marca y dos valores de desplazamiento horizontal y vertical con respecto a ésta.

En la aplicación de prueba hay tres marcas asociadas a tres imágenes de distintas prendas. Es un ejemplo de un prototipo de una aplicación de realidad aumentada para una tienda de ropa (figura 3.16 en la página siguiente). También podemos añadir nuevas imágenes durante la ejecución, utilizando el botón **"Import picture"** de la pestaña marcas (ver sección siguiente).

3.1.3. Descipción de la pestaña de configuración de las marcas

La pestaña **"Marcas"** muestra un listado de todas las marcas que tienen objetos asociados en la escena. Para cada una se muestra información relevante y controles para configurar su apariencia o funcionamiento.

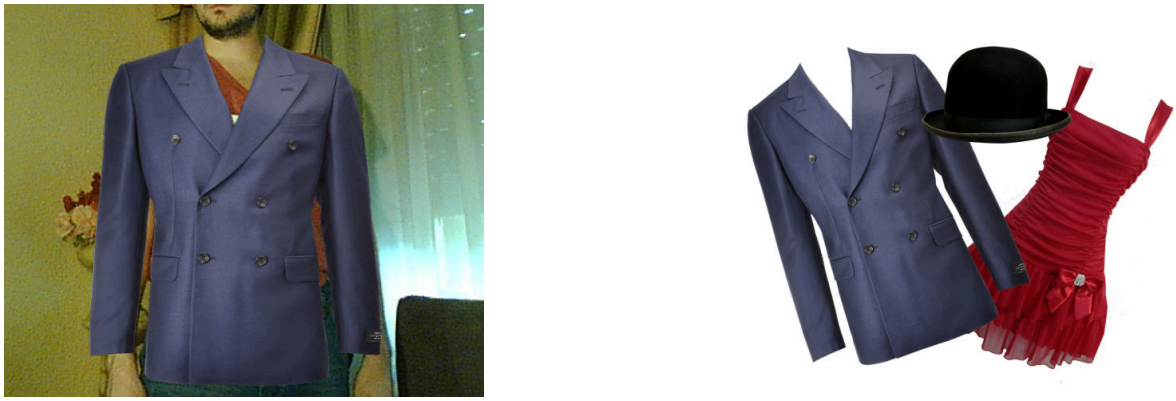


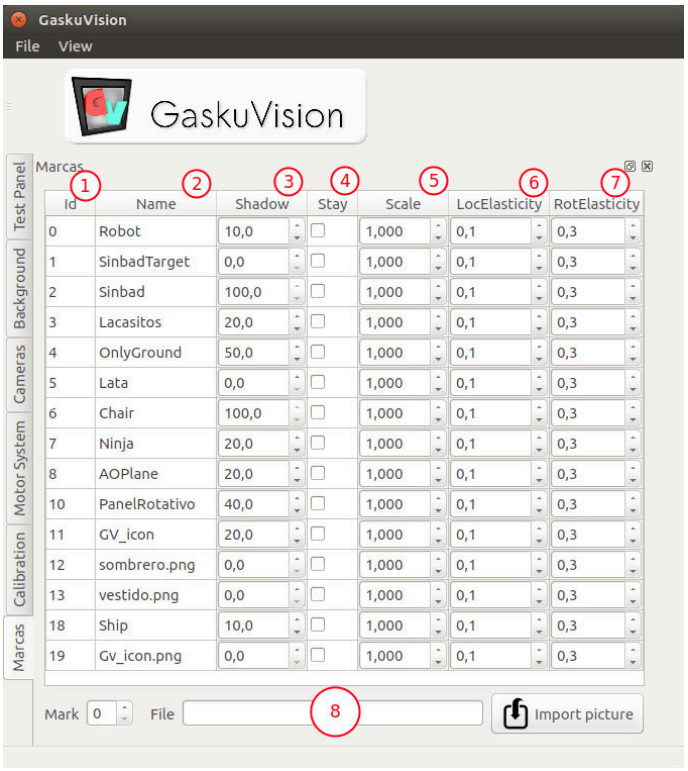
Figura 3.16: Ejemplo de prendas asociadas a marcas.

1. **Número de la marca.**
   Identificador asociado a la marca.

2. **Nombre del objeto.**
   Este
   coincide con la definición del
   nodo en el archivo de escena.

3. **Tamaño**
   del receptor de Sombras.
   Medida en centímetros del
   plano destinado a recoger las
   sombras. Permite ajustarlas
   a la marca física, o ampliarlo
   cuando la marca se encuentre
   sobre una superficie para
   que ésta reciba también las
   sombras del objeto virtual.

4. **Objeto permanente.**
   Permite activar o
   desactivar la permanencia del
   objeto en la escena cuando
   no este presente su marca.



5. **Escala del objeto.**
   Permite variar el tamaño del objeto virtual sobre la marca.

6. **Elasticidad de la posición.**

Indica el peso relativo que tiene la posición de la marca detectada en el cálculo de la posición del objeto.

7. **Elasticidad de la orientación.**
   Peso relativo que tiene la orientación de la marca para el cálculo de la orientación del objeto.

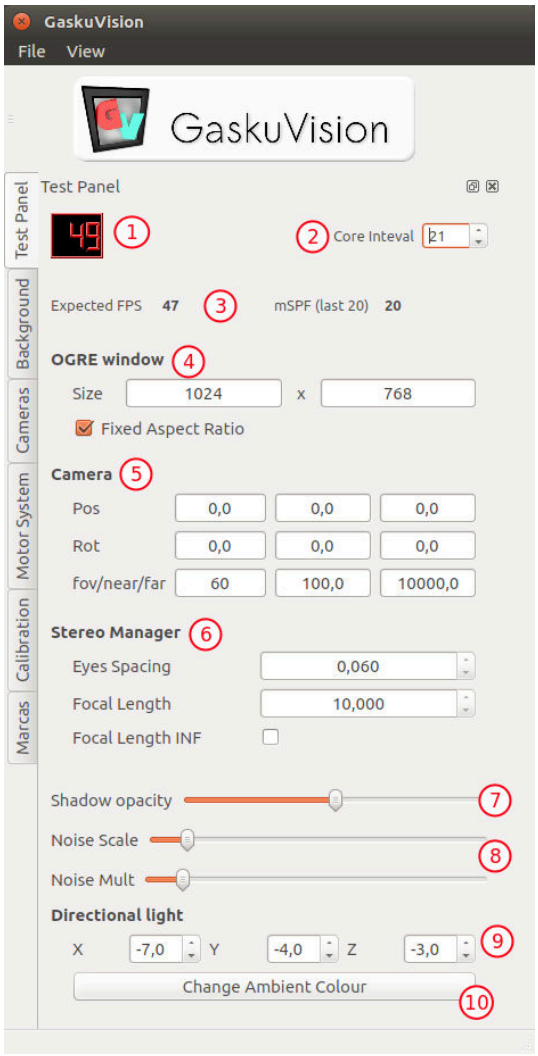
8. **Importar imágenes externas.**
   Permite cargar archivos de imágenes asociados a una nueva marca. Estas imágenes se mostrarán siempre paralelas a la pantalla.

3.1.4. Descipción de la interfaz de desarrollo

La pestaña **"Test Panel"** recoge diversos controles que pueden ser útiles en el desarrollo de las aplicaciones o que no tienen cabida en el resto de las pestañas.

1. **Tasa de refresco de la aplicación.**
   Cantidad de cuadros por segundo del motor gráfico Ogre.

2. **Intervalo de refresco del motor gráfico.**
   Cantidad de milisegundos entre actualizaciones del motor gráfico.



3. **Tasa de refresco esperada y tiempo entre actualizaciones.**
   Estos valores deberían coincidir con los valores anteriores. Se muestran por razones de desarrollo (apartado 2.5.1 en la página 26).

4. **Dimensiones de la ventana principal.**
   Muestra las dimensiones de la ventana del el motor gráfico. La opción "Keep Aspect Ratio"permite elegir si se respetan las dimensiones de la cámara, o se ocupa toda la ventana.

5. **Datos de la cámara principal.**
   Posición, rotación y valores de apertura y plano cercano y lejano de la cámara principal.

6. **Datos estereoscópicos.**
   Muestra separación de las cámaras virtuales, distancia al plano de convergencia y la opción de converger al infinito (modo paralelo).

7. **Intensidad de las sombras.**
   Este deslizador ajusta la intensidad de todas las sombras, pudiendo elegir desde no mostrarlas, hasta sombras 100 % opacas.

8. **Filtro de ruido sintético.**
   Configura el ruido que se aplica a los objetos virtuales (apartado 3.7 en la página 72). El primer deslizador escala el patrón de ruido, ajustando el tamaño del grano. El segundo controla la cantidad de ruido.

9. **Orientación de la luz principal.**
   Permite controlar la dirección de la luz direccional que alumbra la escena y por tanto la dirección de las sombras.

10. **Color ambiente.**
    Controla el color la iluminación de la escena para una mejor integración de los objetos en la escena.

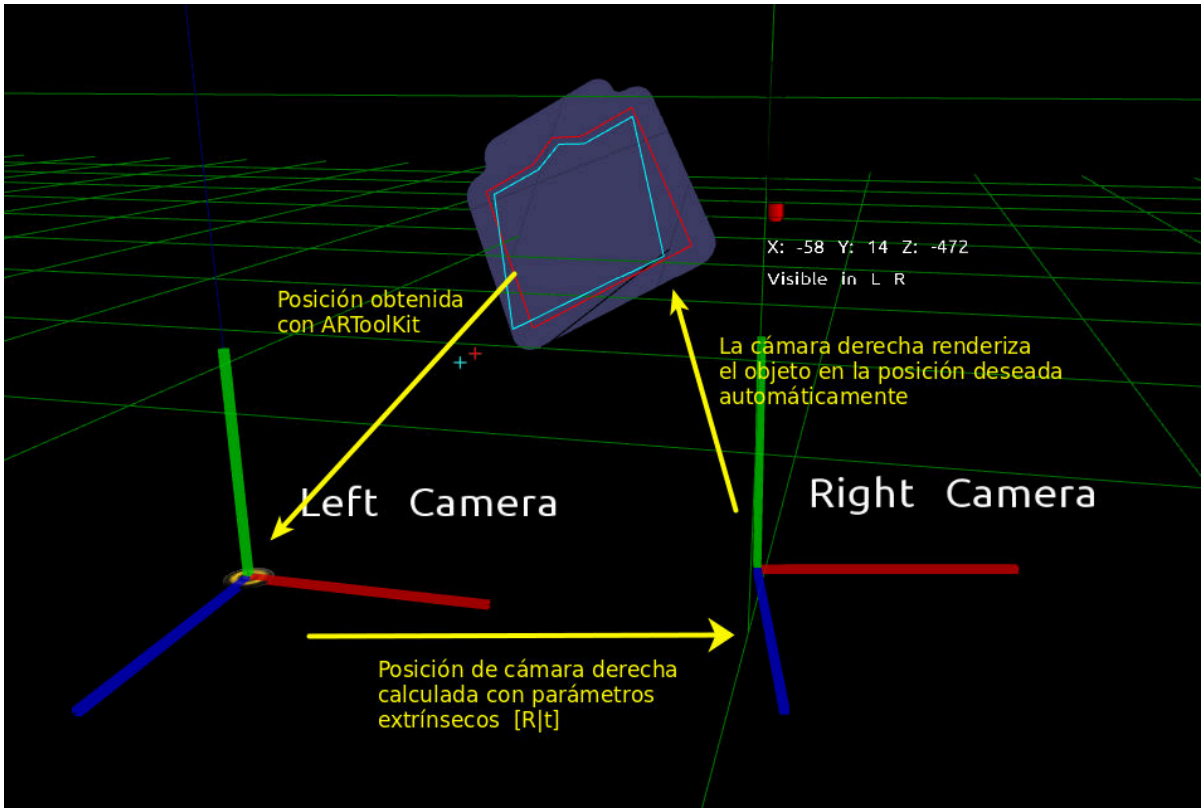


Figura 3.17: Método de visualización por posicionamiento de la cámara derecha.

3.2. Visualización estereoscópica

La visualización estereoscópica de contenidos requiere de dos procesos: la mezcla de las dos señales adquiridas y la renderización del contenido virtual de forma coherente con la escena real.

La idea inicial era utilizar los parámetros extrínsecos adquiridos en la calibración para situar el par de cámaras virtuales en la escena de manera análoga a cómo se encuentran las cámaras reales. Al analizar la marca de la imagen izquierda y colocar el objeto virtual su posición, la cámara derecha debería automáticamente renderizar el mismo objeto sobre la marca, resultando una imagen coherente (figura 3.17).

La integración obtenida con este proceso no es perfecta (figura 3.18 en la página siguiente). Para mejorar esta coherencia se decidió implementar una segunda opción. Utilizando la posición de la marca adquirida por la cámara derecha, situamos directamente el objeto en la marca, logrando que el posicionamiento sea perfecto. Este método tiene la ventaja de que no requiere calibración. Sin embargo, el método original sigue siendo útil, ya que si los objetos se alejan de las marcas que definen su posición,

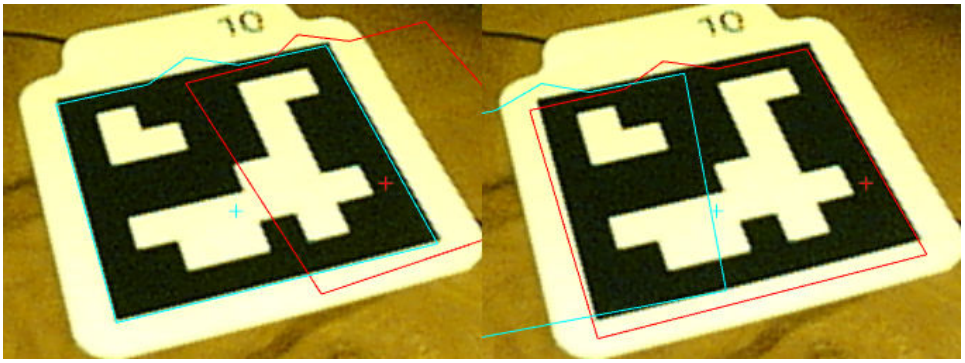


Figura 3.18: Izquierda, en cyan, la posición calculada con ARToolKit. A la derecha, en rojo, la marca deducida a través de los parámetros extrínsecos.

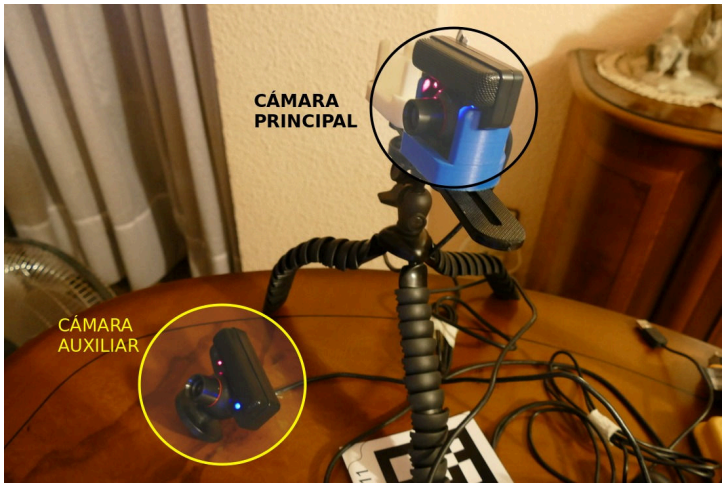


Figura 3.19: Posicionamiento de una cámara auxiliar.

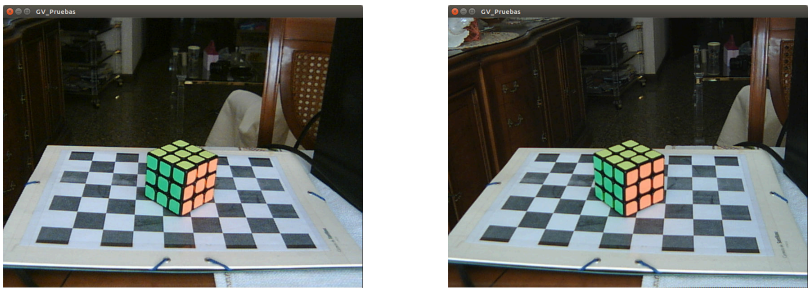
debido a animaciones u otras razones, la posición seguirá siendo coherente, evitando molestias.

Además, extraer los parámetros extrínsecos nos permite utilizar una cámara auxiliar, en una posición lejana al par de cámaras, para conseguir ver la marca en ángulos donde no sería visible en situaciones normales (figura 3.19). Si el conjunto de cámaras se encuentra bien calibrado, el contenido se puede renderizar en la vista principal, con su posición definida a través de la cámara auxiliar.

3.2.1. Modos estereoscópicos

Debido a la variedad de métodos de visualización estéreo, la aplicación debía soportar diversos modos. Además ciertos modos fueron sugeridos por técnicos de estereoscopía usuarios del programa para facilitar la labor de calibración. Los tipos soportados son:

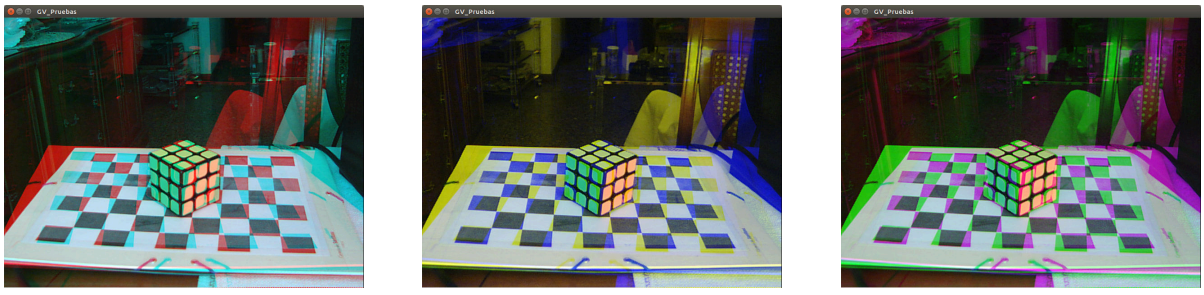
- Vistas independientes de las cámaras izquierda o derecha (figura 3.20).



(a) Imágen cámara izquierda **(b)** Imágen cámara derecha

Figura 3.20: Imágenes izquierda y derecha

- Tres tipos de modo anaglifo (figura 3.21). Es un sistema barato para visualización 3D, aunque se pierde la información de color. Los dos fotogramas utilizan dos espacios de color independientes, que son filtrados por las gafas para separar la información (figura 3.22 en la página siguiente).



(a) Rojo-cian **(b)** Amarillo-azul **(c)** Verde-magenta

Figura 3.21: Modos anaglifo

- Modos Side-by-side, Over-under y modos entrelazados (figura 3.23 en la página siguiente y figura 3.24 en la página 55). Estos modos permiten la proyección en proyectores 3D con gafas activas (figura 3.25 en la página 55). Un mismo fotograma

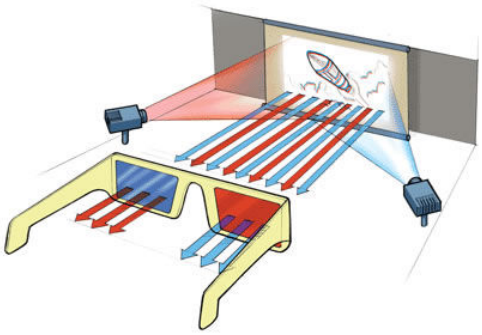
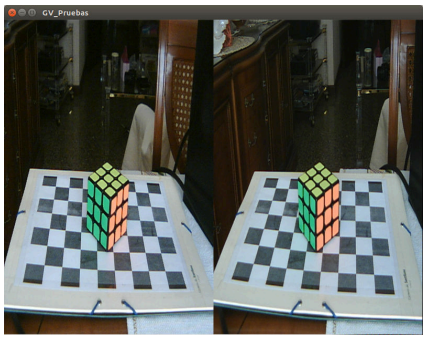
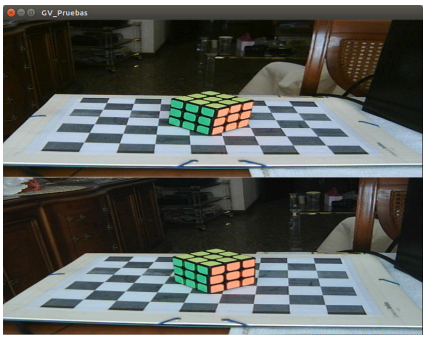


Figura 3.22: Visualización anaglifo con gafas pasivas.

contiene las dos imágenes, siendo la única diferencia la disposición de estas: medio fotograma cada una, lineas alternadas o cuadrícula. Destacar que en todos los casos se pierde la mitad de la resolución en uno de los dos ejes.



(a) Modo Side-by-side

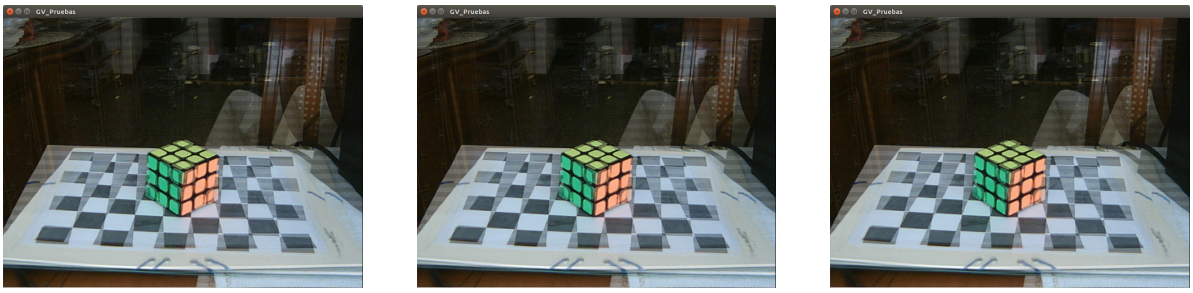


(b) Modo Over-Under

Figura 3.23: Modos de imágenes adyacentes.

- Otros modos auxiliares implementados incluyen (figura 3.26 en la página siguiente):

  - Modo partido: solo se ve la mitad de cada fotograma. Es útil para la calibración fina de las cámaras, para evitar paralajes verticales.
  - Modo mezcla al 50 %: similar al anaglifo pero con información de color. Permite ver las dos vistas sobre-impresionadas.
  - Modo diferencia: se realiza una sustracción de ambos fotogramas. En los modelos de bastidor con espejo es posible disponer las dos cámaras con una separación nula, es decir, las cámaras pueden estar virtualmente en la misma posición. De esa manera podemos corregir pequeñas diferencias de

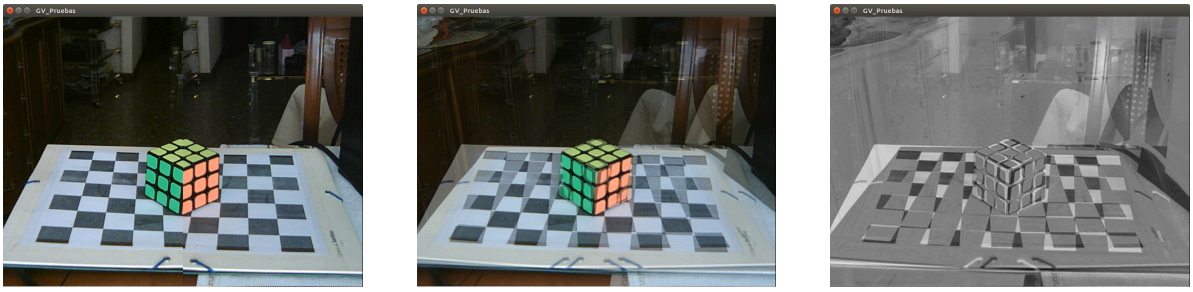


(a) Entrelazado vertical **(b)** Entrelazado horizontal **(c)** Entrelazado en cuadrícula

Figura 3.24: Modos de imágenes entrelazadas.

Figura 3.25: Visualización del resultado en un proyector activo.

colocación antes de separarlas. Una vez se consiga la disposición perfecta, el modo diferencia debería mostrar toda la pantalla en gris.



(a) Imágenes partidas **(b)** Modo mezcla al 50 % **(c)** Modo diferencia

Figura 3.26: Modos útiles en la calibración.

3.3. GvCamManager: captura y análisis de las señales de video

Este módulo gestiona las entradas de video y su posterior tratamiento. Algunas de las funciones asociadas a este módulo son: reconocimiento de las marcas en la imagen y almacenamiento de su posición.

3.3.1. Uso del patrón Singleton

Dada la naturaleza del módulo *GvCamManager*, se requería limitar el número de instancias permitidas, ya que una sola instancia gestiona todas las cámaras. Se evita con ello que por un error de programación dos instancias compitan por los mismos recursos hardware.

El patrón de diseño *singleton*[22] (instancia única) está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella. El patrón singleton se implementa creando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado).

Para ello se diseño la clase principal como derivada de la clase **Ogre::Singleton**, que viene incluida en la librería Ogre. Además se reimplementan los métodos estáticos `static GvCamManager& getSingleton(void)` y `static GvCamManager* getSingletonPtr(void)`, para que puedan ser accedidos desde fuera de la librería.

De esta manera, basta con crear en el código principal una instancia de la clase (que será única) para posteriormente requerir un puntero a la instancia creada:

```
new GvCamManager(mSistemaMotores);
mGvCamManager = &GvCamManager::getSingleton();
```

De esta manera se evita además tener que pasar un puntero de la instancia creada a las distintas clases que utilicen *GvCamManager*, ya que es accesible como una variable global.

3.3.2. Diseño multihilo

El módulo *GvCamManager* contiene las secciones de código más críticas en cuanto a rendimiento. Es el encargado de adquirir los fotogramas de cada cámara, analizarlos

para detectar las marcas visibles, calcular la posición de estas con respecto a la cámara, y servir esta información a la aplicación principal. La etapa más costosa es el análisis del fotograma en busca de marcas, proceso que se realiza de manera independiente para cada cámara. Por tanto, en el caso de uso de dos o más cámaras, la ejecución en paralelo de estos procesos puede suponer una mejora sustancial sobre un procesador con varios núcleos.

Existe además un máximo de tiempo disponible para estas tareas, marcado por la tasa de refresco de la cámara. En el caso que la suma de tiempos de estos procesos fuera mayor que el tiempo entre capturas se perderían fotogramas, resultando una señal de video menos fluida. El diseño multihilo es la única manera de garantizar un óptimo funcionamiento para más de una cámara.

Además, los procesos relativos a cada cámara deben ser independientes del hilo principal, encargado entre otras cosas de renderizar los elementos sintéticos. Esto permite que las animaciones de estos elementos puedan funcionar a una tasa mayor que la marcada por las cámaras. Esto obliga a garantizar el acceso a los datos comunes en exclusión mutua.

Tanto para la creación y gestión de los distintos hilos, como para el acceso a los recursos compartidos, se usan los tipos de la librería Qt (*QThread* y *QMutex* respectivamente). En la figura 3.27 en la página siguiente se muestra un gráfico que representa la interacción entre las distintas clases implicadas.

Según este esquema los hilos de captura y seguimiento de marcas funcionan independientemente del hilo principal de la aplicación. Cada uno de los hilos hace una llamada bloqueante a la cámara. De esta manera el hilo se bloquea hasta que recibe un nuevo fotograma, evitando el gasto innecesario de ciclos de procesador. Después analiza la imagen, extrayendo la información de las marcas y almacenándola en exclusión mutua. Finalmente, copia la imagen en otro *buffer*, utilizado por la aplicación principal para el renderizado. Esta duplicación en memoria es necesaria, ya que se puede solapar la lectura de la imagen para mostrarlo a través de la textura, con la recepción del siguiente frame.

3.3.3. Configuración de las cámaras ARToolKit

Cada cámara está asociada a una instancia de la clase **GvCamTracker**, que es la encargada de adquirir las imágenes, instanciar a ARToolKit, y obtener la información de las marcas en la escena. La configuración de todos los parámetros se hace desde los archivos de configuración (apéndice C.3 en la página 114). Esto permite que cada aplicación defina su comportamiento de manera independiente. Cada cámara lee de los archivos su función (cámara izquierda, derecha, auxiliar...), su frecuencia de adquisición

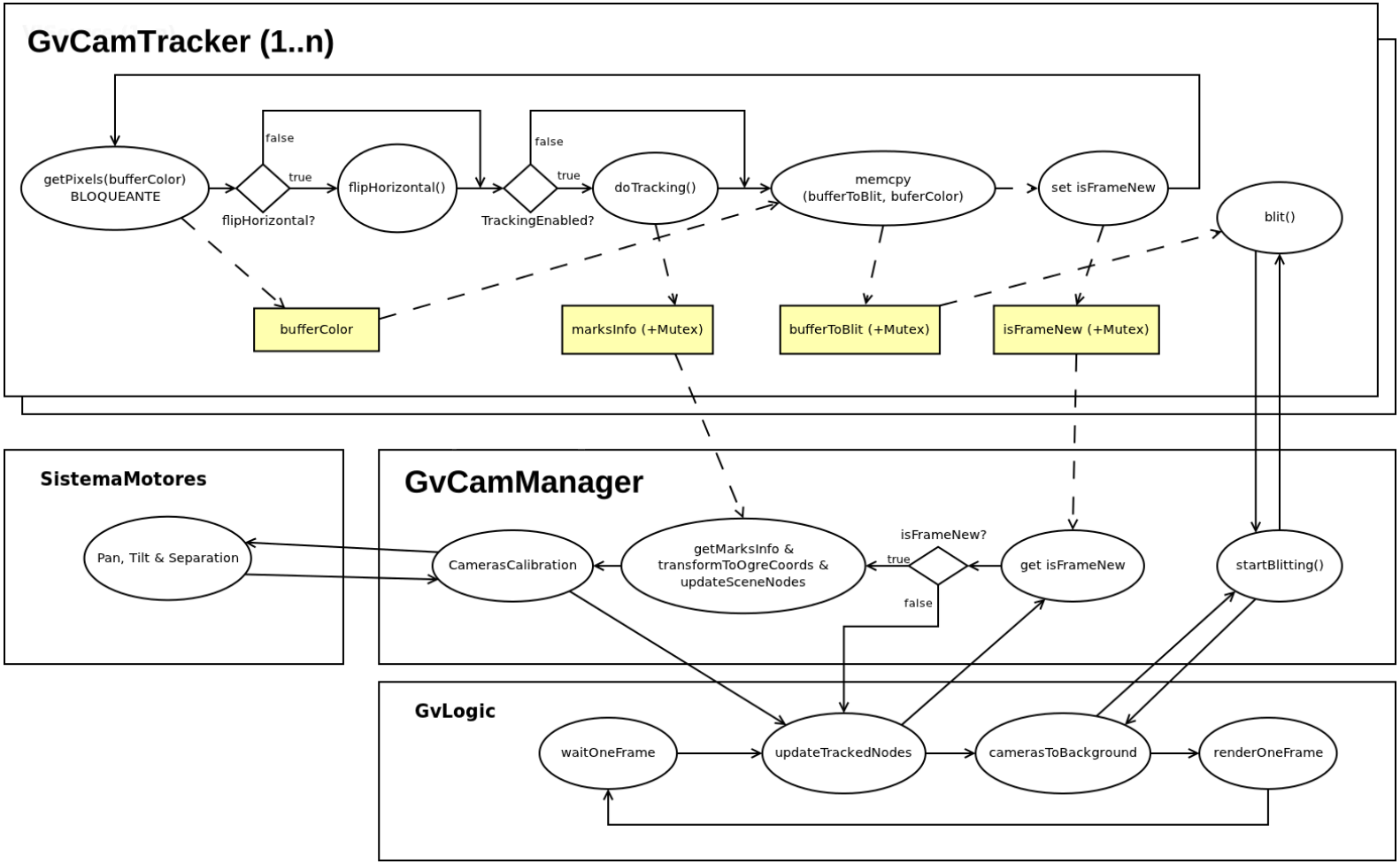


Figura 3.27: Diagrama del diseño multihilo

de imágenes y la resolución. También configura el *tracker* o seguidor de marcas, definiendo el tipo de marcas a seguir (cargando la definición de marcas, si el tipo es de plantilla), el algoritmo a aplicar, etc...

3.3.4. Calibración de las cámaras

Los parámetros intrínsecos de la cámara representan el modelo de una cámara física en un entorno virtual. Además de contener información sobre la apertura (*field of view*), representados por los parámetros α y β, contienen también el desplazamiento del centro de proyección (x_0, y_0) en la imagen obtenida, que idealmente sería el centro de esta, pero debido a que el eje de la óptica no coincide exactamente con el centro del sensor, este punto se encuentra desplazado en la imagen.

$$K = \begin{pmatrix} \alpha & s & -x_0 \\ 0 & \beta & -y_0 \\ 0 & 0 & -1 \end{pmatrix} \tag{3.1}$$

Además, los parámetros de distorsión representan la deformación de la imagen debido a la propia óptica de la cámara.

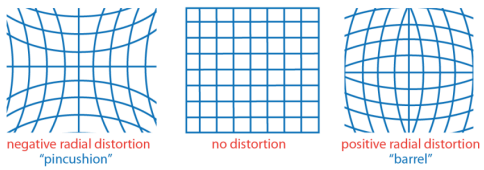


Figura 3.28: Distorsión radial, imagen de [18]

La carga de los parámetros intrínsecos se hace a través de los archivos de configuración incluidos, obtenidos a partir de una aplicación externa (camera-calib [7]), disponible en los repositorios de Ubuntu. Esta aplicación analiza una serie de imágenes con una cuadrícula de calibración, extrayendo la posición de los vértices de esta y calculando los parámetros intrínsecos, así como las poses de las distintas tomas que minimizan el error una vez reproyectados los puntos calculados sobre las imágenes. Pese a que el proyecto contiene todo lo necesario para calibrarlas internamente, ya que el módulo **GvCalibration** es capaz de calcular los parámetros intrínsecos, se decidió mantener la carga de los archivos externos, ya que dicha calibración raramente se modifica, puesto que forma parte de la propia cámara. De esta manera, se pasa como entrada del algoritmo que calcula los parámetros extrínsecos del par de cámaras, mejorando la robustez de esta.

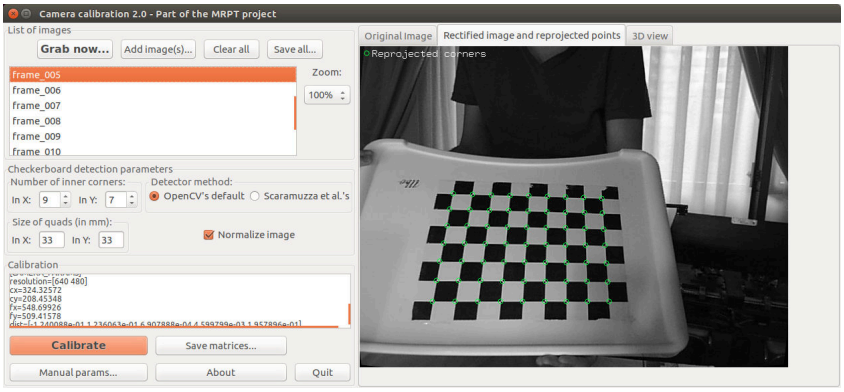


Figura 3.29: Calibración intrínseca con camera-calib

3.3.5. Análisis de marcas y cálculo de la posición

Una vez adquirida la imagen de la cámara, el hilo de ejecución de cada instancia de *GvCamTracker* la analiza en busca de las marcas. Almacena los resultados y marca el fotograma como analizado, quedando a la espera de que el hilo principal use la información. El acceso a las estructuras compartidas (imagen de cámara e información de marcas) se hace en exclusión mutua.

Una vez leídos, ya en el hilo principal de la aplicación, estos han de ser transformados al sistema de coordenadas de Ogre. ARToolKit y Ogre utilizan sistemas de coordenadas de mano derecha, pero con orientaciones distintas. Basta con invertir la escena en los ejes Y y Z (listado 3.3).

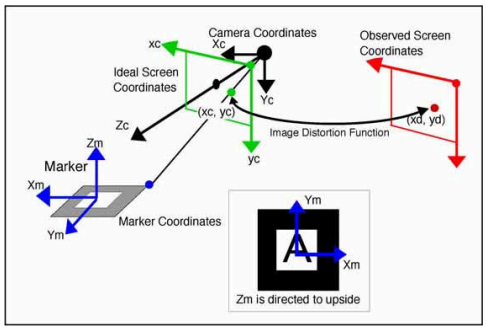


Figura 3.30: Sistema de coordenadas de ARToolKit

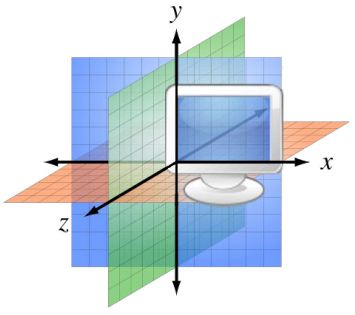


Figura 3.31: Sistema de coordenadas de Ogre

Listing 3.3: Transformación entre sistema de coordenadas

```
Ogre::Matrix4 GvCamManager::ARTK_to_Ogre(Ogre::Matrix4 transfMatrix)
{
```

```
        Ogre::Matrix4 ARTK_To_Ogre = Ogre::Matrix4::IDENTITY;
        ARTK_To_Ogre[1][1] = -1; ARTK_To_Ogre[2][2] = -1;

        return ARTK_To_Ogre * transfMatrix;
}
```

3.3.6. Interfaz del módulo GvCamManager

El control de las cámaras se realiza en la pestaña *Cameras*. La parte inferior de la interfaz permite desplegar los controles para cada cámara individualmente. A continuación se detalla el funcionamiento de los controles.

1. **Lista de cámaras.**
   Se muestran las cámaras activas, su tasa de refresco en tiempo real y la opción de activar el movimiento para esa cámara. La opción **Grids**, activa una rejilla para facilitar el ajuste estereoscópico.

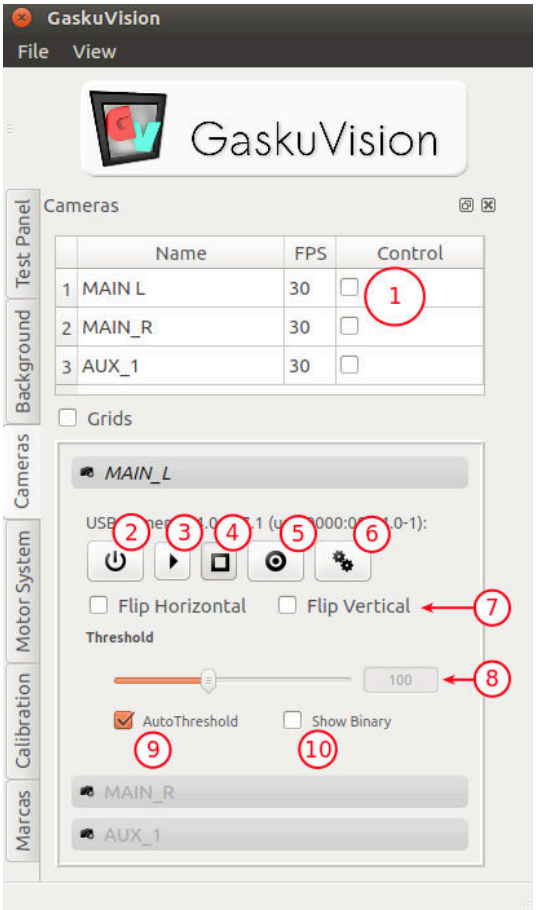
2. **Deshabilitar la cámara.**
   Cierra la cámara y la retira de la lista.

3. **Pausa.**
   Permite pausar la adquisición de nuevas imágenes, manteniendo la actual.

4. **Tracking.**
   Permite habilitar y deshabilitar el *tracking* de marcas para la cámara seleccionada.

5. **Registrar nueva marca.**
   Esta opción permite crear un archivo de patrón para la marca con mayor area que se muestre en la imágen. Se pueden así registrar nuevas marcas de manera manual.

6. **Opciones de cámara.**
   Abre la ventana de configuración de la cámara utilizando la aplicación GUVCview (página 32).



7. **Umbral de binarización.**
   Permite fijar manualmente el umbral a
   partir del cual un píxel es blanco o negro. Se aplica sobre la imagen antes de ser procesada para buscar marcas. Se puede utilizar si las condiciones de luz son estables.

8. **Binarización automática.**

Activa el algoritmo adaptativo de ARToolKit, que rastrea en varios rangos de umbralización hasta que encuentra una marca.

9. **Imagen binaria.**
   Muestra la imagen en blanco y negro producto de la binarización. Permite depurar posibles problemas de iluminación.

3.4. GvCalibration: cálculo de los parámetros extrínsecos

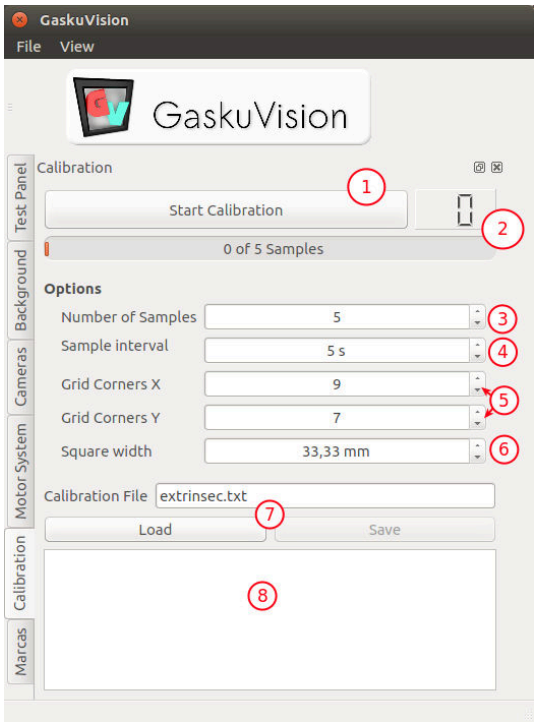
Este módulo permite calibrar las cámaras, utilizando para ello un panel con una cuadrícula impresa. Tras capturar un número determinado de muestras, analiza las imágenes del par de cámaras, extrayendo la posición de los vértices interiores de la cuadrícula. Luego, estos son analizados por la función *cvStereoCalibrate*, que calcula los parámetros extrínsecos, con la que conseguimos la posición de la segunda cámara con respecto a la primera.

Como resultado de la calibración se muestra una cantidad qué representa la calidad de el proceso. El valor **RMS** (Root Mean Square) representa la media de las distancias (en píxeles) entre los vértices detectados en la imagen, con aquellos resultantes de reproyectar los mismos vértices ya en la escena sobre la misma imagen. Este valor debe ser menor que 1 para considerarse una calibración válida.

3.4.1. Interfaz del módulo GvCalibration

Está pestaña muestra la configuración de la calibración del par de cámaras y permiten cargar las configuraciones ya realizadas. A continuación se muestran los controles en detalle.

1. **Iniciar calibración.**
   Comienza el proceso de adquisición de las imágenes.



2. **Estado de la calibración.**
   Se muestra el tiempo restante hasta la siguiente toma, así como el número de muestras que restan por realizar.
3. **Número de muestras.**
   Cantidad total de muestras que se realizarán para la calibración.
4. **Tiempo entre muestras.**
   Intervalo en segundos entre las distintas muestras.
5. **Dimensiones de la cuadrícula.**
   Número de esquinas interiores que contiene la cuadrícula de calibración, tanto en horizontal como en vertical.
6. **Dimensión del cuadrado.**
   Longitud en milímetros del lado de cada cuadrado de la carta de calibración.
7. **Cargar/ salvar desde archivo.**
   Indica el nombre de archivo dónde se almacenarán todos los parámetros resultado de la calibración.
8. **Ventana de visualización.**
   Se muestran los mensajes durante el proceso de calibrado.

3.5. GvBackground: visualización de las imágenes de cámara

Este módulo gestiona los elementos que muestran las imágenes de cámara en el fondo de la aplicación. Sus funciones son la inicialización de los planos que muestran el fondo, la gestión de las texturas de los planos y efectuar diversas correcciones de la visualización estereoscópica.

3.5.1. Efecto keystone: problema y solución propuesta

El efecto keystone en estereoscopía es el paralaje vertical provocado por el posicionamiento no paralelo de las cámaras, que provoca que un mismo punto en la

escena aparezca desplazado verticalmente en ambas imágenes (figura 3.32). Esto provoca malestar en la visión estereoscópica. Pese a que es un problema intrínseco a la estereoscopía, se puede corregir deformando las imágenes antes de visualizarlas.

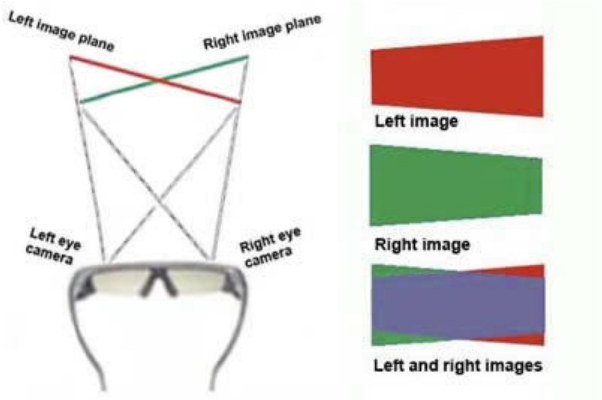


Figura 3.32: Gráfico mostrando la distorsión keystone

En nuestra solución se empezó modificando el polígono sobre el que se aplican las imágenes de cámara. Pero por defecto, Ogre sitúa los elementos 2D en polígonos simples (*quads*) que internamente son renderizados como dos triángulos. Esto provoca que la imagen se deforme de manera incorrecta (figura 3.33 en la página siguiente). Se implementó por tanto una clase que crea el plano de manera manual, con una subdivisión que puede ser inicializada con valores predefinidos para definir el número de subdivisiones verticales y horizontales. Con las cuatro coordenadas de los vértices que definen las esquinas, se crean el resto de vértices interpolando linealmente sus coordenadas. Posteriormente se crean los polígonos en el orden correcto para que las normales de estos apunte a la cámara (sentido antihorario). Se puede ver el algoritmo en el siguiente código:

Listing 3.4: Algoritmo de subdivisión del plano

```
void SubdivRectangle::setCorners(float x1, float y1, float x2, float y2, float x3, float
     y3, float x4, float y4, int rows=defaultRows, int columns=defaultColumns )
{
        float p1x, p2x, p1y, p2y;
        clear();
        begin(_material, RenderOperation::OT_TRIANGLE_LIST);
        for (int r = 0; r < rows + 1; ++r)  {
                float YRatio = ((float)r / rows);
                p1x = x1 + YRatio * (x3-x1);    p1y = y1 + YRatio * (y3-y1);
                p2x = x2 + YRatio * (x4-x2);    p2y = y2 + YRatio * (y4-y2);

                for (int c = 0; c < columns + 1; ++c) {
                        // ratio entre los dos vertices
                        float XRatio = ((float)c / columns);
                        float XX = p1x + XRatio * (p2x-p1x);
                        float YY = p1y + XRatio * (p2y-p1y);
```

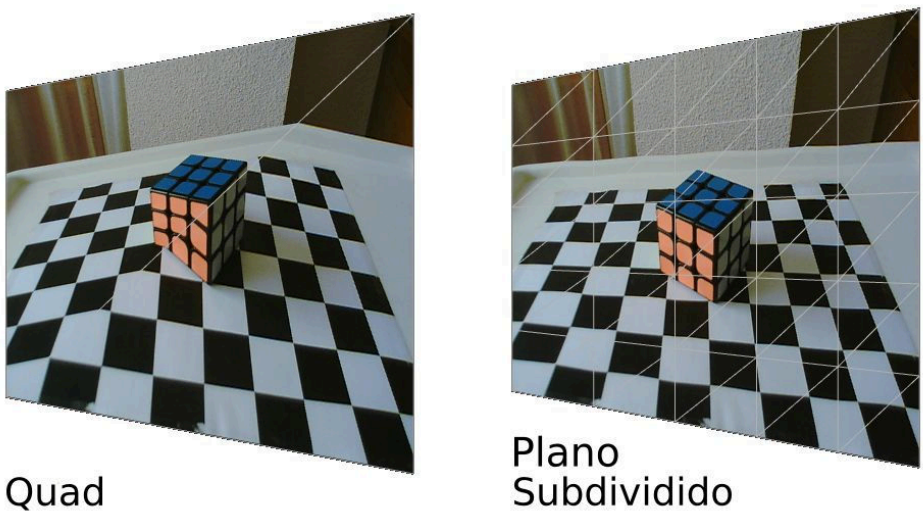


Figura 3.33: Comparativa de deformación del fondo

```
                        // creacion del vertice
                        position(XX, YY, -1);
                        textureCoord(XRatio, YRatio);
                }
        }

        for (int r = 0; r < rows; ++r)
                for (int c = 0; c < columns; ++c)  {
                        int indexIni = (columns+1)*r + c ; // indice inicial del quad
                        // triangulo inferior
                        index(indexIni+0); index(indexIni + (columns+1)); index(indexIni
                            +1);
                        //triangulo superior
                        index(indexIni+1); index(indexIni + (columns+1));       index(
                            indexIni + (columns+2));
                }
        end();
}
```

Esto permite deformar los fondos de manera más regular y es una solución suficiente para ajustes mínimos. En el ejemplo de la figura 3.34 en la página siguiente, se puede comprobar que la deformación se realiza siempre de manera que queden fuera de pantalla las partes de las imágenes que no tienen correspondencia en la otra vista. No obstante, la solución implementada no es la ideal y se propone como mejora la deformación de los planos utilizando una transformación proyectiva implementada en un *shader* (apartado 4.1.2 en la página 81).

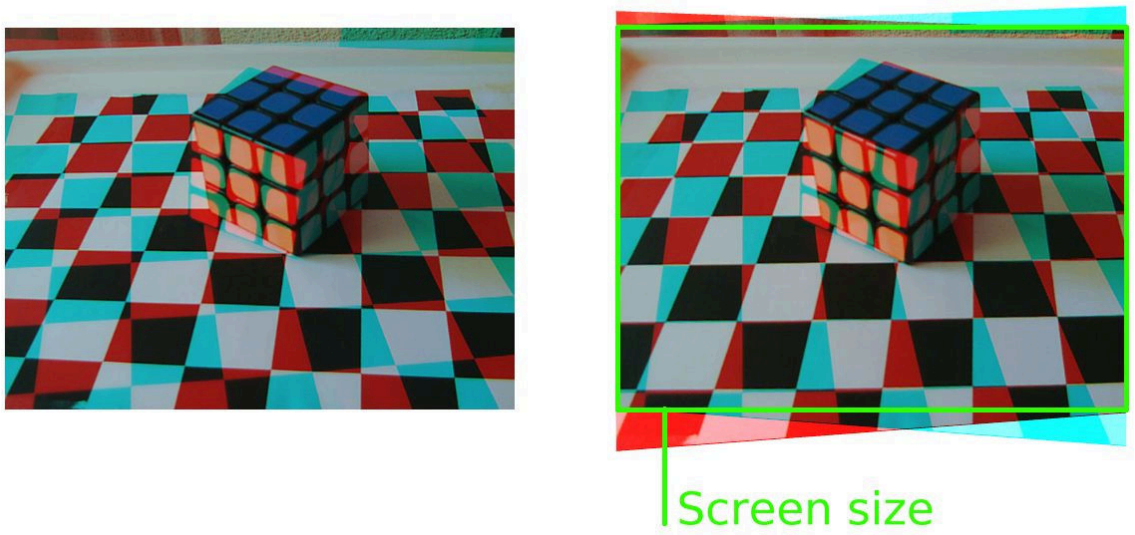


Figura 3.34: Muestra del keystone corregido

3.5.2. Interfaz del módulo GvBackground

El control de los planos de fondo se realiza en la pestaña Background de la interfaz. A continuación se detalla el funcionamiento de los controles.

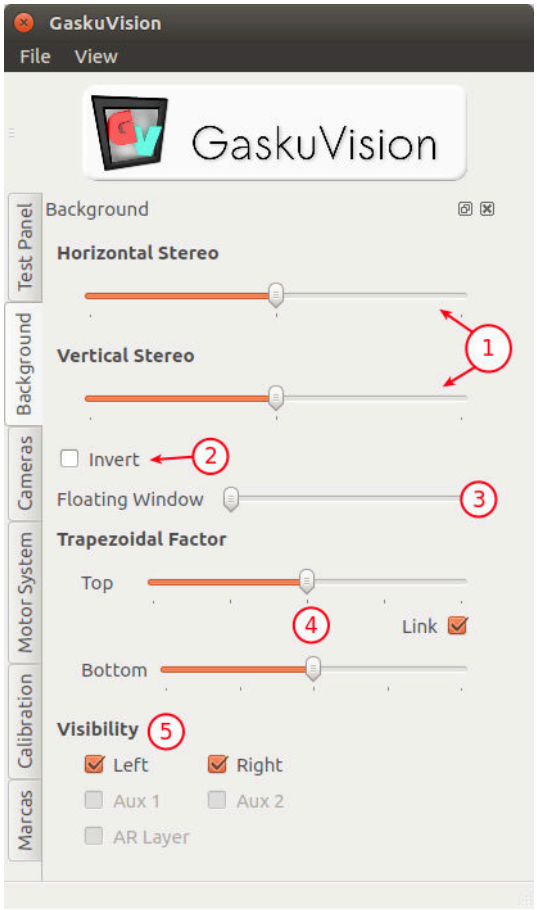
1. **Desplazamiento de los fondos para ajuste estereoscópico.**
   Permite corregir el desplazamiento de las imágenes izquierda y derecha tanto horizontal como verticalmente.

2. **Inversión estereoscópica.** Invierte el fondo de las imágenes izquierda y derecha.

3. **Tamaño de los márgenes.** Añade dos márgenes a ambos lados con el fin de ocluir partes de la imagen donde hay información que no se encuentra en ambas vistas. Esto puede ocurrir de manera natural o por el ajuste horizontal del fondo.

4. **Correción del keystone.** Deforma los planos para corregir paralajes verticales provocado por el efecto keystone. Pueden ajustarse de manera independiente los bordes inferior y superior desmarcando la opción "*Link*".

5. **Visibilidad de los fondos.** Permite controlar la visibilidad de las distintas cámaras de manera independiente.



3.6. Integración de las sombras

Uno de los requisitos era la implementación de sombras translúcidas (R08: apartado 2.1 en la página 17), de tal manera que los objetos virtuales pudieran proyectar sombras sobre planos definidos por los objetos reales. La idea de diseño era que cada nodo correspondiente a un objeto virtual contuviera un plano transparente que recibiese la sombra de dicho objeto virtual.

Tras varios intentos de implementación fallidos, probando con distintos tipos de materiales de Ogre, se llego a la conclusión de que existía una limitación del motor que impedía que materiales declarados como transparentes recibieran sombras de otros

objetos. Dicha limitación se encuentra documentada en el propio manual de Ogre (*http://www.ogre3d.org/docs/manual/manual_14.html*).

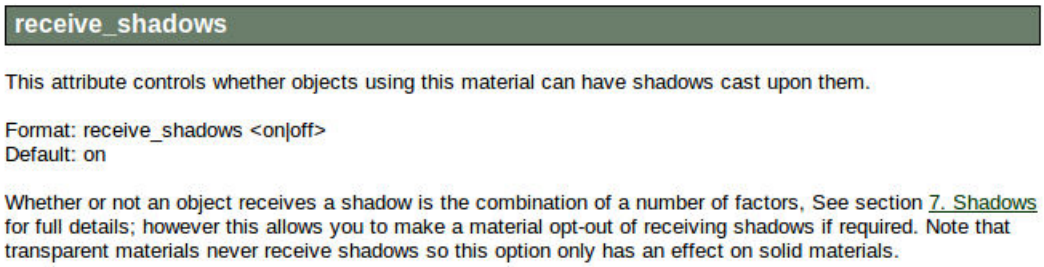


Figura 3.35: Captura del manual de OGRE

Se han implementado dos soluciones distintas, que pueden ser seleccionadas en los archivos de configuración. En ambas soluciones las sombras se aplican sobre un plano que se añade a los objetos en su inicialización, y que actúa como receptor de sombras. Este puede ser un plano, pero se ha diseñado un objeto que corresponde con la forma de las marcas, de tal manera que las sombras se limitan a las marcas físicas cuando éstas están en el aire sostenidas por el usuario (figura 3.36).

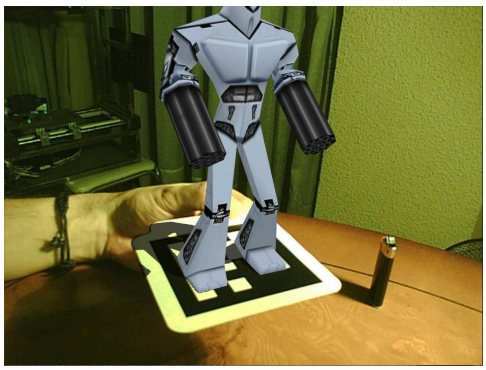


Figura 3.36: Ajuste de la sombra en la marca física

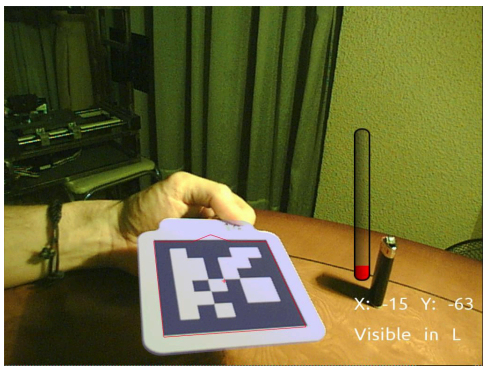


Figura 3.37: Modo depuración del objeto receptor

3.6.1. Implementación utilizando una técnica tipo croma

Tras leer la documentación de Ogre y diversos hilos del foro oficial, se decidió abordar el problema utilizando un Shader personalizado. La primera idea fue tratar el plano receptor como si de un croma se tratase. Una vez proyectada la sombra sobre el material del plano (el proceso normal del motor) se ejecuta un *píxel shader* (*fragment program*

en nomeclatura OpenGL) que analiza cada píxel de la textura. Si la intensidad del píxel es mayor que el umbral escogido (pasado como parámetro) significa que no pertenece a ninguna sombra, y por tanto la salida para dicho píxel será transparente (alpha = 0.0). Si la intensidad es menor la salida será un píxel negro con una transparencia mayor que cero (pasada también como parámetro).

El siguiente fragmento de código corresponde al material asignado a los receptores de sombras.

Listing 3.5: Declaración del material del plano receptor

```
material TransparentPlane
{
   technique
   {
      pass
      {
         scene_blend alpha_blend
         depth_write off

         fragment_program_ref MakeAlpha_PS {}
         texture_unit AlphaMap { content_type shadow }
      }
   }
}
```

El siguiente fragmento corresponde a la declaración del *fragment program*. En la última línea se puede observar el paso de los dos párametros (intensidad y umbral de las sombras) que serán recibidos por el *shader*.

Listing 3.6: Declaración del Shader de sombras

```
fragment_program MakeAlpha_PS cg
{
   source Shadows.cg
   entry_point main_MakeAlpha_fp
   profiles ps_1_1

   default_params { param_named shadowParams float2 0.1 0.9 }
}
```

Por último, la implementación del *shader*. Se hace una comparación de cada píxel de la textura (`tex2D(texture, uv0).r >shadowParams[1]`) y según el resultado la salida será un píxel transparente (`float4(1.0, 0.0, 0.0, 0.0)`) o un píxel translúcido con la transparencia definida por el parámetro (`float4(0.0, 0.0, 0.0, shadowParams[0]`).

Listing 3.7: Implementación del Shader de sombras

```
void main_MakeAlpha_fp(
   float2 uv0: TEXCOORD0, // UV interpolated for current pixel
   out float4 color: COLOR, // Output color we want to write
   uniform sampler2D texture, // Texture we're going to use
```

```
    uniform float2 shadowParams) // Shadow intensity and threshold
{
    color = tex2D(texture, uv0).r > shadowParams[1] ?
       float4(1.0, 0.0, 0.0, 0.0) : float4(0.0, 0.0, 0.0, shadowParams[0]);
}
```

Indicar que es posible pasar los parámetros desde el código, pudiendo así cambiar, por ejemplo, la intensidad de las sombras en tiempo de ejecución. El codigo a continuación muestra las instrucciones necesarias para ello.

Listing 3.8: Paso de parámetros desde el código

```
// prueba de paso de parametros al shader
    Ogre::MaterialPtr mat =
      Ogre::MaterialManager::getSingleton().getByName("TransparentPlane");
    Ogre::GpuProgramParametersSharedPtr params =
      mat->getTechnique(0)->getPass(0)->getFragmentProgramParameters();
    float pars[2] = {0.2f, 0.9f};
    params->setNamedConstant("shadowParams", pars, 2, 1);
```

Con este sistema el programa tan sólo ha de crear los planos de cada nodo de la escena que recibirán las sombras, asígnandoles el material **TransparentPlane**. La solución adoptada fue explicada y compartida en el foro oficial de Ogre [19].

Figura 3.38: Resultado de las sombras translúcidas.

3.6.2. Implementación aplicando la imágen de la cámara sobre el receptor de sombras

Posteriormente se decidió abordar el problema de una manera distinta, debido a ciertas limitaciones de la técnica anterior, especialmente la incapacidad de aplicar filtrado

en un futuro con el fin de obtener sombras más realistas.

La presente implementación aplica la textura obtenida de la cámara directamente sobre el objeto receptor de sombras, utilizando las coordenadas de pantalla en lugar de las coordenadas UV propias del objeto. Esto permite que el objeto se integre perfectamente con el fondo de las cámaras.

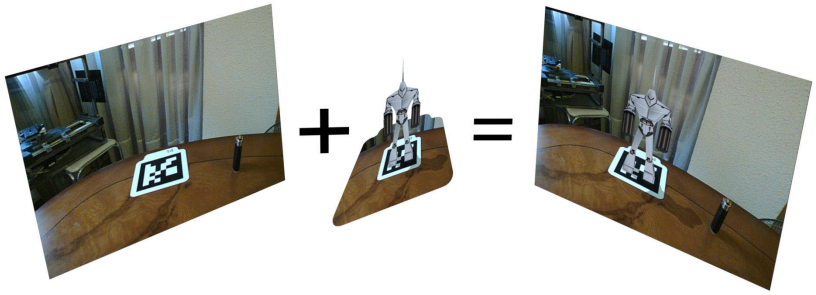


Figura 3.39: Composición de las sombras

Esta técnica no estuvo libre de complicaciones, sin embargo. Cada cámara debe aportar su contenido a el mismo plano, dependiendo de la vista que represente (izquierda, derecha, auxiliar...). Esto se consiguió utilizando una característica de Ogre, que permite aplicar banderas (*flags* en inglés) a diversos objetos, limitando su visibilidad en función de la vista que se esté renderizando. Además hay que adaptar las texturas cada vez que se cambia la resolución de la ventana, o se cambia la relación de aspecto de esta. Esto se hace pasando ciertos parámetros al *shader* que controla el material del plano (listado 3.9 en la página siguiente).

Listing 3.9: Shader del receptor de sombras

```
void main_TranspBackgr_fp(
float2 uv0                      : TEXCOORD0,
float4 wpos                     : WPOS,
out float4 color                : COLOR,
uniform sampler2D backgTex      : TEXUNIT0,
uniform float2 screensize,
uniform float keepAspectRatio,
uniform float mirror,
uniform float aspectRatio)

{
float2 screencoords;
float ratio = screensize.x / screensize.y;

if (keepAspectRatio > 0)
{
        if (ratio > aspectRatio)
        {
                float nWidth = screensize.y * aspectRatio;
                float offset = 0;//(screensize.x - nWidth) / (2 * nWidth);
                screencoords = float2(wpos.x/nWidth - offset, wpos.y/screensize
                    .y);
        }
        else
        {
                float nHeight = screensize.x / aspectRatio;
                float offset = 0;//(screensize.y - nHeight) / (2 * nHeight);
                screencoords = float2(wpos.x/screensize.x,wpos.y/nHeight -
                    offset);
        }
}
else screencoords.xy = wpos.xy / screensize.xy;

color =  tex2D(backgTex, screencoords);
}
```

3.7. Aplicación de ruido al contenido sintético

Uno de los problemas al mezclar contenido sintético sobre imagen real es que los objetos virtuales destacan por estar demasiado definidos, tener colores demasiado saturados y su ausencia de ruido. Las cámaras reales por el contrario tienen cierta cantidad de ruido o grano, dependiente de la calidad, temperatura y sensibilidad del sensor (podemos ver un ejemplo en la figura 3.40 en la página siguiente). Para solventar este problema, se ha implementado un método para aplicar ruido al contenido virtual. Con el fin de aplicar el ruido solo al contenido virtual, la escena se renderiza en distintas capas. Aprovechando una característica de Ogre, que permite asignar a cada elemento una capa de renderizado (*RenderQueues*), asignamos al objeto 3D una capa superior, y al elemento receptor de sombras uno inferior, puesto que la textura asignada a este ya contiene ruido proveniente de la cámara. Posteriormente, se

compone la escena en el *compositor*, aplicando un *shader* de ruido (listado 3.10 en la página siguiente) solo a la capa que contiene a los objetos virtuales.

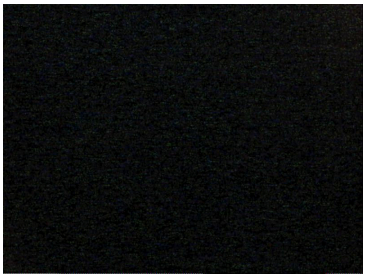


Figura 3.40: Muestra del ruido de la cámara PS3eye

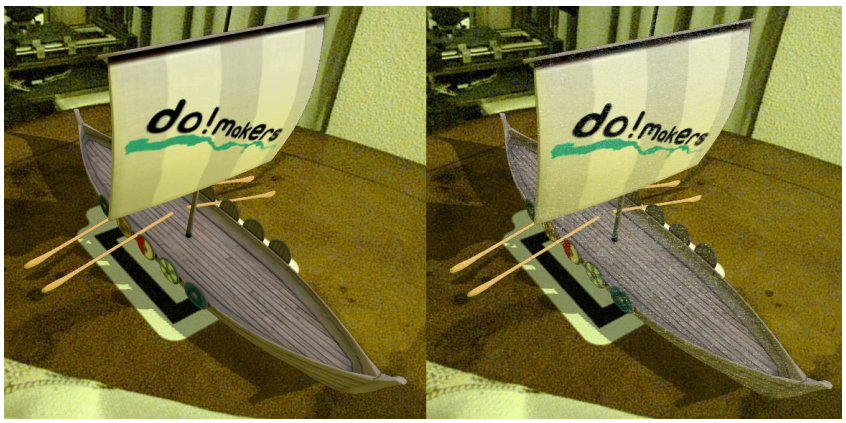


Figura 3.41: Comparación del efecto ruido

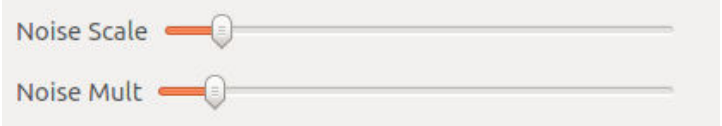


Figura 3.42: Configuración del ruido

Listing 3.10: Shader generador de ruido

```
sampler2D Image0: register(s0);
sampler2D Image1: register(s1);
sampler3D Rand: register(s2);

float4 Noise_ps(float4 posIn: POSITION, float2 img0: TEXCOORD0, float2 img1: TEXCOORD0,
uniform float time_0_X: register(c0),
uniform float noise_mult: register(c1),
uniform float noise_scale: register(c2)

) : COLOR {

        float4 image0 = tex2D(Image0, img0);
        float4 image1 = tex2D(Image1, img1);

        float4 rand = tex3D(Rand, float3(noise_scale * img0, time_0_X));
        rand.rgb = noise_mult * rand.rgb * (float3(1.0, 1.0, 1.0) - image1.rgb);

        float4 result = image1.a * (image1 + rand) + (1.0 - image1.a) * image0;
        return result;

}
```

3.8. SistemaMotores: comunicación y manejo del bastidor de cámaras

La librería SistemaMotores permite la comunicación con el bastidor estereoscópico, el movimiento separado o conjunto de los cuatro motores que modifican tanto la separación como la convergencia del par de cámaras.

La empresa 3+D Entertainment llegó a construir tres bastidores motorizados, dos de ellos con espejos, lo que permitía que la separación de las cámaras no estuviera limitada por las dimensiones de éstas. Sin embargo todas ellas son similares en cuanto a funcionamiento, ya que todas contienen los cuatro motores (dos para convergencia y dos para separación) y solo difieren en los sentidos en los que hay que moverlos. En esta memoria se muestra el bastidor VegasVision, de movimiento paralelo sin espejo.

3.8.1. El bastidor VegasVision

El bastidor consta de dos plataformas sobre raíles lineales, con dos motores que permiten la separación de las dos plataformas. Cada plataforma sostiene una zapata sobre un rodamiento que permite que la cámara rote sobre su eje vertical. Adicionalmente se instalaron unas cuñas que permiten un ajuste fino en los otros dos ejes de rotación, con el fin de corregir pequeños errores en la imagen estereoscópica

En la parte electrónica, el bastidor contiene una caja con los integrados que controlan los motores paso a paso y un Arduino para el control y comunicación con el ordenador. Permite la conexión de cuatro finales de carrera para resetear la máquina y evitar que se lleguen a los límites de los raíles.

Desde el punto de vista del protocolo, los cuatro motores se encuentran números para especificar el motor a mover. Se puede ver la numeración en la figura 3.43. El sentido positivo de los motores se detalla en la figura 3.44.

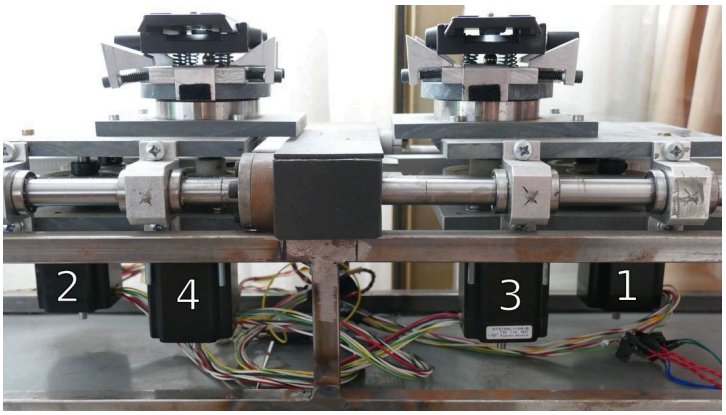


Figura 3.43: Vista frontal del bastidor, con los motores numerados.

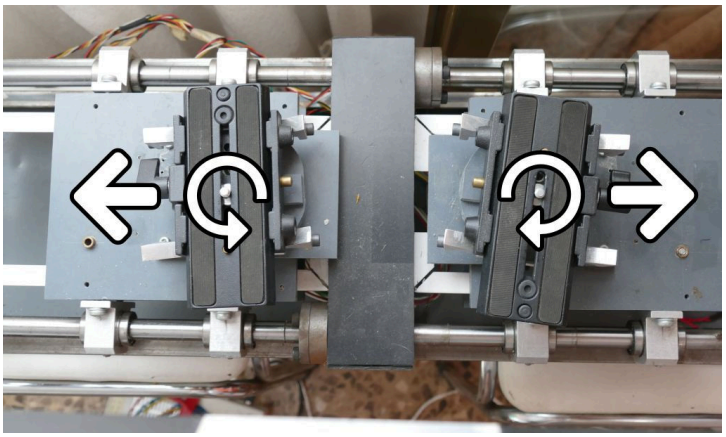


Figura 3.44: Vista cenital, con el movimiento que provocan los cuatro motores.

3.8.2. Protocolo de comunicación con Arduino

Para el manejo del bastidor se ha especificado un protocolo de comunicación por puerto serie. La aplicación, una vez conectada al Arduino, envia mensajes númericos que

representan movimientos de las cámaras. Se detalla el protocolo en el cuadro 3.1.

Valor	Significado	Longitud	Ejemplo
01	Operación mover	2	01
m	Número de motor	1	1
d	Dirección	1	1
longPasos	Longitud del parametro pasos	1	3
pasos	Cantidad de pasos	longPasos	320
longVelocidad	Longitud del parametro velocidad	1	4
velocidad	Velocidad del movimiento	longVelocidad	1200
0	Fin del mensaje	1	1

Cuadro 3.1: Formato de mensaje para el movimiento de un motor

Por ejemplo, el mensaje **"01213320412000"** indica un movimiento del motor 2 (desplazamiento de la cámara derecha), acercándose al eje central, 320 pasos a una velocidad de 1200. Las velocidades representan internamente una demora en microsegundos entre cada paso, y por tanto es el inverso de la velocidad. El Arduino responde con un mensaje que indica que el movimiento ha finalizado y está preparado para recibir otro mensaje. Este mensaje indica también la posición de los motores, teniendo como referencia la posición acumulada desde el último reseteo.

Es posible concatenar dos mensajes seguidos para lograr un movimiento en paralelo de dos motores a la vez. Por ejemplo, el mensaje **"013133204120041332041200 0"** hace que los dos motores de convergencia roten las cámaras de manera que acerca el plano de convergencia. El movimiento por defecto se efectúa de manera consecutiva; primero un motor y luego otro. Es posible operar dos motores de manera paralela activando el **modo paralelo**, enviando el mensaje **"14"** a la cámara. A partir de ese momento, todos los mensajes que indiquen el movimiento simultáneo de dos motores, hará que estos se muevan a la vez.

El mensaje **"00"** provoca un reseteo del bastidor, haciendo que todos los motores se muevan hasta que se activan los finales de carrera.

3.8.3. Interfaz de la pestaña SistemaMotores

Se pueden controlar los motores desde la pestaña *SistemaMotores* de la interfaz principal. A continuación se detalla el funcionamiento de cada elemento:

Bastidor a controlar.
Permite
elegir entre los dos bastidores soportados.

2. **Puerto**
serie para la conexión con el Arduino.

3. **Conectar, desconectar.**
Inicia o termina la conexión por puerto serie.

4. **Modo paralelo.**
Activa el
movimiento simultáneo de los motores, tanto para los movimientos manuales, como para la convergencia automática con marca fiducial.

5. **Movimiento de la cámara izquierda.**
Mueve la cámara en cada uno de los dos ejes (separación o rotación) y en los dos sentidos.

6. **Movimiento vinculado.**
Se mueven ambas cámaras con los controles de la cámara izquierda. Una vez realizado el ajuste inicial se recomienda activar esta opción.

7. **Número de pasos** que se mueven con cada pulsación.

8. **Movimiento de la cámara derecha.**
Se desactiva si la opción de movimiento vinculado se encuentra activada.

3.9. Creación del contenido 3D

Algunos de los programas de ejemplo requerían la creación de contenido 3D. El alumno tenía ciertos conocimientos en algunos programas libres para la creación de contenidos, y en otros tuvo que desarrollar los conocimientos necesarios.

Los programas utilizados fueron los siguientes:

- **Inkscape[14]:** editor de contenido vectorial. Fué utilizado para la creación de algunas texturas y para el diseño de las marcas fiduciales necesarias.
- **Gimp[11]:** programa para la creación y manipulación de imágenes. Se utilizó para la creación y edición de texturas.

Figura 3.45: Marcas fiduciales y botes de graffiti.

- **Blender[5]:** creacion de contenido gráfico 3D. Con el se crearon los diversos objetos que aparecen en las aplicaciones, exceptuando el personaje del ogro que forma parte de la librería Ogre.

3.10. GvProfiler: medidas de rendimiento

Para poder efectuar medidas de rendimiento y tiempos, se requería utilizar alguna librería de análisis de rendimiento. Las librerias Boost [6], incluyen un módulo de *profiling*. Sin embargo se encontró una librería (*http://www.cdiggins.com/profiler/*) que ampliaba a la anterior, mejorando la resolución en Windows, y recurriendo a la librería original para otros sistemas operativos. Esta mejora la logra midiendo directamente ciclos de reloj del procesador, con lo que se logra una precisión del orden de nanosegundos.

El alumno, además, modificó la librería para poder almacenar medias, máximos y mínimos. De esta manera se pueden extraer medidas muy precisas de cualquier bloque de código.

Un ejemplo de la salida se muestra en el listado 3.11 con tiempo recogidos de distintas funciones críticas.

Listing 3.11: Ejemplo de salida del profiler

```
profile name     total elapsed   count           min             max             average
blit()           0.197637        49              0.003882        0.004563        0.004033
doTracking()     0.803417        49              0.013292        0.018398        0.016396
grabFrame()      0.626337        49              0.003723        0.416929        0.012782
update()         0.206348        169             0.000019        0.004667        0.001221
updateNodes()    0.000214        49              0.000003        0.000005        0.000004
```

Capítulo 4

Conclusiones y posibles mejoras

Contenidos

Capítulo dedicado a valorar el desarrollo y posibles aplicaciones del proyecto, así como de dar una perspectiva de hacia donde podría ampliarse.

4.1. Mejoras gráficas

En esta sección se describen posibles mejoras que se han ido detectando durante el desarrollo y que mejorarían ciertos aspectos en la visualización.

4.1.1. Mejorar la estabilidad del tracking

Actualmente los objetos virtuales presentan algo de temblor (*jitter*), especialmente cuando la marca se encuentra excesivamente perpendicular o lejana con respecto a la cámara. Esto es debido al propio algoritmo de ARToolKit, pese a que ya se tiene en cuenta la pose anterior para el cálculo. Teniendo en cuenta que pueden existir marcas cuya posición sea constante (cuando están sobre una mesa o en la pared), se puede utilizar esta información para minimizar el error en el posicionamiento de la cámara. Se podría analizar la posición de las marcas en los últimos cuadros con el fin detectar cuándo una marca se considera estática, y de esta manera disminuir el peso que tienen las nuevas posiciones analizadas en el cálculo de la posición final. La implementación de esta mejora podría aprovechar los parámetros de elasticidad que ya se utilizan de manera manual apartado 3.1.3 en la página 47.

4.1.2. Subdivisión proyectiva del plano de imagen

En la solución propuesta la corrección del *keystone* se realiza deformando un plano subdividido utilizando una transformación bilineal. Es decir, la subdivisión de cada lado del cuadrilátero se hace por interpolación lineal. Esto no es del todo correcto, se puede comprobar observando las líneas qué forman la malla, que no son rectas (figura 4.1 en la página siguiente). Esto provoca cierta deformación de la estereoscopía, aunque en correcciones pequeñas es despreciable. No obstante, lo correcto sería aplicar una transformación proyectiva a dicho plano de manera análoga a cómo se corrige la imagen de los proyectores. Observando la comparación de ambas transformaciones se pueden comprobar las diferencias (figura 4.2 en la página siguiente).

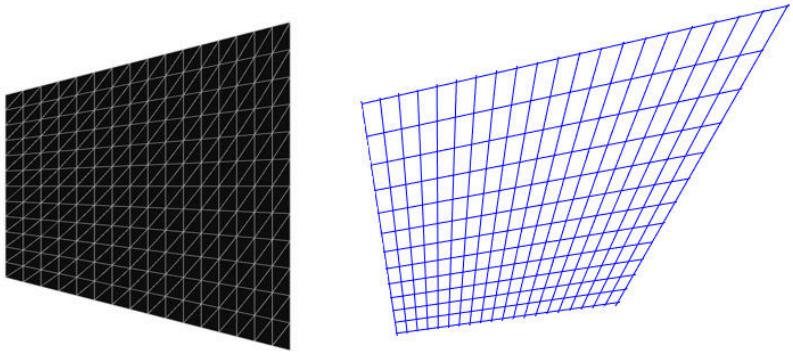


Figura 4.1: Solución implementada (izquierda) y subdivisión óptima (derecha).

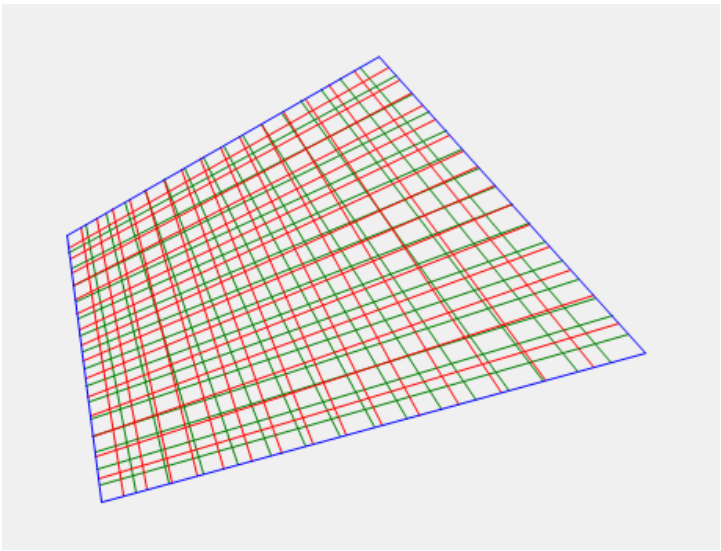


Figura 4.2: Diferencia entre los dos tipos de subdivisiones: bilineal en verde, proyectiva en rojo.

4.1.3. Distorsión de la imagen

En la versión actual la coherencia entre la imagen real y la sintética no es perfecta utilizando los parámetros extrínsecos obtenidos en la calibración. Una de las razones es que no se corrige la distorsión de la imagen recibida provocada por las lentes. La librería ARToolKit usa una función que corrige la distorsión, pero solo la aplica para aquellos píxeles que necesita para el cálculo de las marcas.

Para corregir la distorsión de toda la imagen se puede utilizar la librería **OpenCV**, que dispone de una función a tal efecto. Pero debido a que dicha imagen se utiliza para ser renderizada como fondo, se podría utilizar la capacidad de la tarjeta gráfica para

distorsionar texturas, y de esta manera mejorar el rendimiento. Una solución puede ser subdividir el plano de fondo y mover los vértices de acuerdo con los parámetros de distorsión obtenidos de la calibración. Mejor todavía es aplicar un shader a la textura de fondo para corregir la distorsión.

En el caso de que se quiera conservar la imagen intacta (por ejemplo si se quiere mezclar la imagen sintética con la imagen original proveniente de las cámaras en una mesa de mezclas externa) se puede aplicar la misma distorsión detectada en la calibración sobre el render final. Debido a que la cámara virtual utilizada en los motores gráficos solo es capaz de aplicar transformaciones lineales, se deberá aplicar un shader que efectúe las correspondientes distorsiones radial y tangencial.

Las correcciones de este apartado y el anterior podrían combinarse en un mismo shader, ya que se actúa sobre los mismos elementos.

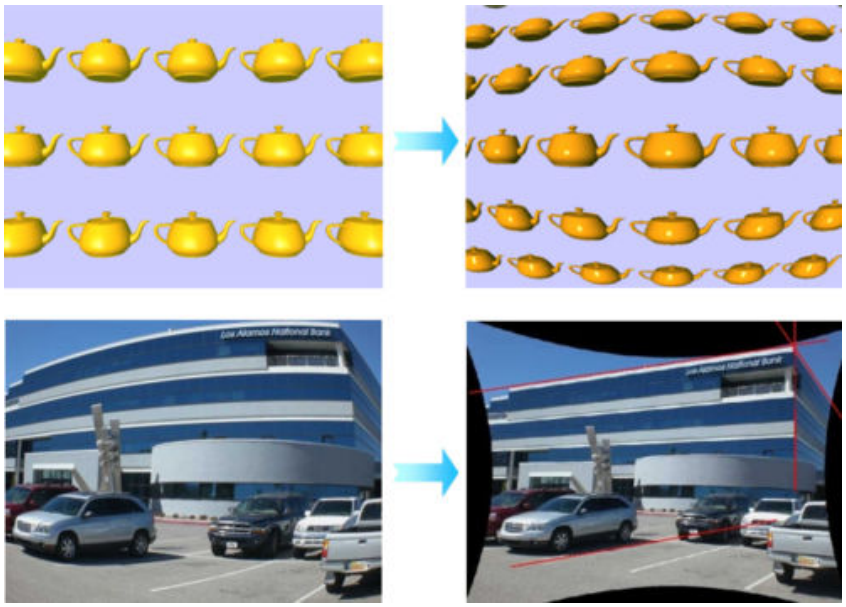


Figura 4.3: Ejemplo de distorsion y desdistorsión.

4.1.4. Mejora del anaglifo

En este apartado se hace una comparativa de los distintos métodos de mezcla de los canales para crear el anaglifo. Las fórmulas descritas sirven para calcular el valor RGB del pixel anaglifo (r_a, g_a, b_a) a partir de los valores RGB de la imagen izquierda (r_1, g_1, b_1) y de la imagen derecha (r_2, g_2, b_2). Los datos se han extraído de la web 3dtv.at [3]. Cabe recordar que la luminancia de un píxel se obtiene con la formula Y = 0,299R

+ 0,587G + 0,114B, de ahí que se utilicen estos valores en algunos de los modos de mezclado.

- **Anaglifo verdadero:** produce una imagen oscura, con poca fidelidad de color, pero muy poco *ghosting*.

$$\begin{bmatrix} r_a \\ g_a \\ b_a \end{bmatrix} = \begin{bmatrix} 0{,}299 & 0{,}587 & 0{,}114 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} r_1 \\ g_1 \\ b_1 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0{,}299 & 0{,}587 & 0{,}114 \end{bmatrix} \begin{bmatrix} r_2 \\ g_2 \\ b_2 \end{bmatrix}$$

Figura 4.4: Anaglifo verdadero.

- **Anaglifo gris:** produce una imagen con poca fidelidad de color y algo más de *ghosting* que el anterior.

$$\begin{bmatrix} r_a \\ g_a \\ b_a \end{bmatrix} = \begin{bmatrix} 0{,}299 & 0{,}587 & 0{,}114 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} r_1 \\ g_1 \\ b_1 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0{,}299 & 0{,}587 & 0{,}114 \\ 0{,}299 & 0{,}587 & 0{,}114 \end{bmatrix} \begin{bmatrix} r_2 \\ g_2 \\ b_2 \end{bmatrix}$$

Figura 4.5: Anaglifo gris.

- **Anaglifo de color:** produce una imagen con más de fidelidad de color pero genera rivalidad retiniana. En este caso hablamos de rivalidad retiniana en luminancia, ya que siempre existirá

$$\begin{bmatrix} r_a \\ g_a \\ b_a \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} r_1 \\ g_1 \\ b_1 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_2 \\ g_2 \\ b_2 \end{bmatrix}$$

Figura 4.6: Anaglifo de color.

- **Anaglifo de color parcial:** produce una imagen con menos fidelidad de color que el anterior modo pero con menos rivalidad retiniana.

$$\begin{bmatrix} r_a \\ g_a \\ b_a \end{bmatrix} = \begin{bmatrix} 0{,}299 & 0{,}587 & 0{,}114 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} r_1 \\ g_1 \\ b_1 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_2 \\ g_2 \\ b_2 \end{bmatrix}$$

Figura 4.7: Anaglifo de color parcial.

- **Anaglifo optimizado:** en este modo no se tiene en cuenta el canal rojo de las imágenes de entrada. Posteriormente al mezclado se debe aplicar una corrección gamma al canal rojo para hacerlo más brillante (valor 1.5). Produce imágenes con una moderada fidelidad de color (excepto en el rojo), y prácticamente sin rivalidad retiniana.

$$\begin{bmatrix} r_a \\ g_a \\ b_a \end{bmatrix} = \begin{bmatrix} 0 & 0{,}7 & 0{,}3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} r_1 \\ g_1 \\ b_1 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_2 \\ g_2 \\ b_2 \end{bmatrix}$$

Figura 4.8: Anaglifo optimizado.

4.1.5. Calidad de las sombras

Las sombras implementadas presentan una serie de defectos debidos a la técnica utilizada basada en texturas. Presentan una notable pixelación que se nota sobretodo cuando están cercanas a la cámara. Ademas por el método de proyección utilizado se llegan a proyectar sobre otros planos incluso cuando estos están entre la luz y el objeto que causa las sombras.

Existen soluciones más complejas que mejoran la calidad de las sombras, como por ejemplo la técnica **depth shadow mapping**, que utiliza un mapa de profundidad para evitar los defectos de proyección. En este enlace se detalla una posible implementación: (`http://www.ogre3d.org/tikiwiki/Depth+Shadow+Mapping&structure=Cookbook`).

Además se pueden aplicar *shaders* específicos que aplican un desenfocado de las sombras segun la distancia al objeto que las causa. Incluso existen implementaciones de *ambient oclussion*. En este hilo se expone una implementación de esta técnica: (`http://www.ogre3d.org/forums/viewtopic.php?f=11&t=47927`).

4.2. Mejoras de la usabilidad

4.2.1. Incorporación de metadatos al archivo de escena

En la versión actual la configuración de las marcas, sus objetos asociados y su comportamiento se definen mediante código, necesitando compilar cualquier cambio en el funcionamiento de la aplicación. Todos estos aspectos deberían definirse en archivos de configuración, de tal manera que cualquier usuario pueda añadir esta información sin necesidad de acceso al código. En el archivo de escena tan solo se enumeran los objetos virtuales que tiene que cargar el motor gráfico. Las últimas versiones del cargador de escenas de Ogre permite añadir información definida por el usuario. Se podría añadir información referente a las marcas (número de marca, ancho del receptor de sombra, comportamiento de la marca...), incluso la información propia de las cámaras y configuraciones generales. De tal manera, toda la configuración de la aplicación estaría contenida en un mismo archivo.

4.2.2. Diálogo de configuración de las cámaras

Actualmente solo se muestra el diálogo inicial de selección de cámara cuándo la cámara principal (izquierda) no está correctamente configurada. Sin embargo no es posible cambiar su configuración una vez iniciada la aplicación. En el caso de más de una cámara, todas ellas deben ser configuradas correctamente a través de los archivos de configuración. Debería ser posible añadir nuevas cámaras, cambiar sus parámetros o su función a través de un menú diseñado a tal efecto.

4.2.3. Nuevos tipos de marcas

Las versiones más recientes de ARToolKit soportan nuevos modos de seguimiento de marcas, destacando el uso de imágenes como marcas (Natural Feature Tracking, figura 4.9 en la página siguiente). La implementación de esta característica abriría nuevas posibilidades en el diseño de aplicaciones interactivas. Para ello habría que migrar el proyecto a la última versión de esta librería.

4.2.4. Mejoras en la calibración estereoscópica

Teniendo un buen modelo de la construcción física del bastidor y realizando una correcta inicialización de los motores sería posible conocer la separación física y la convergencia de las cámaras, o al menos una aproximación suficiente para la estimación

Figura 4.9: Natural feature tracking.

del paralaje sin tener que recurrir a la calibración estéreo. Esto permitiría tener una retroalimentación más directa de las distancias a las que podemos grabar para conseguir una esteroscopía confortable.

4.2.5. Soporte de nuevas plataformas

El diseño inicial contemplaba como plataformas soportadas los entornos Windows y Linux. Pero debido al auge de las plataformas móviles y web en los últimos años, el alcance de cualquier solución que utilice este se ve muy limitado. Casi todas las librerías utilizadas ya disponen de versiones móviles. Pero no hay que menospreciar el esfuerzo necesario de migrar todo el proyecto a estas plataformas. Además hay que tener en cuenta la limitación de prestaciones de los dispositivos a los que se daría soporte. Cabe valorar por lo tanto si sería más conveniente volver al diseño inicial si se quiere llevar a cabo esta mejora.

4.3. Valoración personal del proyecto

Pese a la demora en la presentación del presente proyecto, cabe destacar su importancia en la formación del alumno. Todo lo aprendido sobre las distintas áreas del conocimiento que toca (realidad aumentada, informática gráfica, estereoscopía, diseño 3D, electrónica y Arduino...) tuvo una aplicación a posteriori en el desarrollo profesional del alumno.

En un primer momento el propio proyecto se utilizó de manera satisfactoria en

producciones audiovisuales o instalaciones interactivas que supusieron una enriquecedora experiencia. Además el conocimiento adquirido ha sido importante en otros proyectos de investigación en los que el alumno ha estado presente.

Cabe destacar la importancia que tuvo como primer acercamiento a la electrónica de prototipado, y a la plataforma Arduino en particular. Este conocimiento es una de las principales fortalezas del alumno en el sector educativo y de divulgación, área en la que se desarrolla profesionalmente en la actualidad.

Apéndice A

Estudio del arte

Contenidos

En este anexo se da una visión general de la estereoscopía clásica, así como algunos problemas asociados a las técnicas estereoscópicas. Además se exponen ciertos conceptos de geometría proyectiva relevantes al proyecto.

A.1. Estereoscopía

La estereoscopía, imagen estereográfica, o imagen 3D (tridimensional) es cualquier técnica capaz de recoger información visual tridimensional o de crear la ilusión de profundidad en una imagen. La ilusión de la profundidad en una fotografía, película, u otra imagen bidimensional es creada presentando una imagen ligeramente diferente para cada ojo, como ocurre en nuestra forma habitual de recoger la realidad. [10]

Uno de los requisitos del proyecto era familiarizarse con la estereoscopía, estudiar las bases teóricas en las que se basa y conocer los diversos métodos de grabación y reproducción.

Existen multitud de recursos disponibles, tanto libros como artículos en Internet. Se puede destacar el libro **3D Movie Making**[1], donde el autor, **Bernard Mendiburu**, explica de manera clara las bases y aplicación de la estereoscopía.

La visión estereoscópica humana es un complejo sistema que permite extraer información de profundidad de los objetos del entorno. El sistema de visión binocular humano, permite a los ojos converger y enfocar el objeto del que se desea extraer información. La convergencia se consigue rotando los ojos mientras que el enfoque, también llamado acomodación, se consigue curvando el cristalino. Las imágenes son posteriormente analizadas por el cerebro, utilizando la disparidad o el paralelismo de cada punto para deducir su lejanía con respecto a los ojos.

De manera análoga, es posible utilizar dos cámaras situadas bien en paralelo, bien perpendiculares con la ayuda de un espejo, consiguiendo dos señales de vídeo que puedan ser posteriormente visualizadas estereoscópicamente. Habrá que elegir por tanto, el ángulo entre ellas, para conseguir la convergencia deseada. Además hay que tener en cuenta que mientras que la distancia interocular es parecida en casi todos los humanos (en torno a los 6,5 cm.), la distancia entre las cámaras puede ser mayor o menor, ya sea por elección del encargado de la grabación, o por limitaciones técnicas. Esto puede provocar que el tamaño percibido de los objetos sea mayor o menor del propio de los objetos, efecto conocido como *hiperestéreo* y *hipoestéreo* respectivamente.

Para visualizar vídeos estereoscópicos existen diferentes tecnologías desarrolladas, utilizando en casi todas ellas gafas especiales, que son capaces de separar de nuevo cada una de las dos imágenes, de tal manera que cada ojo vea su fotograma correspondiente.

Existen sin embargo dificultades a tener en cuenta al realizar estas grabaciones.

Primeramente, en la visión natural, el punto al que se converge y el punto al que se enfoca es el mismo. Pero cualquiera de los sistemas de visualización basados en pantallas, el plano de acomodación o enfoque es siempre la pantalla, mientras que la convergencia se efectúa al punto elegido por el espectador. Esta diferencia resulta en cierto malestar que puede ser minimizado con el paso del tiempo. Otro problema es que según el plano escogido, se le pueden ofrecer al espectador objetos borrosos en el plano de enfoque, y si éste centra su atención en estos objetos, su cerebro notará que esos objetos deberían estar enfocados y no lo están. De manera análoga, objetos que de normal estarían desenfocados se pueden mostrar enfocados, y si el paralaje horizontal que muestran es superior a un límite determinado, provocara malestar, efecto conocido como divergencia.

Otros problemas que se pueden dar son paralajes verticales por un posicionamiento defectuoso de las cámaras o por la diferencia de los prismas de perspectiva de cada cámara (efecto *keystone*), problemas de *goshting* provocados por el método de visualización, o problemas derivados de fallos de sincronización entre las imágenes izquierda y derecha.

A.1.1. Percepción de la profundidad

Nuestra habilidad para combinar las dos imágenes que nuestro cerebro recibe de nuestros ojos, para percibir profundidad, se denomina **estereopsis**. Sin embargo, esta es solamente una de las formas en las que recibimos información de profundidad del mundo que nos rodea. Algunas personas que, por alguna discapacidad no ven por alguno de sus ojos, pueden sin embargo estimar la distancia a la que se encuentran los objetos que les rodean. Esto es debido a que existe mucha información de profundidad en una simple imagen 2D, si no de manera explícita, si en base a la experiencia previamente adquirida por el cerebro.

Pistas monoscópicas

Conjunto de pistas de profundidad que podemos extraer con la señal recibida por un solo ojo. Casi todas, excepto la acomodación ocular, son de carácter puramente psicológico, interpretadas por el cerebro a partir exclusivamente de la señal ocular.

- **Perspectiva y tamaño relativo:** El efecto de perspectiva produce una clara sensación de profundidad. Dos líneas paralelas convergen conforme aumenta la distancia al punto de vista. Además si dos objetos que percibimos de igual tamaño, están a diferente distancia del punto de fuga, podemos deducir que uno es mayor que el otro.

Figura A.1: Perspectiva y tamaño relativo

- **Gradiente de una textura:** Si una textura contiene un patrón repetitivo, este parecerá aumentar de frecuencia según nos acerquemos al horizonte.

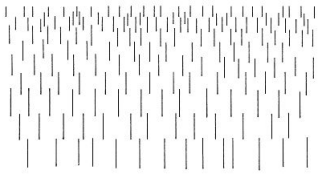


Figura A.2: Gradiente de textura

- **Oclusión parcial:** Se da cuando un objeto ocluye parcialmente a otro. El cerebro es capaz de deducir que el objeto ocluido se encuentra más lejos.

Figura A.3: Oclusión parcial

- **Perspectiva atmosférica y saturación:** Se refiere al efecto que produce la atmósfera sobre los objetos en relación a la distancia al observador. Afecta al contraste del objeto con respecto a objetos cercanos, disminuye la saturación de los colores y puede producir una variación de los colores (hacia el azul o el rojo, dependiendo de la hora del día).
- **Luces y sombras:** Existen multitud de pistas relacionadas con las luces implicadas en la escena y las sombras que generan sobre los objetos. Para luces artificiales, relativamente cercanas a los objetos, podemos extraer la profundidad de estos por el ángulo que forman sus sombras sobre el suelo. Las sombras además nos dan información sobre la forma del objeto en la dimensión perdida en la proyección. El reflejo de la luz sobre el objeto o reflejo especular también nos da pistas sobre su forma.

Figura A.4: Perspectiva atmosférica y saturación

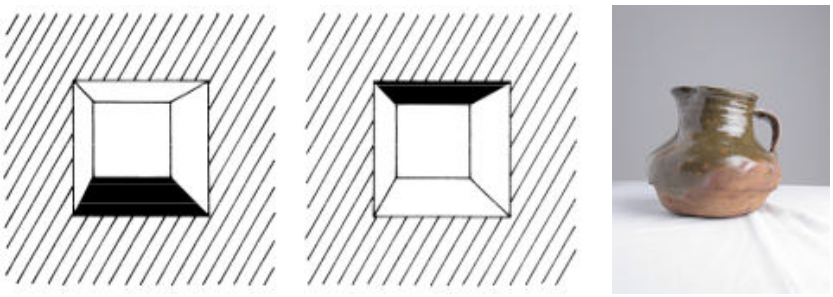


Figura A.5: Luces y sombras

- **Conocimiento previo del objeto:** Si tenemos un conocimiento previo del objeto, recordaremos su forma, pudiendo deducir información sobre su volumen u orientación con respecto al punto de vista.

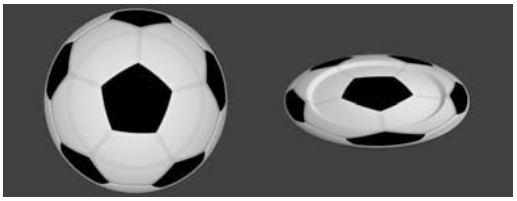


Figura A.6: Conocimiento previo del objeto

- **Posición relativa al horizonte:** Si podemos observar el horizonte en nuestro campo de visión, podremos deducir que los objetos más cercanos a este serán más lejanos. Este efecto está relacionado con el efecto de perspectiva.
- **Acomodación:** Se denomina acomodación al aumento de la potencia refractiva del cristalino que permite al ojo enfocar objetos cercanos. Este fenómeno se produce debido a que, en su estado relajado, el ojo está preparado para enfocar objetos lejanos. El aumento de potencia se consigue mediante un incremento de su espesor y

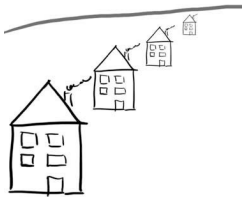


Figura A.7: Posición relativa al horizonte

de la curvatura de las superficies del cristalino, gracias a la contracción del músculo ciliar.[28, Wikipedia]

Este movimiento muscular provee una retroalimentación al cerebro, que es capaz de deducir la distancia al objeto enfocado. Tiene carácter tanto psicológico como fisiológico.

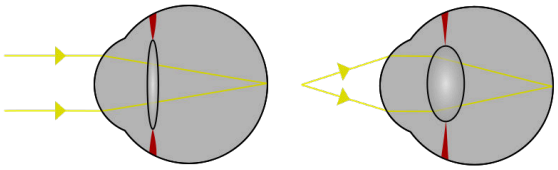


Figura A.8: Acomodación

Pistas relacionadas con el movimiento

El movimiento de la cámara con respecto al tiempo o el movimiento de los mismos objetos, proporciona al cerebro pistas para deducir su posición en el espacio. A continuación se detallan algunas de estas pistas.

- **Paralaje inducido por el movimiento del punto de vista:** Un movimiento lateral del punto de vista entre dos imágenes separadas en el tiempo provocará un paralaje (desviación angular de la posición aparente de un objeto, dependiendo del punto de vista elegido) en los objetos. Este paralaje será menor cuanto más lejano se encuentre el objeto.

  Un buen ejemplo consiste en nuestra observación de un paisaje desde la ventanilla de un tren. Podremos observar claramente como los objetos cercanos parecen moverse más rápido mientras que los objetos cercanos al horizonte apenas cambian de posición. Algunos animales que carecen de estereopsis (debido a la situación lateral de los ojos en la cabeza) utilizan este método para deducir la distancia de los objetos. Es el caso de las palomas, que hacen rápidos movimientos verticales de la cabeza para extraer pistas sobre la distancia a otros objetos.

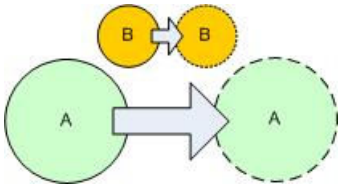


Figura A.9: Paralaje inducido por el movimiento del punto de vista

- **Paralaje inducido por el movimiento de los objetos:** Un conjunto de objetos, moviéndose a una velocidad uniforme entre ellos, provocarán un paralaje desigual en función de la distancia al punto de vista. Así, en una bandada de pájaros o en varios aviones volando en formación, aquellos más cercanos a nosotros parecerán moverse a más velocidad.

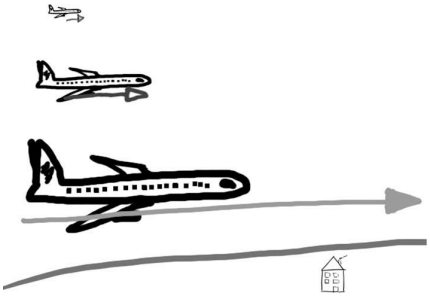


Figura A.10: Paralaje inducido por el movimiento de los objetos

Pistas estereoscópicas

Las pistas estereoscópicas tienen su origen en la visión binocular de nuestro sistema visual. Son extraídas a partir de la disparidad retinal entre las dos imágenes que recibe el cerebro. Debido al procesamiento de las dos imágenes simultaneamente por parte de neuronas especializadas situadas en el cortex visual podemos extraer información más detallada sobre la profundidad de los objetos.

- **Convergencia:** Las lentes de cada ojo proyecta una imágen de los objetos en cada retina. Para que estos objetos puedan ser vistos como una única imágen por el cerebro, la porción central de cada retina debe contener el mismo punto del objeto. Este punto llamado fovea es la zona donde percibimos un mayor nivel de detalle. Podemos observar este efecto si miramos a uno de nuestros dedos y luego convergemos a un punto más lejano, veremos como nuestro dedo parece duplicado o borroso. Los músculos responsables de este movimiento de rotación de los globos oculares proporcionan información de distancia sobre el entorno.

- **Paralaje horizontal o disparidad:** Cuando convergemos sobre un punto del espacio, todos los objetos que se encuentren delante o detrás del plano definido por este punto (llamado plano de convergencia) serán vistos como imágenes dobles. Objetos detrás del plano tienen disparidad no cruzada o paralaje positivo, mientras que los objetos por delante tienen disparidad cruzada o paralaje negativo.

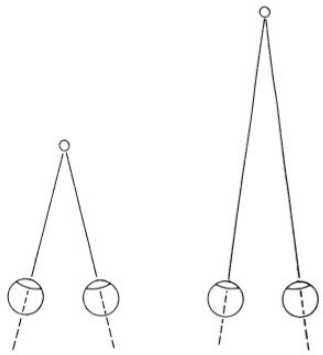


Figura A.11: Convergencia

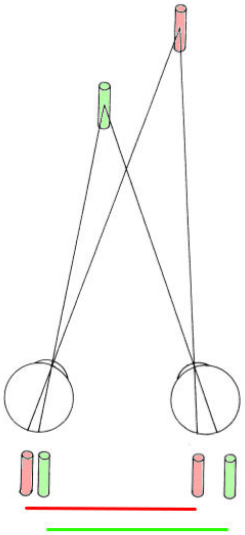


Figura A.12: Disparidad

A.2. Geometría proyectiva

En esta sección se detallan los parámetros relativos las cámaras, tanto físicas como su modelo matemático.

A.2.1. Introducción

Las imágenes digitales son producidas por una cámara que captura una escena, recibiendo la luz proyectada a través de una lente sobre un sensor. Esta proyección implica una relación entre la escena y su imagen 2D, y en el caso de escenas y objetos estáticos, una relación entre distintas imágenes de la misma escena. La geometría proyectiva es la disciplina que trata de describir y caracterizar en términos matemáticos el proceso de formación de la imagen.

A.2.2. Formación de la imagen y parámetros de cámara

En la fotografía tradicional, la imagen se forma por la luz recibida de los objetos, que tras pasar por una apertura frontal, es concentrada por una lente para finalmente ser recogida por un sensor situado detrás de la cámara.

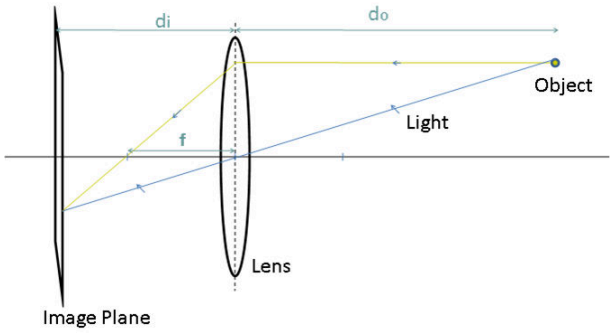


Figura A.13: Modelo de una cámara.

En este modelo, d_0 es la distancia desde la lente hasta el objeto observado, d_i la distancia desde la lente hasta el sensor, y f la distancia focal de la lente. La relación entre estos valores está descrita por la **ecuación de lentes delgadas**.

$$\frac{1}{f} = \frac{1}{d_0} + \frac{1}{d_i}$$

En computación gráfica, este modelo se simplifica. El uso de la lente no es necesario, por lo que se utiliza un modelo de cámara con una apertura infinitesimal. Además, puesto que en general $d_o >> d_i$, podemos asumir que el plano de la imagen se encuentra a la distancia focal. Finalmente, para evitar que la imagen se forme invertida, se posiciona el plano de la imagen delante de la lente. Esto, pese a ser imposible en la vida real, es matemáticamente equivalente. Este modelo simplificado recibe el nombre de **modelo pin-hole** o **cámara estenopeica**.

Según este modelo, y usando la semejanza de triángulos podemos derivar la equación proyectiva. Esta relación permite posicionar un punto de la escena en la imagen.

$$h_i = f\frac{h_o}{d_o}$$

Los parámetros esenciales de una cámara bajo este modelo son su distancia focal y el tamaño del plano de la imagen, que a su vez define el campo de visión (*field of view (FOV)*). Además, tratándose de imágenes digitales, es importante saber el número

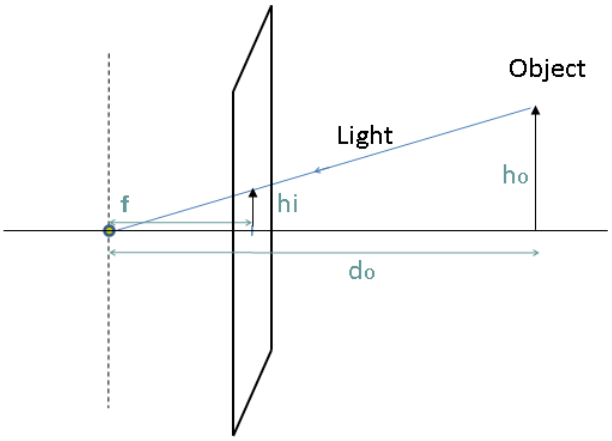


Figura A.14: Modelo pin-hole.

de píxeles de la imagen. Finalmente necesitamos saber el PUNTO PRINCIPAL. Este es la proyección perpendicular del punto focal sobre el plano de la imagen. Este punto no siempre se encuentra en el centro de la imagen en el caso de cámaras reales, debido a fallos de fabricación. Es por tanto un parámetro importante en el campo de la realidad aumentada.

Con estos parámetros podemos establecer la relación entre un punto 3D en la posición (X,Y,Z) y su imagen sobre la cámara (x,y), especificado en píxeles. Para ello añadimos al modelo unas coordenadas de referencia situadas en el centro de proyección.

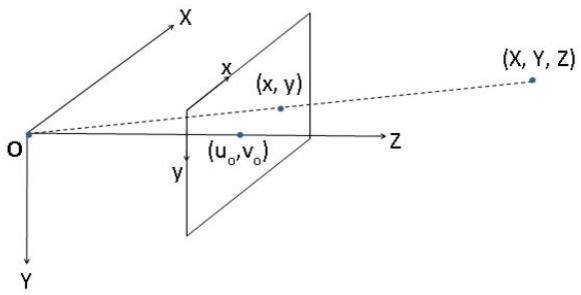


Figura A.15: Modelo pin-hole con referencia.

Utilizando las ecuaciones antes expuestas, podemos deducir que el punto (X,Y,Z) será proyectado sobre el plano de la imagen en el punto (fX/Z,fY/Z). Para pasar este punto a píxeles debemos dividir este punto por la anchura en píxeles (px) y la altura en píxeles (py) del plano de la imagen. Podemos obtener también la distancia focal f en píxeles horizontales (f_x) y en píxeles verticales (f_y), dividiéndola por estos valores. La ecuación proyectiva completa es:

$$x = \frac{f_x X}{Z} + u_0$$
$$y = \frac{f_y Y}{Z} + v_0$$

En esta ecuación, (u_0, v_0) denota el punto principal que es sumado al resultado para mover el origen. Estas ecuaciones pueden ser formuladas en forma matricial introduciendo COORDENADAS HOMOGÉNEAS en las cuales los puntos 2D se representan por vectores de tres elementos, y los puntos 3D por vectores de cuatro elementos.

$$s \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

A.2.3. Parámetros extrínsecos

La matriz extrínseca describe la posición de la cámara en coordenadas del mundo. Se puede descomponer en dos componentes: Una matriz de rotación R, y un vector de traslación t ([9])

$$[R \,|\, \boldsymbol{t}] = \left[\begin{array}{ccc|c} r_{1,1} & r_{1,2} & r_{1,3} & t_1 \\ r_{2,1} & r_{2,2} & r_{2,3} & t_2 \\ r_{3,1} & r_{3,2} & r_{3,3} & t_3 \end{array}\right]$$

Sin embargo es habitual añadir una última fila (0,0,0,1), que hace la matriz cuadrada y permite descomponer la matriz en una rotación seguida por una translación.

$$\left[\begin{array}{c|c} R & \boldsymbol{t} \\ \hline \mathbf{0} & 1 \end{array}\right] = \left[\begin{array}{c|c} I & \boldsymbol{t} \\ \hline \mathbf{0} & 1 \end{array}\right] \times \left[\begin{array}{c|c} R & \mathbf{0} \\ \hline \mathbf{0} & 1 \end{array}\right] \tag{A.1}$$

$$= \left[\begin{array}{ccc|c} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \times \left[\begin{array}{ccc|c} r_{1,1} & r_{1,2} & r_{1,3} & 0 \\ r_{2,1} & r_{2,2} & r_{2,3} & 0 \\ r_{3,1} & r_{3,2} & r_{3,3} & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{A.2}$$

Apéndice B

Motor gráfico: comparativa y arquitectura de Ogre3D

Contenidos

En este anexo se realiza una comparativa entre los diversos motores gráficos analizados, se exponen las razones de la elección de Ogre3D y se realiza una descripción detallada del motor escogido, sus características y arquitectura.

B.1. Comparativa de motores gráficos

La elección del motor gráfico está condicionada en gran parte por las características del proyecto y sus necesidades. Pese a tener puntos en común con un juego, como puede ser la gestión de recursos 3D, animaciones, materiales, luces y sombras, etc., no requiere, en principio, funcionalidades más elevadas como son la integración de audio, rutinas de inteligencia artificial, búsqueda de caminos, librerías de red, o editores avanzados para la gestión de estos elementos. Se tiene en cuenta, sin embargo, la posibilidad de necesitar algunos de estos elementos en un futuro, por ejemplo, si se deseara desarrollar una aplicación de realidad aumentada mucho más ambiciosa. Por tanto se valora positivamente que el motor disponga de estas funcionalidades, pero se busca en la medida de lo posible, que no tenga una arquitectura monolítica, y que opte por el uso de *plugins* o librerias externas.

Juegan un papel fundamental en la comparativa el tipo de licencias empleadas y las plataformas soportadas, valorándose muy positivamente las alternativas de código libre, así como su funcionamiento tanto en Windows como en Linux.

Para efectuar una elección lo más acertada posible, y tras una introducción más general de cada motor, se valoraran los siguientes aspectos, relacionados muchos de ellos con los reqisitos de la aplicación. Estos podrán ser después comparados en una tabla a modo de conclusión.

- Tipo de licencia.
- Plataformas soportadas. Funcionamiento en Windows y Linux.
- Sistemas graficos soportados.
- Soporte para shaders personalizados.
- Soporte para efectos de post-procesado final (para la mezcla estéreo).
- Capacidad de modificar las cámaras con matrices de proyección personalizadas (mezcla de imágen real y 3D).

B.1.1. Ogre3D

Ogre 3D [19] es quizás el motor libre más conocido y utilizado. Posee una arquitectura bien diseñada y muy bien documentada. Pese a no ser un motor de juegos completo, utiliza un sistema de *plugins* que puede extender las funcionalidades básicas del motor y que se integra bien con la arquitectura. Esto es así hasta el punto de que se deja el sistema de renderizado en manos de librerías externas, proporcionando una interfaz común para despues elegir si se desea utilizar DirectX, OpenGL, etc...

Figura B.1: Logo de Ogre3D

Debido a la comunidad de usuarios existente, muy activa y numerosa, existen numerosos *addons* disponibles, incluyendo varios gestores de escena (*scenegraphs*), sistemas de gestión de terrenos o sistemas de renderizado de sombras.

Utiliza el lenguaje C++, apoyándose en patrones de diseño para muchas áreas, dotándolo así de gran flexiblidad y facilidad de uso. Existen además *bindings* para otros lenguajes como Python, .NET, o Java.

Un punto en contra es la falta de herramientas de desarrollo orientado a juegos, por lo que queda en manos del usuario la elección de la cadena de producción. Sin embargo se pueden importar diversos formatos, ya sea de manera nativa, o por medio de librerías o *addons*. Implementa un lenguaje para scripts de materiales que permite la gestión de los recursos fuera de la aplicación.

Licencia	MIT (open source)
Plataformas	Windows, Linux, MacOS
API gráfica	DirectX y OpenGL
Shaders	Vertex y Pixel Shaders: ensamblador, Cg, HLSL, GLSL
Post-procesado	Compositor . Creación por medio de scripts
Cámaras personalizables	Si (Ogre::Frustum::setCustomProjectionMatrix)

B.1.2. Irrlicht

Irrlicht [15] es el segundo motor gráfico libre despues de Ogre3D. Irrlicht es un motor 3D gratuito y de código abierto, escrito en C++, el cual puede ser usado tanto en C++ como con lenguajes .Net.

Figura B.2: Logo de Irrlicht

Licencia	ZLib (open source)
Plataformas	Windows, Linux, MacOS, Solaris
API gráfica	DirectX y OpenGL
Shaders	Vertex y Pixel Shaders: Shaders 1.1 hasta 3.0, ARB, HLSL, GLSL
Post-procesado	No, aunque existe algun avanze al respecto en los foros oficiales
Cámaras personalizables	Si (irr::scene::ICameraSceneNode::setProjectionMatrix)

B.1.3. Panda3D

Panda3D [20] es un motor de juegos libre bajo la licencia BSD revisada. Pese a que el núcleo está escrito en C++ el lenguaje de de desarrollo es Python, lo que hace que sea muy fácil llegar a aplicaciones funcionales y aumenta mucho la productividad. Sin embargo tiene un rendimiento menor a otros motores, lo que puede llegar a ser un problema en aplicaciones con altos requerimientos gráficos.

Figura B.3: Logo de Panda3D

Un punto a favor de Panda3D es que esta más orientado a ser un motor de juegos completo. Trae incorporadas librerías para la gestión de audio, red, inteligencia artificial, motor físico, etc. Además hay disponibles herramientas oficiales para la creación de escenas, análisis de rendimiento

Posee características muy interesantes relacionadas con el proyecto, como es la inclusión de la librería ARToolKit, para el reconocimiento de marcas fiduciales, que liberaría al alumno de la labor de integración con el motor, pero que dificultaría posiblemente la opción de integrarlo de manera más eficiente por medio de hilos. Además tiene implementado un sistema de cámaras estéreo, para visualizar directamente la escena en anaglifo.

Licencia	Revised BSD (open source)
Plataformas	Windows, Linux, MacOS, FreeBSD
API gráfica	DirectX y OpenGL
Shaders	Vertex y Pixel Shaders: Cg, GLSL
Post-procesado	No, pero soporta renderizado estereoscópico de manera nativa
Cámaras personalizables	No se ha encontrado en la documentación

B.1.4. Unity3D

Unity [26] es un conjunto de herramientas de autor para la creación de juegos y animaciones 3D en tiempo real. Proporciona un entorno gráfico integrado como método principal de desarrollo, pudiendo ampliar las funcionalidades por medio de scripts. Estos pueden estar escritos en Mono (.NET), UnityScript (lenguaje propio inspirado en JavaScript), C# o Boo.

Figura B.4: Logo de Unity3D

Licencia	Unity (gratuito) y Unity Pro (comercial)
Plataformas	Windows, MacOS
API gráfica	Direct3D, OpenGL, OpenGL ES (iPhone OS), APIs propietarias (Wii)
Shaders	Cg, GLSL, lenguaje ShaderLab
Post-procesado	Si
Cámaras personalizables	Si (Camera.projectionMatrix)

B.2. Elección del motor gráfico

La elección del motor gráfico fue una decisión tomada después de decidir que el lenguaje a utilizar sería C++, con el apoyo de las librerías Qt. Así pues, y pese a que Panda3D tenía características muy interesantes en cuanto a renderización estéreo o el módulo ARToolKit integrado, se decidió optar por los motores que usaban C++ como lenguaje nativo. Además el rendimiento que se hubiera conseguido con Python hubiera estado por debajo del conseguido con C++.

También se descartó Unity3D, por ser software comercial. Pese a tener una version gratuita, podría verse cerrada una futura salida comercial de la aplicación por un cambio en la licencia de uso. Además no existiría la posibilidad de modificar el código en el caso de que se necesitara una carácteristica no implementada.

Entre los dos motores restantes, Ogre y Irrlicht, pese a contar con caracterísitacas y potencial semejantes, se optó por Ogre, debido a su estado de desarrollo más maduro y la gran base de usuarios existente, que garantizaba tanto la supervivencia del proyecto a largo plazo, como la posibilidad de encontrar recursos y ayuda en los foros oficiales.

Apéndice C

Entregables: compilación y configuración de la aplicación

Contenidos

Este anexo recopila la información útil para compilar el proyecto y configurar y ejecutar las distintas aplicaciones.

C.1. Estructura de los entregables

La carpeta superior del proyecto contiene todos los módulos y otras carpetas que se detallan a continuación:

- **bin**: Contiene los archivos de configuración para cada una de las tres aplicaciones entregadas. El archivo *cleye.config* solo se usa en windows, para configurar la librería de las cámaras.
- **GvAppBase, GvCamManager, QtOgre, SistemaMotores**: son librerías intermedias que se compilan como librerías estáticas, y se enlazan al crear el ejecutable.
- **Gv_Pruebas, Gv_Graffiti, Gv_StereoMon**: Las tres aplicaciones desarrolladas.
- **shared**: contienen archivos que son accedidos desde varios módulos.
- **media**: carpeta con todos los recursos que necesitan las aplicaciones.

Cada aplicación está asociada a una carpeta de medios, presentes en la carpeta "media", además de ciertos recursos comunes (media_comun). El archivo de configuración define los recursos necesarios a través del archivo "**resources.cfg**", que contiene rutas a diversas carpetas. Además cada aplicación puede requerir un archivo de escena, donde se incluyen los nodos que utiliza la aplicación, en un archivo XML.

C.2. Compilación

En esta sección se detalla el proceso de compilación del proyecto. En el directorio de la solución se encuentra el archivo de proyecto **GVision.pro**. Para compilar el proyecto se requiere QtCreator, el IDE propio de Qt. El entorno de desarrollo es Ubuntu 16.10. Ademas hay que instalar distintas librerías y hacerlas accesibles al compilador a través de *pkg-config*:

- ARToolKit 2.3.1, y la librería ARVideo de ARToolKit5
- Qt: el alumno utilizó la version 5.4

- Ogre3D: se utilizó la version 1.10.6
- openCv

La aplicación resultante tiene que tener acceso a los archivos de configuración y a la carpeta "**media**". Se pueden copiar de sus respectivas carpetas o crear enlaces dinámicos a la carpeta de los ejecutables.

C.3. Archivos de configuración

En esta sección se listan los distintos archivos de configuración de las aplicaciones. Cada aplicación tiene un archivo de configuración propio, el archivo de los recursos necesarios (modelos 3D, texturas, etc...), y opcionalmente el archivo de la escena inicial.

Este archivo que contiene la configuración de las cámaras y de la aplicación debe encontrarse en el mismo directorio que el ejecutable, y su nombre debe ser **<nombre de la aplicación>_settings.ini**. Si el ejecutable lo encuentra al inicio leerá los valores contenidos para configurar distintas partes de la aplicación. Contiene diversas secciones incluyendo una específica para cada cámara.

- [**Application**]: Contiene la ruta del archivo de recursos y el archivo de escena.
- [**Common**]: Variables de inicialización de ARToolKit, además de a configuración de las rejillas y del algoritmo para aumentar la saturación (en prueba).
  - **AutoThreshold=[true, false]** Indica si el algoritmo de autoumbralización empieza activado por defecto.
  - **DefaultThreshold=⟨int 0-255⟩** Indica el valor por defecto del umbral para la umbralización manual.
  - **DefaultPatternWidth=⟨float⟩** Indica la anchura por defecto de las marcas. Si alguna marca es de otro tamaño habrá que especificarlo explícitamente en el código, aunque esto tan solo es relevante para la marca de calibración (para determinar la distancia de convergencia)
  - **MarkConfidenceThreshold=⟨float 0-1⟩** La tolerancia en la detección del patrón de las marcas. Este valor solo tendrá efecto si se usan marcas TEMPLATE.
  - **PatternsResolution=int** Determina la resolución de las marcas tipo patrón.
  - **ExtremeRGB_activated=[true, false]** Activa el algoritmo para acentuar los colores.

- **ExtremeRGB_threshold=⟨int 0-255⟩** Valor umbral para el algoritmo.
- **ExtremeRGB_factor=⟨float⟩** Factor que se aplicara al valor.
- **RejillaAlto, RejillaAncho, RejillaMargen** definen las dimensiones de la rejilla que se sobreimpresiona a la imagen de las cámaras.

- **[Cam_<función de la camara>]**: Un aparatado por cada función disponible: LEFT, RIGHT, AUX1, AUX2. Si uno de los valores no se encuentra en el apartado específico de la cámara, lo buscara en la sección [Cam_defaults].
  - **Width=⟨int⟩** Anchura de la imagen capturada. Debe ser uno de los modos disponibles para dicho dispositivo.
  - **Height=⟨int⟩** Altura de la imagen capturada. Debe ser uno de los modos disponibles para dicho dispositivo.
  - **CalibrationFile=⟨string⟩** Archivo de calibración del dispositivo. De este archivo se extraerán los parámetros intrínsecos para dicha cámara.
  - **Enabled=[true, false]** Indica si la cámara se usará en la aplicación.
  - **Device=⟨int⟩** Número de dispositivo. En linux corresponde con el dispositivo correspondiente que se encuentra en */dev/videoN*.
  - **IdealFPS=⟨int⟩** Tasa de refresco deseada.
  - **TrackingEnabled=[true, false]** Indica si se realizará seguimiento de marcas con la cámara.
  - **TrackAtStart=[true, false]** Indica si el seguimiento se activará de inicio.
  - **TrackHalfRes=[true, false]** Indica si se utiliza la resolución completa para realizar el seguimiento. Puede ser útil si la resolución de la cámara es muy alta y es suficiente la mitad de resolución para el seguimiento, con el consiguiente ahorro de CPU.
  - **TrackingType=[TEMPLATE, BCH]** Tipo de marcas que se utilizaran en la aplicación.
  - **RPPPoseEstimator=[true, false]** Indica si se quiere utilizar el algoritmo Robust Planar Pose"para el cálculo de la posición de la marca. Este algoritmo obtiene resultados más estables pero puede dar problemas en algunos dispositivos móviles.
  - **BlitOrLock=⟨int 0-1⟩** Indica el método para el volcado de la imágen en la textura, de los dos que se implementaron para comparar rendimiento. Se recomienda mantenerlo en valor 1.

- **[System]**

- **ShowConfigDialog=[true, false]** Muestra la ventana de configuración al iniciar la aplicación.
- **DefaultWindowOpacity=⟨float 0-1⟩** Opacidad de las ventanas de Qt.
- **DefaultSceneFile=⟨string⟩** Archivo que contiene la escena por defecto.
- **DefaultUpdateInterval=⟨int⟩** Frecuencia de refresco de la ventana gráfica en milisegundos.
- **CameraWidth=⟨int⟩** Anchura de la cámara principal.
- **CameraHeight=⟨int⟩** Altura de la cámara principal.

- [**SistemaMotores**]
  - **Puerto=⟨string⟩** Puerto por defecto de conexión con Arduino.
  - **Velocidad=⟨int⟩** Velocidad de la conexión serie, en baudios.
  - **Modelo=⟨int⟩** Indica el tipo de cámara (Paralela o con espejo)
- [**Rejilla**] Parámetros de la rejilla auxiliar para el estereoscopista.
  - **Ancho=⟨int⟩** Número de líneas verticales.
  - **Alto=⟨int⟩** Número de líneas horizontales.
  - **Margen=⟨float⟩** Factor que indica el margen a la rejilla.
- [**Graphics**]
  - **RenderSystem=⟨string⟩** Sistema de renderizado utilizado en la última ejecución (DirectX u OpenGL). Se modifica automáticamente al cambiar la configuración.
  - **WindowModes=[640x480, 800x600, 960x720, 1024x768, 800x450, 1280x720, 1280x1024, 1680x1050, FullScreen]** Modos de ventana que aparecerán en la configuración. Se pueden añadir nuevas resoluciones si es necesario.
  - **SelectedWindowMode=⟨int⟩** Resolución por defecto de la aplicación. Indica el índice sobre la lista anterior.
  - **AllowPerfHUD=[true, false]** Indica si la aplicación permite ser analizada por el programa de NVidia PerfHUD.
  - **EnableGammaCorrection=[true, false]** Habilita la corrección gamma.
  - **FSSAFactor=⟨int⟩** Nivel de filtrado anisotrópico.
  - **EnableVerticalSync=[true, false]** Habilita la sincronización vertical.

- **MirrorMode=[true, false]** Inicia la aplicación con la imagen de la cámara volteada horizontalmente.
- **KeepAspectRatio=[true, false]** Indica si mantiene la relación de aspecto de la ventana gráfica al redimensionarla.
- **AspectRatio=⟨float⟩** Relación de aspecto de la ventana. Dependiente de la resolución escogida.

- [**UI**]
  - **StyleFile=⟨string⟩** Archivo usado para cargar estilos de interfaz.
- [**Stereoscopy**]
  - **EyesSpacing=⟨float⟩** Separación de las cámaras virtuales por defecto.
  - **FocalLength=⟨float⟩** Convergencia de las cámaras virtuales por defecto. Indicada en metros hasta el plano de convergencia.
  - **InverseStereo=[true, false]** Indica si se inicia la aplicación con las cámaras estereoscópicas invertidas.
  - **ScreenSize=⟨float⟩** Tamaño de la pantalla proyectada, para el cálculo de la disparidad.
  - **ViewerDistance=⟨float⟩** Distancia del observador a la pantalla.
- [**Graffiti**] Apartado de configuración exclusivo para GV_Graffiti.
  - **GraffitiWidth=⟨float⟩** Anchura del trazo del graffiti.
  - **GraffitiAlpha=⟨float⟩** Transparencia del trazo.
  - **GraffitiMaterial=⟨string⟩** Material utilizado para el trazo. Debe estar incluido en los archivos de materiales.
  - **FlareWidth=⟨float⟩** Anchura del indicador de pintado.
  - **FlareAlpha=⟨float⟩** Transparencia del indicador de pintado.
  - **TargetFlareWidth=⟨float⟩** Anchura del indicador de posición.
  - **InactivityTimeToScreenSaver=⟨int⟩** En GV_Graffiti, tiempo hasta que se activa el salvapantallas en segundos.

Glosario

Arduino Arduino es una plataforma de hardware y software de código abierto para el desarrollo de prototipos electrónicos. Basada en una sencilla placa con entradas y salidas, analógicas y digitales, en un entorno de desarrollo que está basado en el lenguaje de programación Processing. 8, 74

Direct3D es una API desarrollada por Microsoft utilizada para el procesamiento y la programación de gráficos en tres dimensiones. 24

fiducial Una marca fiducial o fiduciario es un objeto utilizado para la observación de sistemas de imágenes, el cual aparece en la imagen para ser usado como punto de referencia o de medida. 11

keystone El efecto keystone es la distorsión de una imagen causada por la proyección sobre un plano no perpendicular al eje de proyección. En estereoscopía se aplica también al defecto provocado por un par de cámaras dispuestas en ángulo que provoca un paralaje vertical en las cuatro esquinas de la mezcla estereoscópica. 12, 63

OpenGL es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. 24

paralaje es la desviación angular de la posición aparente de un objeto, dependiendo del punto de vista elegido. 10, 90

parámetros extrínsecos hacen referencia a la relación entre los sistemas referencia de la cámara con respecto a la escena. Desde el punto de vista de un par de cámaras estereoscópicas, hace referencia a la posición, tanto en orientación como en traslación, de una cámara con respecto a otra. 51

parámetros intrínsecos son aquellos que definen la naturaleza física de la cámara, incluyendo la geometría interna y la óptica. Son constantes para las cámaras de focal fija, ya que dependen de la posición relativa entre la óptica y el sensor. 30, 59

patrón singleton (instancia única) es un patrón de diseño que consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella. 22

Bibliografía

[1] Bernard Mendiburu. *3D Movie Making: Stereoscopic Digital Cinema from Script to Screen*. 2009.

Recursos de internet

[2] *3+D ENTERTAINMENT*. URL: http://www.3masd.es.

[3] *3dtv.at - Comparación de métodos anaglifo*. URL: http://3dtv.at/Knowhow/AnaglyphComparison_en.aspx.

[4] *A multi camera development framework for custom sensor arrays*. URL: https://codelaboratories.com/products/eye/.

[5] *Blender: open source 3D content creation suite*. URL: http://www.blender.org.

[6] *Boost - C++ Libraries*. URL: http://www.boost.org.

[7] *camera-calib (Camera intrinsic calibration)*. URL: http://www.mrpt.org/list-of-mrpt-apps/application_camera-calib/.

[8] *Canon SDK forum*. URL: http://www.canonsdk.com.

[9] *Dissecting the Camera Matrix*. URL: http://ksimek.github.io/2012/08/22/extrinsic/.

[10] *Estereoscopía - wikipedia*. URL: http://es.wikipedia.org/wiki/Estereoscopia.

[11] *GIMP - The GNU Image Manipulation Program*. URL: http://http://www.gimp.org.

[12] *GTK+ UVC Viewer*. URL: http://guvcview.sourceforge.net/.

[13] *IberOgre, La wiki de Ogre3D en español*. URL: http://osl2.uca.es/iberogre.

[14] *Inkscape: An Open Source vector graphics editor*. URL: http://inkscape.org.

[15] *Irrlicht Engine: A free open source 3d engine*. URL: http://irrlicht.sourceforge.net.

[16] *Kile - an Integrated LaTeX Environment*. URL: http://kile.sourceforge.net.

[17] *LaTeX - A document preparation system*. URL: http://www.latex-project.org.

[18] *MathWorks*. URL: https://www.mathworks.com.

[19] *OGRE3D open-source rendering engine*. URL: http://www.ogre3d.org.

[20] *Panda3D - Free 3D Game Engine*. URL: http://www.panda3d.org/.

[21] *Qt: a cross-platform application and UI framework*. URL: http://qt.nokia.com.

[22] *Singleton - wikipedia*. URL: http://es.wikipedia.org/wiki/Singleton.

[23] *Stereo Vision with OpenCV and QT*. URL: https://github.com/GeospatialDaryl/opencvstereovision.

[24] *Tripod Mount for PS3 Eye*. URL: https://www.thingiverse.com/thing:4154.

[25] *Understanding Comfortable Stereography*. URL: http://bit.ly/2whXrAG.

[26] *UNITY: Game Development Tool*. URL: http://unity3d.com/.

[27] *videoInput: free windows video capture library*. URL: http://muonics.net/school/spring05/videoInput/.

[28] *Wikipedia*. URL: http://es.wikipedia.org.

Índice de figuras

Índice de cuadros