ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

ARES
(Augmented Reality for Embedded Systems)

Para acceder al Título de

Graduado en Ingeniería de Tecnologías de Telecomunicación

Autor: José Alberto Gutiérrez

Julio - 2016

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACION

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE GRADO

Realizado por: José Alberto Gutiérrez Gutiérrez
Director del TFG: Pablo Pedro Sánchez Espeso

Título: ARES - Augmented Reality for Embedded Systems
Title: ARES – Realidad aumentada para sistemas embebeidos

Presentado a examen el día: 27/07/2016

para acceder al Título de

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Composición del Tribunal:
Presidente (Apellidos, Nombre):
Secretario (Apellidos, Nombre):
Vocal (Apellidos, Nombre):

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: El Vocal

Fdo.: El Director del TFG
(sólo si es distinto del Secretario)

TÍTULO				
AUTOR				
DIRECTOR				
TITULACIÓN	**GRADUADO EN INGENIERIA DE TECNOLOGIAS DE TELECOMUNICACIÓN**	**FECHA**	MES-AÑO	**TOMO I DE I**

ÍNDICE

1 INTRODUCCIÓN Y MOTIVACIÓN

Los continuos avances en capacidad de integración han permitido que los dispositivos electrónicos cada vez cuenten con más prestaciones, lo cual permite el desarrollo de aplicaciones cada vez más complejas y potentes.

Además, el desarrollo de tecnologías de bajo consumo permite que estos dispositivos puedan funcionar, alimentados por baterías, durante un tiempo razonable lo que posibilita la creación de aplicaciones de altas prestaciones portátiles y/o autónomas. La evolución de la tecnología es tan importante que hoy en día es posible contar con dispositivos móviles de pequeño tamaño y alta autonomía con una capacidad de cómputo similar a ordenadores de sobremesa de hace menos de 5 años.

A la potencia de cómputo se añade su portabilidad y conectividad con una red casi siempre disponible, lo que permite el desarrollo de innumerables nuevas aplicaciones que hasta la fecha no había sido podido crear. A continuación se introducen algunas de estas nuevas aplicaciones, que constituyen el objetivo del trabajo.

1.1 REALIDAD VIRTUAL

El objetivo de la realidad virtual (VR) es crear un mundo sintético y lograr que el usuario tenga la sensación de estar presente en él. Las técnicas de realidad virtual tienen que ser capaces de proporcionar a los sentidos del usuario la información que precisen para que la persona sienta que forma parte de la realidad creada por el ordenador. Dicha realidad es totalmente sintetizada por el sistema, sin incluir elementos del mundo real que rodea al usuario.

La realidad virtual es una tecnología muy apreciada por la industria del videojuego, que desde siempre busca la máxima inmersión del jugador. Dicha industria ha sido la que más ha potenciado, definido y condicionado el desarrollo de la VR.

Las técnicas de VR generan información orientada a los sentidos del usuario. Como la vista es el principal sentido que permite la inmersión en la realidad, la mayoría de tecnologías de VR se centran en generar imágenes que puedan ser vistas por el usuario. Por esta razón, la realidad virtual se suele asociar con un visor adaptable a la cabeza o casco, el cual cuenta con dos lentes por las cuales se muestra la imagen de un monitor a cada ojo. Esta técnica, conocida como visión estereoscópica, permite al usuario “ver el mundo virtual” en tres dimensiones, como si fuera real.

Además de mostrar imágenes estereoscópicas, otro requisito del casco de VR es ser capaz de determinarla orientación y movimientos del usuario, para poder gestionar su actividad/movimiento en el mundo virtual.

1.1.1 Breve historia de la realidad virtual

Uno de los elementos esenciales de la realidad virtuales la visión estereoscópica. Esta técnica permite que cada ojo reciba una imagen diferente, lo que posibilita que el cerebro genere percepción de profundidad (imagen en 3 dimensiones ó 3D).

El concepto de imagen 3D surge en el año 1844, cuando Charles Wheatstone inventó el "estereoscopio" que sirvió de base a los primeros visores de realidad virtual en el siglo XX. El "estereoscopio" utiliza dos fotografías de la misma escena, pero tomadas desde posiciones distintas (posición de los "ojos del observador"). Estas imágenes serán observadas por cada ojo de manera separada, lo que permite que el cerebro las procese y genere una única visión de la escena con sensación de profundidad (efecto tridimensional).

En 1950, Morton Heilig amplió el concepto de realidad virtual con el "Cine de Experiencia", que permitía sincronizar todos los sentidos de una manera efectiva, integrando al espectador con la actividad en la pantalla de cine. Construyo un prototipo llamado el Sensorama en 1962 y produjo 5 filmes cortos que permitían aumentar la experiencia del espectador a través de sus sentidos (vista, olfato, tacto, y oído).

En 1955, Nintendo presentó el primer visor estereoscópico para juegos: la consola Virtual Boy.

Ilustración 1 Virtual Boy

Este primer casco mostraba una imagen estereoscópica monocromática, consiguiendo así un efecto de visión en 3D. Curiosamente esta consola fue un auténtico fracaso comercial y nunca llego a comercializarse en Europa debido a su alto precio y problemas de uso (cansaba la vista y podía producir mareos).

En 1968, Ivan Sutherland, con la ayuda de su estudiante Bob Sproull, construyeron lo que es ampliamente considerado como el primer visor montado en la cabeza (o Head Mounted Display, HMD) para Realidad Virtual.

El HMD usado por el usuario era tan grande y pesado que debía colgarse del techo. Además el sistema era muy primitivo en términos de interfaz de usuario y realismo, siendo las imágenes virtuales generadas simples "modelos de alambres". A finales de los 80 se popularizo el término "Realidad Virtual" por Jaron Lanier, uno de los fundadores de la empresa VPL Research que produjo algunos de los primeros guantes y visores de Realidad Virtual.

Las continuas mejoras de las técnicas de VR han permitido su popularización y aceptación comercial. Por esta razón están apareciendo continuamente nuevos productos en el mercado como Samsung Gear VR, Playstation VR (Sony), HTC Vive (Valve y HTC) y Oculus Rift (recientemente adquirida por Facebook).

La mayoría de las aplicaciones actuales de VR se orientan al mercado de videojuegos, aunque también se utilizan en educación, control de sistemas y entrenamiento.

Un ejemplo de aplicación de las técnicas de VR es su uso en la formación de soldados. VR permite crear entornos bélicos en donde los soldados "se mueven" como si fuese el mundo real, con enemigos virtuales.

Otro ejemplo de aplicación lo podemos encontrar en el campo de la medicina, en donde los cirujanos pueden formarse con sistemas de VR.

Ilustración 2 Soldado usando un sistema de realidad virtual

1.2 REALIDAD AUMENTADA

Realidad aumentada (AR) es el término que se utiliza para definir técnicas que, de forma directa o indirecta, combinan elementos del entorno físico (mundo real) con elementos virtuales para la creación de una realidad mixta en tiempo real. La realidad aumentada incluye al conjunto de dispositivos que añaden información virtual a la información física ya existente, es decir, añaden una parte sintética virtual a lo real. Esta es la principal diferencia con la realidad virtual, puesto que no sustituye la realidad física, sino que incluye elementos virtuales en el mundo real.

Se puede hablar de realidad aumentada cuando se cumplen los siguientes requisitos:

1. Se combinan elementos reales y virtuales.
2. Es interactiva y responde en tiempo real.
3. Toda la información se gestiona en 3D.

1.2.1 Breve historia sobre la realidad aumentada

Históricamente, la realidad virtual (VR) y la realidad aumentada (AR) se han desarrollado de forma conjunta.

El término "Realidad Aumentada" fue introducido en 1992 por el investigador Tom Caudell en Boeing. Caudell fue contratado para encontrar una alternativa a los tediosos tableros de configuración de cables se utilizan en la fabricación de aviones. En investigador concibió unos anteojos especiales que combinaban tableros virtuales sobre tableros reales, por lo que estaba "aumentando" la realidad que veían los electricistas. El término "Realidad Aumentada" fue introducido en un artículo de Caudell publicado en 1992.

Aunque desde 1994 se empezaron a desarrollar diversos sistemas de AR (como KARMA, ARToolKit o FLARTookKit) no fue hasta 2012 cuando dichas técnicas se potencian y popularizan con el anuncio de Google Glass, unas gafas que permitían incluir información adicional en lo que se está viendo.

A diferencia de la realidad virtual, la realidad aumentada tiene aplicaciones más extendidas que el entretenimiento y los videojuegos. Casi cualquier tarea de carácter manual puede verse mejorada si se añade una capa virtual. De la misma manera, pueden crearse simuladores que ayuden a la formación y que mezclen elementos reales y virtuales, como los simuladores de cabinas de pilotajes de aviones.

Ilustración 3 - Aplicación de realidad aumentada en una móvil.

Durante las últimas semanas estamos asistiendo al éxito comercial de una aplicación móvil de AR para entretenimiento: Pokemon GO de Nintendo. Esta app mezcla imágenes captadas con la cámara del móvil con monstruos virtuales (Pokemons) utilizando además información de la posición del móvil (geolocalización obtenida mediante GPS), datos cartográficos, puntos de interés y geoposición de los establecimientos comerciales que pagan a Niantic, Inc (creadores de esta aplicación) por atraer a los usuarios del juego a sus tiendas (Marketing 3.0).

1.3 Objetivos del Proyecto

El objetivo de este trabajo es el desarrollo de un entorno de realidad aumentada que utilice un visor de realidad virtual, los sensores de posición que integra el visor y un par de cámaras para capturar la realidad. Además, el sistema tiene que ser implementado en una plataforma embebida, que permita al usuario desplazarse sin necesidad de estar conectado con cables al sistema de cómputo.

El sistema debe utilizar software libre, no solo para eliminar el coste de la licencia, sino también para permitir acceder al código fuente y poder adaptar el sistema a casos particulares o nuevas plataformas.

Como se trata de una aplicación de realidad aumentada, será necesario el uso de dos cámaras para conseguir una visión estéreo, y además el uso de un motor gráfico para la generación de gráficos en 3D.

En resumen, el objetivo principal del proyecto es desarrollar un entorno de realidad aumentada que pueda servir como punto de partida del desarrollo de nuevas aplicaciones que utilicen esta tecnología. El entorno estará formado por una serie de dispositivos hardware y el software que permite crear y gestionar la realidad aumentada.

2 REQUERIMIENTOS DEL SISTEMA

El objetivo del proyecto es desarrollar un entorno de realidad aumentada que sirva como modelo en el desarrollo de futuras aplicaciones. Dicho entorno debe verificar los siguientes requisitos:

1.- Incluir unas gafas de realidad virtual con sensores que determinen su posición, orientación y movimiento en tiempo real.

- Se ha seleccionado el HMD o visor de realidad virtual Oculus Rift DK1. Dicho visor es controlado mediante la librería libOVR.

2.- Incluir cámaras para capturar imágenes del entorno real.

- Se han seleccionado 2 cámaras de Logitech que se comunican con el sistema mediante una interfaz USB. Las cámaras serán gestionadas mediante la librería V4L (Video for Linux).

3.- Los cómputos se realizan en una plataforma embebida y portátil, que se pueda desplazar con el usuario.

- Se ha seleccionado una placa de desarrollo con arquitectura ARM: la Odroid XU3. Dicha placa es gestionada con el sistema operativo Ubuntu 15.

4.- Se utilizará un sistema de generación de realidad virtual (motor) de código abierto y multiplataforma. Dicho motor debe soportar visión estéreo y ser capaz de crear gráficos en 3D con modelos, texturas y luces.

- Se ha seleccionado Ogre3D

5.- El entorno se desarrollará en C++, y las librerías y módulos adicionales utilizados debe ser de código abierto y multiplataforma.

- Se han utilizado varias librerías, aplicaciones y sistemas operativos de código abierto, como Ubuntu 15, Ogre3D o Eclipe CDT.

6.- Las prestaciones del sistema deben ser suficientes como para proporcionar al usuario la sensación de inmersión en la realidad "aumentada".

- El sistema al menos debe generar 30 fps (frames per second).

7.- Se desarrollará una aplicación ejemplo, que incluya en la realidad física un cuadrado y un modelo (monstruo) en 3D.

8.- La aplicación se podrá contralar desde teclado o desde un programa ejecutado por otro ordenador. La aplicación también podrá requerir información almacenada en servidores remotos.

- Se ha integrado un módulo que permite transferir mediante socket datos y control a la aplicación a desarrollar

Estas especificaciones definen un sistema de realidad aumentada que integra las capas mostradas en la figura adjunta. La aplicación de usuario utiliza un entorno que incluye un motor de realidad virtual (Ogre3D), un visor o casco de realidad virtual (HMD) gestionado por una librería específica (libOVR) y unas funciones que permiten acceder a las imágenes de las cámaras (funciones de V4L). La aplicación se ejecuta en una placa Odroid XU3 con sistema operativo Linux (Ubuntu 15). Además de la placa con el procesador, el hardware incluye el hardware del visor Oculus Rift (monitor y sensores de posición) así como la cámara estéreo.

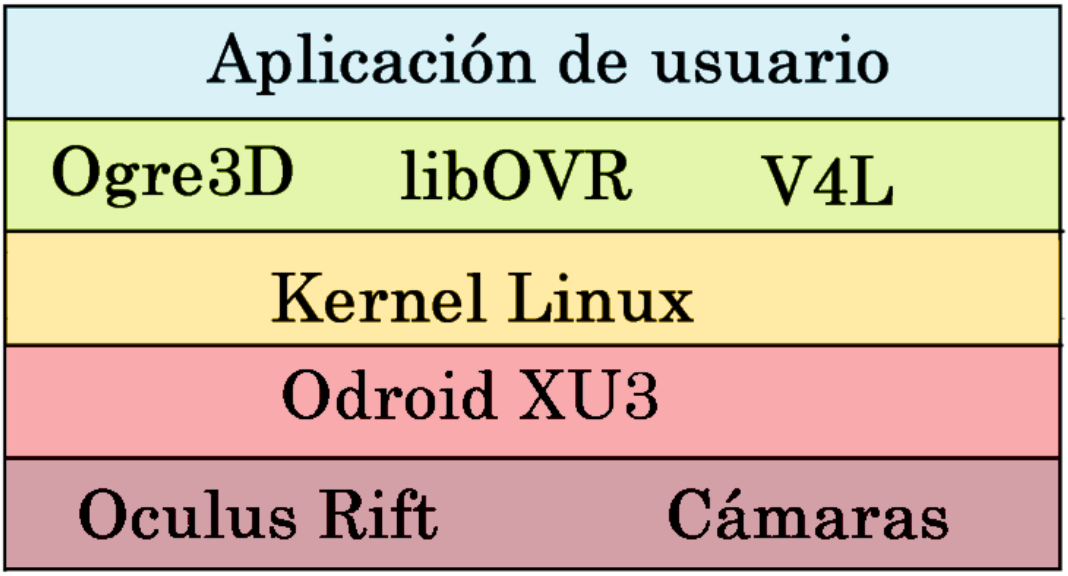


Ilustración 4- Capas del sistema

3 ESTADO DEL ARTE

3.1 OGRE 3D

3.1.1 Introducción

Ogre3D [1], acrónimo del inglés "Object-OrientedGraphicsRendering Engine" es un entorno (motor) de generación de imágenes 3D (renderizado) orientado a escenas, escrito en el lenguaje de programación C++ y de código abierto. Fue diseñado para que a los desarrolladores les resulte más fácil e intuitiva la producción de aplicaciones que utilizan gráficos 3D acelerados por hardware. Las clases incluidas en sus librerías evitan la dificultad de utilización de las capas inferiores de librerías gráficas como OpenGL y Direct3D, además de proveer una interfaz basada en objetos del mundo virtual y clases de alto nivel. A diferencia de un motor de videojuegos (como Unreal Engine o Unity), no está específicamente diseñado para la creación de juegos lo que amplía sus posibles aplicaciones.

OGRE tiene una comunidad muy activa, y fue el proyecto del mes de marzo de 2005 de SourceForge. Además, ha sido usado en varios videojuegos comerciales como Ankh, Torchlight, Garshasp y Earth Eternal. Actualmente OGRE es un entorno multi-plataforma y es soportado por varios sistemas operativos, como Linux, Windows (las últimas versiones), OS X, WinRT, Windows Phone 8, iOS y Android. La comunidad FreeBSD mantiene una versión no oficial.

La versión 1.0.0 ("Azathoth") fue publicada en febrero de 2005. La versión actual es la 1.9.0 ("Ghadamon"), lanzada en Noviembre de 2013. La información contenida en esta sección está basada del manual de usuario de OGRE3D, versión 1.9 [2].

Ilustración 5–Ejemplo de aplicación de OGRE que muestra como incluir una capa de agua que se mueve sobre las baldosas de una escena.

3.1.2 Objetos y jerarquía

Ogre3D es un motor orientado a escena, es decir, parte de una escena y sobre ella crea todos los demás objetos que contendrá el mundo virtual. Ogre3D es únicamente un motor gráfico, por lo que no cuenta con elementos incluidos en motores de videojuegos, como librerías de física (que simulan el movimiento del objeto conforme a las leyes de la física) ni interfaces gráficas de usuario. Si deseamos crear un videojuego, tendremos que añadir librerías específicas que proporcionen comportamientos físicos e interacción con el usuario.

A continuación se muestra un diagrama con algunos de los objetos más importantes de Ogre [2].

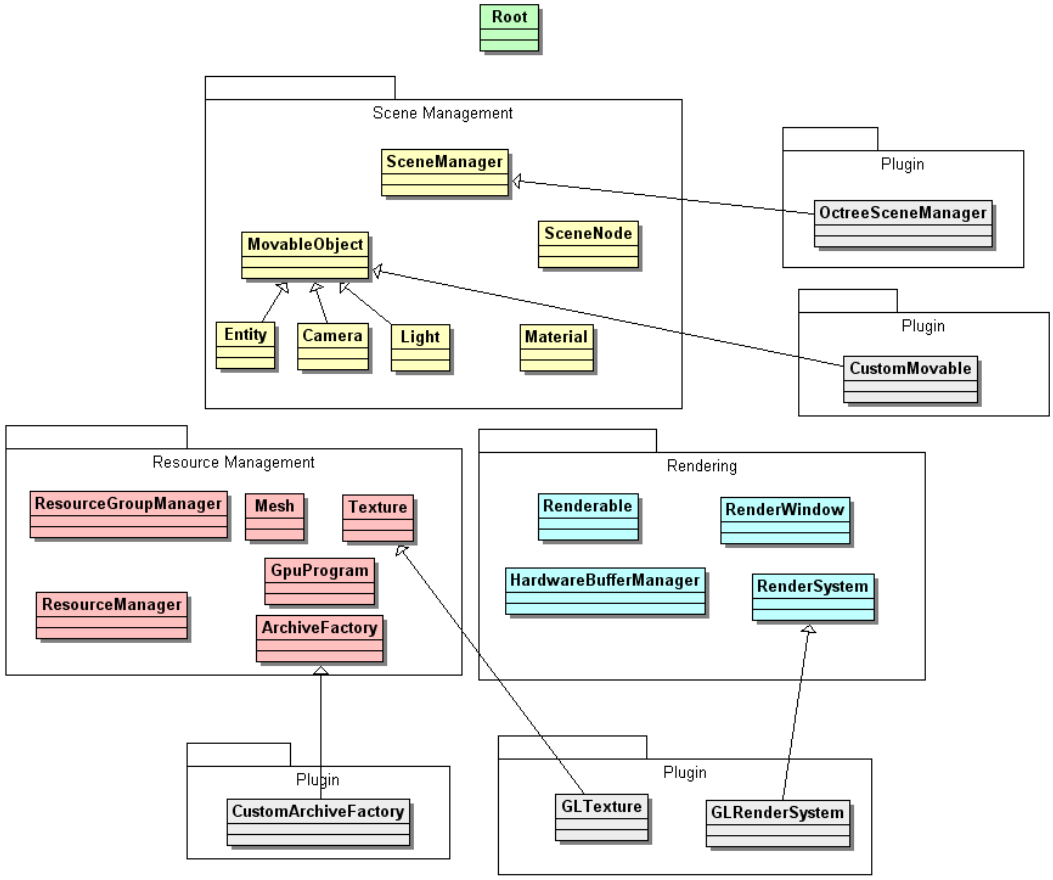


Ilustración 6 - Esquema de clases de OGRE

Por encima de todo se encuentra la clase Root, que es la interfaz principal que permite acceder a las funcionalidades de Ogre3D. Desde Root es posible configurar prácticamente cualquier elemento, desde la creación de escenas, gestión de ventanas y sistemas de renderizado hasta la carga de plugins con funcionalidades adicionales o recursos. En definitiva, Root es un objeto que facilita el desarrollo y ordenar la aplicación de usuario.

El resto de clases se dividen principalmente en tres grupos:

- Scene Management: Incluye todos los elementos que tratan sobre cómo se posicionan los objetos en la escena y como están estructurados. Se incluyen en este apartado las cámaras que muestran vistas de la escena, luces y objetos en general. Normalmente en Ogre se definen los objetos desde alto nivel, indicando donde se colocan y desde que cámaras van a ser visibles.
- Resource Management: Incluye todos los recursos que van a ser necesarios para generar la imagen (renderización) de la escena. En este grupo se integran todas las texturas, fuentes, modelos de objetos, etc. Es importante que los recursos se carguen, usen y liberen con cuidado, para evitar sobrecargar al motor. Además, hay que conocer que métodos y clases usar para cada tipo de recurso y caso de uso.
- Rendering: Incorpora todas las funcionalidades relacionadas con el control de la pantalla. En general se incluyen todos los elementos de bajo nivel, como buffers de almacenamiento de imágenes, estados de render (generador de escenas) y la gestión del pipeline de imágenes/pixeles. Mediante las clases del grupo "Scene Management" es posibles efectuar estas operaciones a más alto nivel y de una forma más sencilla.

Ogre3D cuenta además con una gran variedad de complementos (plugins) que añaden o modifican funcionalidades del sistema. Muchos de ellos son partes fundamentales del sistema, mientras que otros son solo elementos suplementarios. Esto hace que el sistema sea muy modular siendo posible cambiar una parte del sistema sin tener que modificar el código de usuario.

Entre los elementos incluidos como complemento o plugin se encuentra el sistema de renderizado lo que permite a OGRE utilizar distintas librerías y generadores gráficos sin tener que modificar la aplicación. Esto proporciona una gran portabilidad y permite que las aplicaciones funcionen en diferentes plataformas y sistemas operativos con cambios mínimos.

3.1.3 Objeto Root

Se puede definir el objeto root como el punto de entrada a Ogre. Root debe ser el primer objeto en ser creado y el último en ser destruido. Root incluye todos los parámetros de configuración del sistema, siendo posible modificarlos mediante el método "showConfigDialog ()". Dicho método permite al usuario ajustar todo el sistema, seleccionando el render que desee, la resolución, profundidad de color y otros parámetros de la imagen. El objeto Root puede ser modificado desde el código de aplicación o desde un archivo de configuración.

Además, desde el objeto Root es posible obtener punteros a otros objetos esenciales como el SceneManager, el RenderSystem o los ResourceManager de cada uno de los tipos de recurso.

Por último, para que la aplicación comience a renderizar todos los objetos de la escena se deberá llamar al método "startRedering ()", que incluye un bucle infinito que solo finalizará cuando todas las ventanas se cierren o algún evento de tipo FrameListening se lo indique. También es posible que el usuario desee

gestionar dicho bucle, en cuyo caso tendrá que utilizar el método “RenderOneFrame()” en cada iteración para generar la imagen.

3.1.4 Objeto RenderSystem

El objeto RenderSystem es en realidad una clase abstracta que define la interfaz subyacente a la API (Aplication Programming Interface) 3D. Podemos definirla como un puente entre OGRE y la librería gráfica, que a su vez se comunicará con el sistema operativo y el hardware.

Este objeto es el responsable de enviar operaciones de rendering (generación de imagen) a la API gráfica y de configurar todas las opciones de rendering. Después del que el sistema haya sido inicializado con “Root::initialise”, se podrá determinar la API grafica seleccionada por el objeto “RenderSystem” con el método “Root:getRenderSystem()”. Sin embargo, en una aplicación típica no debería ser necesario manipular directamente el objeto RenderSystem. La funcionalidad que será necesaria para renderizar objetos y cambiar configuraciones debería estar disponible en los objetos de tipo SceneManager, Material u otras clases orientadas a la escena.

Solo si queremos crear varias ventanas de rendering (con escenas completamente separadas, no múltiples vistas de la misma escena, como una pantalla partida con dos imágenes) o acceder a otras características más avanzadas, necesitaremos acceder al objeto RenderSystem.

3.1.5 Objeto SceneManager

Aparte del objeto Root, esta es probablemente la parte más crítica del sistema desde un punto de vista de aplicación. Probablemente, SceneManager será el objeto que más utilizará la aplicación.

El SceneManager está a cargo de contenidos acerca de la escena que será renderizada por el motor. Es el responsable de organizar el contenido usando las técnicas que considere mejores, crear y controlar todas las cámaras, objetos movibles (Entities), luces y materiales (Propiedades de superficie de los objetos). Es por tanto es el que dirigirá la geométrica en el mundo.

No es necesario que la aplicación lleve un control sobre los objetos que se van creando, ya que cuando se crea un objeto, se le deberá asignar un nombre y más adelante se podrá acceder a ese objeto mediante los métodos getCamera, getLight, etc.

SceneManager además envía la escena al objeto RenderSystem cuando va a renderizar la escena. Nunca se tiene que llamar al método “SceneManager::renderScene” directamente, ya que es llamado automáticamente cuando una ventana o pantalla hace una petición de actualización.

La mayoría de las interacciones con el SceneManager ocurrirán mientras se configura la escena. Probablemente se necesite llamar a un gran número de métodos para preparar la escena antes de que

se inicie la visualización, aunque es posible modificar el contenido de forma dinámica si se prepara un FrameListener en la aplicación de usuario.

Debido a que los distintos tipos de escena requieren algoritmos muy diferentes para decidir que objetos se enviarán al RenderSystem de la forma más óptima, la clase SceneManager está diseñada para ser subclase de diferentes tipos de escenas. El objeto SceneManager por defecto renderizará una escena, pero hará poco o nada por la organización y no se debería esperar resultados con un buen rendimiento en el caso de escenas grandes. Lo lógico sería utilizar una escena específica para cada tipo.

Un ejemplo es la clase BspSceneManager que esta optimizada para grandes niveles de interiores basados en una partición binaria del espacio (Binary Space Partition).

La aplicación no tiene por qué saber que subclases están disponibles, simplemente llamara a "Root::createSceneManager" pasando el argumento del tipo de escena(ST_GENERIC,ST_INTERIOR) y OGRE usará automáticamente la mejor subclase de SceneManager disponible o por defecto la SeceneManager básica si no lo está.

Esto permite que los desarrolladores de OGRE añadan nuevas escenas especializadas más adelante sin tener que modificar el código de usuario.

3.1.6 Objetos ResourceGroupManager

La clase ResourceGroupManager es en realidad un conjunto de gestores de recursos que la aplicación puede utilizar como texturas o mallas (Mesh). En esta clase es posible acceder a otros gestores que podrán ser cargados o liberados cuando se quiera.

ResourceManagerse asegura que los recursos solo sean cargados una vez y a continuación sean compartidos a través del motor OGRE. Además se gestionaran los requerimientos de la memoria que los recursos van a requerir. El gestor permite buscar los recursos en múltiples rutas de forma recursiva e incluso dentro archivos comprimidos de tipo ZIP.

La mayoría de las veces, no necesitaremos interactuar con los gestores de recursos directamente. Los gestores de recursos serán llamados por otras partes del sistema de OGRE cuando sean requeridos, por ejemplo cuando se haga una petición de una textura para ser añadida a un material, el TextureManager será llamado automáticamente. Si se quiere, se podrá llamar al gestor de recursos directamente para precargar recursos pero en la mayoría de los casos es mejor que el entorno OGRE decida qué hacer.

Una tarea necesaria es indicar al gestor donde están los recursos. Para ello se utiliza el método "Root:getSingleton().addResourceLocation", que transfiere la información al ResourceGroupManager.

Debido a que solo existe una instancia por cada gestor de recursos en el motor, si se desea obtener una referencia de un gestor usaremos la siguiente sintaxis:

"TextureManager::getSingleton().someMethod()"

"MeshManager::getSingleton().someMethod()"

3.1.7 Objeto Mesh

Un objeto mesh representa un modelo discreto de un conjunto geométrico auto contenido y escalado según la escena. Se asume que los objetos mesh representan objetos inmovibles y son usados en los escenarios normalmente para crear fondos y adornos.

Los objetos mesh son un tipo de recurso y son gestionados por el gestor de recursos MeshManager. Los archivos que almacenan datos del objeto mesh suelen tener un formato propio de Ogre, el formato '.mesh'. Para crear un archivo mesh podemos utilizar la aplicación de modelado que se distribuye con Ogre.

Además, es posible crear objetos mesh invocando al método "MeshManager::createManual". De esta forma es posible definir la geometría del objeto sin utilizar archivos adicionales.

Los objetos mesh son la parte básica de los objetos individuales que se pueden mover en el mundo virtual. Dichos objetos movibles se denominan Entities que podrán además ser animados mediante el método SkeletalAnimation.

3.1.8 Otros objetos: Entities, Materials y Overlays

Un Entity es una instancia de un objeto movible en la escena. Puede ser un coche, una persona, un perro un shuriken, etc. Esto significa que no tiene por qué tener necesariamente una posición fija en el mundo.

Los entities están basados en meshs discretos, es decir, en un conjunto de elementos geométricos auto-contenido. Múltiples entities pueden estar basados en un mismo mesh, y ser simplemente copias, cada cual con una posición diferente dentro de la escena.

Se creará una entity llamando al método "SceneManager::createEntity"; se le dará un nombre y se especificará el objeto mesh en el cual se basa. El SceneManager se asegurara que ese mesh es cargado llamando al gestor de recursos MeshManager. Solo una copia de ese mesh será cargada.

Cuando un mesh es cargado, el sistema identifica los materiales definidos en el mesh. Es posible que el mesh tenga más de un material asignado ya que partes diferentes del objeto pueden usar distintos materiales.

Cualquier entity creado de un mesh automáticamente usa los materiales por defecto. Sin embargo, es posible definir que ciertas entities estén basados en el mismo mesh pero con diferentes texturas.

El objeto material controla como los objetos en la escena son renderizados. Especifica que superficies de los objetos reflejan la luz o son traslucidos, indica como las texturas se presentan en las superficies y que imágenes utilizan, señala que efectos especiales aparecen y cómo se comportan en el entorno del mundo.

Los materiales pueden ser definidos mediante una llamada a "SceneManager::createMaterial", indicando sus parámetros de configuración o especificado un script externo que los defina en tiempo de

ejecución. Básicamente todo lo relacionado con la apariencia de un objeto está controlado por la clase material.

La clase SceneManager gestiona la lista de los materiales disponibles para la escena. A esta lista se le pueden añadir elementos llamando a "SceneManager::createMaterial" o cargando un mesh, que contendrá las propiedades de los materiales. Cuando un material es añadido a SceneManager, el mismo parte con unas propiedades por defecto, que son definidas por OGRE:

- ambient reflectance = ColourValue::White (full)
- diffuse reflectance = ColourValue::White (full)
- specular reflectance = ColourValue::Black (none)
- emissive = ColourValue::Black (none)
- shininess = 0 (not shiny)
- No texture layers (& hence no textures)
- SourceBlendFactor = SBF_ONE, DestBlendFactor = SBF_ZERO (opaque)
- Depth buffer checking on
- Depth buffer writing on
- Depth buffer comparison function = CMPF_LESS_EQUAL
- Culling mode = CULL_CLOCKWISE
- Ambient lighting in scene = ColourValue(0.5, 0.5, 0.5) (mid-grey)
- Dynamic lighting enabled
- Gourad shading mode
- Solid polygon mode
- Bilinear texture filtering

Es posible cambiar esta configuración llamando a "SceneManager::getDefaultMaterialSettings()" y haciendo los cambios requeridos en la referencia al material que es retornado por la clase. Podemos además cambiar el material usado por una entidad. Tan solo se creara un nuevo material y lo asignaremos a la entidad o a parte de ella (una subentidad) usando el método "SubEntity::setMaterialName()".

El Overlay permite renderizar elementos en 2D y 3D por encima del contenido de la escena, creando efectos de pantallas, sistemas de menú, paneles de estado, etc. Se crean overlays con el método "SceneManager::createOverlay " o también es posible definirlo en un script externo, con extensión ".overlay". En la realidad es más práctico utilizar scripts ya que es más fácil modificarlos sin tener que recompilar el código.

Los Overlays están diseñados para pantallas no interactivas, aunque es posible usarlos como una GUI (Interfaz Gráfica de Usuario) pura. Para una solución GUI completa, es necesario usar librerías adicionales, como CEGui, MyGUI o libRocket.

3.2 OCULUS RIFT

3.2.1 Introducción

Oculus Rift [3] son unas gafas de realidad virtual que está siendo desarrolladas por Oculus VR, que en 2014 fue adquirida por Facebook. Inicialmente fue un proyecto Kickstarter.

Las gafas con el Development Kit 1, también llamado DK1, fue enviado a aquellos que respaldaron el proyecto de Kickstarter con 300 dólares o más. Posteriormente fue distribuido a otros usuarios por Oculus VR. DK1 fue retirado del mercado en marzo de 2014, días antes de la presentación del segundo kit de desarrollo o DK2.

Ilustración 7 - Vista frontal de Oculus

El primer prototipo del dispositivo utilizaba una pantalla de 5,6 pulgadas. Después del éxito en Kickstarter, Oculus decidió cambiar el diseño y utilizar una nueva pantalla de 7 pulgadas, por problemas de suministro. Esto hizo que hizo que el DK1 fuera algo más voluminoso que los primeros prototipos.

El tiempo de refresco de imagen de este nuevo panel es mucho más rápido que los prototipos iníciales, lo que reduce la latencia y el desenfoque de movimiento al girar la cabeza de una forma rápida. La pantalla LCD es más brillante y la profundidad de color es 24 bits por píxel. La pantalla de 7 pulgadas también hace que la imagen 3D estereoscópica ya no tenga superposición de 100%: el ojo izquierdo ve área adicional a la izquierda y el ojo derecho ve área adicional a la derecha lo que imita la visión humana normal.

Ilustración 8 - Vista trasera de Oculus. Se pueden ver las dos lentes

El campo de visión es de más de 90 grados horizontales (110 grados en diagonal), que es más del doble del campo de visión de la mayoría de los dispositivos del mercado, y es uno de los atractivos del dispositivo. Este está destinado a cubrir casi todo el campo visual del usuario, para crear un fuerte sentido de la inmersión. La resolución total es de 1280 × 800 (Con relación de aspecto 16:10), que permite una imagen de 640 × 800 por ojo (Con relación de aspecto 4:5). Sin embargo, el DK1 no cuenta con un 100% de coincidencia entre los ojos por lo que la resolución horizontal combinada es efectivamente superior a 640 pixeles. La imagen para cada ojo se muestra en el panel como una imagen con distorsión de tipo barril que se corrige a continuación por las lentes del visor para generar una imagen esférica. Se espera

que la resolución del Oculus Rift aumente a 2560 × 1440 pixeles en próximas versiones. A esto hay que sumarle la pérdida de resolución visible al no aprovechar toda la pantalla y ampliar la imagen con las lentes.

Los prototipos iníciales utilizaron un sensor de movimiento de cabeza de tipo Hillcrest 3DOF (3 grados de libertad) que normalmente muestrea la posición con una frecuencia de 120 Hz. Una optimización posterior lo hizo funcionar a 250 Hz. La frecuencia de muestro es importante debido a la dependencia entre el realismo de la realidad virtual y el tiempo de respuesta de los sensores. La última versión incluye el nuevo sensor de Reality, que funciona a 1000 Hz y que permite el seguimiento de los movimientos de la cabeza con una latencia mucho más baja. Este sensor utiliza una combinación de giroscopios de 3 ejes, 3 acelerómetros y 3 magnetómetros, que lo hacen capaz determinar la posición del visor en relación al campo magnético de la tierra.

El peso del visor es de aproximadamente 379 g (un aumento de alrededor de 90 gramos es debido al aumento de tamaño de la pantalla) y no incluye auriculares. El visor dispone de un tornillo cada lado, el cual se puede mover con un destornillador, que permite ajustar la distancia entre el visor y los ojos. El kit de desarrollo también incluye lentes intercambiables que permiten corregir las dioptrías del usuario. El ajuste de la distancia entre cada ojo se hace por software.

Ilustración 9 - Caja de control de Oculus

La versión DK1 tiene entradas DVI y HDMI en la caja de control, e incluye un cable DVI, dos cables HDMI y un adaptador de DVI a HDMI. La interfaz USB se utiliza para enviar datos de posición y movimiento del visor a la unidad de cómputo además de poder proporcionar alimentación al dispositivo. Sin embargo, ya que los requerimientos máximos de energía del visor superan ligeramente la clasificación de USB, el equipo se distribuye con un adaptador de corriente que se puede utilizar para alimentar la caja de control en equipos que no proporcionan suficiente potencia a través del cable USB.

Ilustración 10 - Oculus al completo

3.2.2 Componentes

Oculus Rift (o simplemente Rift) cuenta con los siguientes componentes:

- Lentes: Son intercambiables y producen un efecto de amplio campo de visión (FOV), aunque también producen distorsión y aberración cromática
- Display LCD: La resolución es de 1280 × 800 (con relación de aspecto 16:10), que significa una resolución efectiva de 640 × 800 pixeles por ojo
- Sensores: 3 giroscopios, 3 acelerómetros y 3 magnetómetros
- Caja de control: Con conectores DVI, HDMI, USB y alimentación

3.2.3 Librería OVR: libOVR

3.2.3.1 Introducción a OVR

La librería OVR [4] permite integrar la aplicación de usuario y Oculus Rift. Esta librería es la parte fundamental del SDK de Oculus. Una aplicación basada en OVR incluye tres fases principales: arranque, bucle principal y apagado. Para utilizar Oculus en una aplicación se debe hacer lo siguiente:

- Inicializar la librería OVR (libOVR) mediante la función ovr_Initialize
- Llamar a ovr_Create y comprobar que el valor devuelto es correcto. Es posible periódicamente comprobar la presencia del caso HMD con ovr_GetHmdDesc(nullptr)
- Integrar el seguimiento de la cabeza con la visión de la aplicación. Para ello es necesario:
    - Obtener una predicción de la orientación mediante GetPredictedDisplayTime y ovr_GetTrackingState
    - Aplicar la orientación y posición de la visión de la cámara mientras se tienen en cuenta otros controles de la aplicación.
    - Modificar el movimiento y la orientación según la posición de la cabeza
- Inicializar el renderizado de la imagen
    - Seleccionar parámetros tales como resolución o la profundidad de campo basándose en la capacidad del visor HMD
    - Configurar el renderizado creado por las especificaciones de la librería gráfica (OpenGL) en cuanto intercambios de texturas
- Modificar la rederizacion de imágenes (frames) de la aplicación para integrar el visor HMD y los tiempos de frame.
    - Se deberá asegurar que el motor soporta renderizado de visión estéreo
    - Añadir una lógica en el bucle del rederizado para poder predecir una posición de ojo
    - Renderizar cada vista de ojo para un objetivo de render intermediario
    - Enviar el frame renderizado al casco llamando a ovr_SubmitFrame
- Personalizar la UI para que funcione bien dentro del casco
- Destruir todos los recursos creados y finalizar la librería

3.2.3.2 Inicialización y enumeración de sensores

El siguiente ejemplo inicializa libOVR y comprueba que el visor HMD está disponible.

```
// Include the OculusVR SDK
 #include <OVR_CAPI.h>
 void Application()
 {
ovrResult result = ovr_Initialize(nullptr);
 if (OVR_FAILURE(result))
 return;
ovrSession session;
ovrGraphicsLuidluid;
 result = ovr_Create(&session, &luid);
 if (OVR_FAILURE(result))
 {
ovr_Shutdown();
 return;
 }
ovrHmdDescdesc = ovr_GetHmdDesc(session);
ovrSizei resolution = desc.Resolution;
ovr_Destroy(session);
ovr_Shutdown();
 }
```

Como se puede ver, ovr_initialize es llamado antes de otras funciones y ovr_Shutdown después que todas las funciones y antes de salir del programa. Entre estas llamadas a función se deberán crear objetos OVR para acceder al estado actual de seguimiento de movimientos, además de renderizar la imagen de la aplicación.

En este ejemplo, ovr_Create(&session,&luid) crea el entorno para gestionar el visor yovr_Destroy debe ser llamado para detener la gestión antes de cerrar el acceso a la librería OVR. Podemos usar ovr_GetHmdDesc para obtener una descripción del visor. Si el visor Oculus Rift no está conectado, ovr_Create(&session,&luid) retornara un error, a no ser que este activado un visor virtual mediante RiftConfigUtil. Aunque el visor virtual no proporcionará ninguna entrada de sensor, puede ser útil para depurar sin necesidad de tener el dispositivo físico. El acceso a un gestor del visor se realiza mediante ovr_GetHmdDesc(session). Las propiedades de un gestor del visor son las siguientes:

Field	Type	Description
`Type`	ovrHmdType	Type of the HMD, such as `ovr_DK2` or `ovr_DK2` .
`ProductName`	char[]	Name of the product as a string.
`Manufacturer`	char[]	Name of the manufacturer.
`VendorId`	short	Vendor ID reported by the headset USB device.
`ProductId`	short	Product ID reported by the headset USB device.
`SerialNumber`	char[]	Serial number string reported by the headset USB device.
`FirmwareMajor`	short	The major version of the sensor firmware.
`FirmwareMinor`	short	The minor version of the sensor firmware.
`AvailableHmdCaps`	unsigned int	Capability bits described by `ovrHmdCaps` which the HMD currently supports.
`DefaultHmdCaps`	unsigned int	Default capability bits described by `ovrHmdCaps` for the current HMD.
`AvailableTrackingCaps`	unsigned int	Capability bits described by `ovrTrackingCaps` which the HMD currently supports.
`DefaultTrackingCaps`	unsigned int	Default capability bits described by `ovrTrackingCaps` for the current HMD.
`DefaultEyeFov`	ovrFovPort[]	Recommended optical field of view for each eye.
`MaxEyeFov`	ovrFovPort[]	Maximum optical field of view that can be practically rendered for each eye.
`Resolution`	ovrSizei	Resolution of the full HMD screen (both eyes) in pixels.
`DisplayRefreshRate`	float	Nominal refresh rate of the HMD in cycles per second at the time of HMD creation.

Ilustración 11 – Propiedades del gestor del visor

La descripción del sensor (ovrTrackerDesc) puede obtenerse mediante ovr_GetTrackerDesc(sensor).

Sus propiedades son las siguientes

–

Field	Type	Description
FrustumHFovInRadians	float	The horizontal FOV of the position sensor frustum.
FrustumVFovInRadians	float	The vertical FOV of the position sensor frustum.
FrustumNearZInMeters	float	The distance from the position sensor to the near frustum bounds.
FrustumNearZInMeters	float	The distance from the position sensor to the far frustum bounds.

Ilustración 12 - Parámetros del sensor

3.2.3.3 Seguimiento de la cabeza y sensores

El hardware de Oculus Rift contiene sensores de posición y movimiento. Estos sensores proporcionan información en 3 ejes ortogonales (X,Y,Z). Específicamente, el hardware integra 3 giróscopos, 3 acelerómetros y 3 magnetómetros.

Estos sensores permiten determinar la posición, el ángulo y la aceleración del movimiento del casco. La información de cada uno de los sensores es combinada para determinar la orientación en el mundo real del usuario y de esta forma poder sincronizar su visión del mundo virtual.

Una vez que ovrSession es creado, podemos obtener la posición de la cabeza y su orientación mediante la función ovr_GetTrackingState. El siguiente ejemplo muestra un ejemplo de uso:

```
// Query the HMD for ts current tracking state.
ovrTrackingStatets = ovr_GetTrackingState(session, ovr_GetTimeInSeconds(), ovrTrue);
if (ts.StatusFlags& (ovrStatus_OrientationTracked | ovrStatus_PositionTracked))
{
ovrPosef pose = ts.HeadPose.ThePose;
    ...
}
```

Este ejemplo inicializa los sensores con la orientación, corrección de giro y seguimiento de posición. Si el sensor no está disponible durante el la llamada pero es conectado más tarde, el sensor es automáticamente habilitado por el SDK. Después de que los sensores son inicializados, el estado del sensor es obtenido mediante la función ovr_GetTrackingState. Por último la información de ovrPoseStatef incluye los seis grados de libertad (6DoF) del seguimiento de la cabeza, incluyendo la orientación, posición y sus primera y segunda derivadas.

El valor de pose es declarado como un punto absoluto en el tiempo y típicamente corresponde al tiempo en el futuro en el que un frame será mostrado en pantalla. Para facilitar la predicción, ovr_GetTrackingState toma un tiempo absoluto, en segundos como segundo argumento. El actual valor del tiempo absoluto puede ser obtenido con la llamada a ovr_GetTimeInSeconds. Si el tiempo pasado dentro de ovr_GetTrackingState es el tiempo actual o anterior, el estado de seguimiento retornado estará basado en la última lectura del sensor sin predicciones. En una aplicación práctica, sin embargo, se deberá usar el tiempo real del equipo mediante GetPredictedDisplayTime.

El valor generado incluye un vector de posición en 3D y un cuaternión de orientación. La orientación se entrega como una rotación, como se ilustra en la siguiente figura:

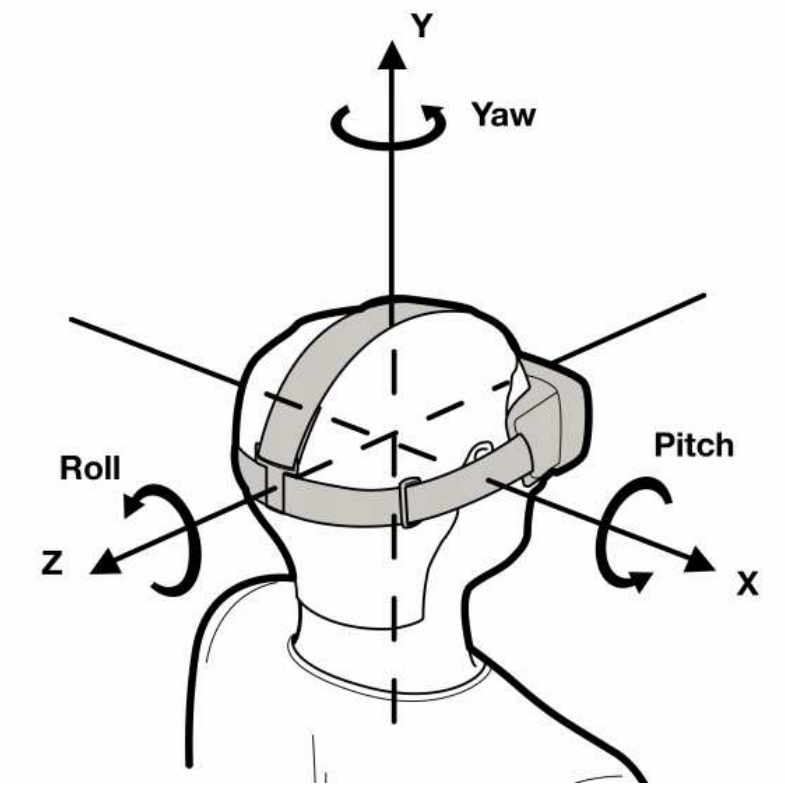


Ilustración 13 - Ángulos en los tres planos

El plano x-z está alineado con el suelo independientemente de la orientación de la cámara. Como se ve en el diagrama, el sistema de coordenadas se utiliza las siguientes definiciones de los ejes:

- Y es positivo en la dirección de arriba
- X es positivo en la derecha
- Z es positivo en la dirección hacia atrás

La rotación se mantiene como un cuaternión unidad, pero también puede darse en forma de ángulos de Euler. Una rotación positiva es en sentido anti-horario (CCW, la dirección de las flechas de rotación en el diagrama) cuando se mira en la dirección negativa de cada eje. Las rotaciones de los componentes son:

- El pitch es la rotación alrededor de X, positivo cuando se mira hacia arriba.
- El yaw es la rotación alrededor de Y, positivo cuando se mira hacia la izquierda.
- El roll es la rotación alrededor de Z, positivo cuando se tuerce la cabeza hacia la izquierda en el sobre plano XY.

La forma más sencilla de extraer de yawn-pitch-roll de ovrPose es utilizar las clases de ayuda matemáticas que se incluyen con la librería. En el siguiente ejemplo se utiliza la conversión directa para asignar ovrPosef a la clase C++ Posef equivalente. A continuación, se pueden utilizar los Quatf::GetEulerAngles<> para extraer los ángulos de Euler en el orden de rotación del eje deseado.

Todos los tipos C matemáticos sencillos proporcionados por OVR como ovrVector3f y ovrQuatf son tipos C++ que proporcionan constructores y operadores que facilitan el cálculo.

Si una aplicación utiliza un sistema de coordenadas de mano izquierda, se puede utilizar la función ovrPosef_FlipHandedness para cambiar el sistema de referencia ovrPosefde mano-derecha proporcionada por ovr_GetTrackingState, ovr_GetEyePoses, o funciones ovr_CalcEyePoses de mano-izquierda. Se debe tener en cuenta que el RenderPose y QuadPoseCenterrequested para los ovrLayers todavía deben utilizar el sistema de coordenadas de mano derecha.

3.2.4 Renderizando para Oculus Rift

Oculus Rift requiere una visión estéreo mediante una pantalla dividida con la corrección de distorsión para cada ojo para cancelar la distorsión relacionada con la lente.

Ilustración 14 - Vista de la demo incluida en el SDK

La corrección de la distorsión puede ser complicada, ya que los parámetros de distorsión son distintos para cada tipo de lente. Para facilitar el desarrollo, Oculus SDK se encarga de la corrección de la distorsión de forma automática, como un proceso de Oculus Compositor.

Con el SDK de Oculus haciendo una gran parte del trabajo, el trabajo principal de la aplicación es llevar a cabo la simulación y hacer que la cámara siga al movimiento del casco. Cada vista estéreo se puede representar en una o dos texturas individuales y se transfieren al compositor llamando a la función ovr_SubmitFrame. Oculus Rift requiere que la escena sea representada en estéreo mediante una pantalla dividida, con la mitad de la pantalla utilizada para cada ojo.

Cuando se utiliza Rift, el ojo izquierdo ve la mitad izquierda de la pantalla, y el ojo derecho ve la mitad derecha.

Aunque varía de persona a persona, las pupilas de los ojos humanos están separadas aproximadamente 65 mm. Este valor se conoce como la distancia interpupilar (IPD). Las cámaras dentro de la aplicación deben estar configuradas con la misma separación.

Las lentes de Rift magnifican la imagen para proporcionar un amplio campo de visión (FOV) que mejora la inmersión. Sin embargo, este proceso distorsiona la imagen de manera significativa. Si el motor llegara a mostrar las imágenes originales en el Rift, el usuario podría observar las imágenes distorsionadas.

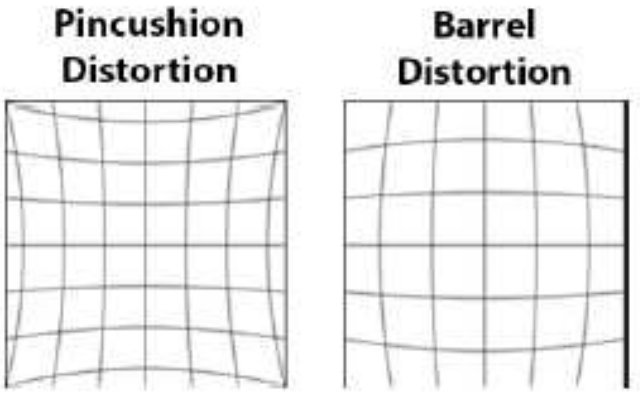


Ilustración 15 - Distorsión de las lentes, y distorsión para compensar

Para contrarrestar esta distorsión, el SDK aplica post-procesamiento a las vistas con una distorsión de barril opuesta para que las dos se anulan entre sí, lo que resulta en una visión sin distorsiones para cada ojo.

Además, el SDK también corrige la aberración cromática, que es un efecto de separación de colores en los bordes causado por la lente. A pesar de que los parámetros exactos de distorsión dependen de las características del objetivo y la posición del ojo respecto a la lente, el SDK de Oculus se encarga de todos los cálculos necesarios en la generación de la malla de distorsión.

Cuando la prestación, los ejes de proyección deben ser paralelos entre sí, como se ilustra en la siguiente figura. Las vistas izquierda y derecha son completamente independientes entre sí. Esto significa que configuración de la cámara es muy similar a la utilizada para la representación no estéreo normal, a excepción de que las cámaras se desplazan hacia los lados para ajustar cada región del ojo.

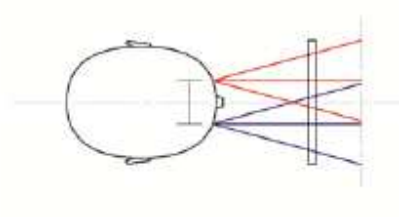


Ilustración 16 - Cono de visión de ambos ojos

En la práctica, las proyecciones del Rift están a menudo ligeramente fuera del centro, porque la nariz está en el medio. Pero los puntos de vista de los ojos izquierdos y derecho del Rift son totalmente independientes entre sí, a diferencia de puntos de vista estéreo generados por un televisor o una pantalla de cine.

Las dos cámaras virtuales en la escena se colocarán de manera que apunten en la misma dirección (determinado por la orientación del visor en el mundo real), y tal que la distancia entre ellas es la misma que la distancia entre los ojos, o la distancia interpupilar (IPD). Esto normalmente se realiza añadiendo

el vector de traslación ovrEyeRenderDesc::HmdToEyeOffset al componente de traducción de la matriz de punto de vista.

Aunque las lentes del Rift están aproximadamente a la distancia correcta para la mayoría de los usuarios, que pueden no coincidir con la distancia interpupilar en personas concretas. Sin embargo, debido a la forma en la que la óptica está diseñada, cada ojo podrá ver de forma correcta.

Es importante que el software haga que la distancia entre las cámaras virtuales coincida con el IPD del usuario. Dicho valor se puede encontrar en su perfil (definido en la utilidad de configuración).

3.3 ODROID XU3

ODROID es una serie de placas orientadas a dispositivos móviles creadas por Hardkernel Co., Ltd., una empresa de hardware abierto localizada en Corea del Sur.

El nombre de ODROID viene dado por Open+Android aunque el hardware en realidad no es abierto porque algunas partes del diseño pertenecen a la empresa. Muchos sistemas ODROID son capaces no solo de funcionar con Android, sino con algunas distribuciones normales de Linux, como por ejemplo Ubuntu.

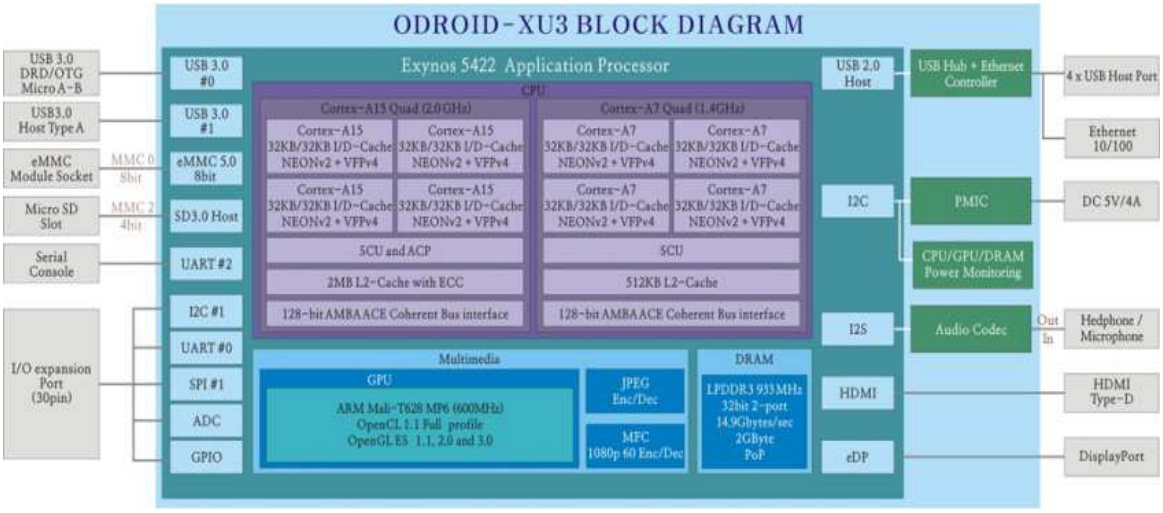


Ilustración 17 - Esquema de Odroid y su SoC

3.3.1 Plataforma hardware

Exynox es un SoC (Systemon Chip) que cuenta con 8 núcleos: 4 ARM Cortex-A15 de altas prestaciones y 4 ARM Cortex-A7 de bajo consumo.

Utiliza la tecnología ARM big.LITTLE con la cual cuando es requerida una gran potencia de computo se habilitan los núcleos con más potencia de computo, mientras que cuando el sistema tiene poca actividad solo funcionan los de menor consumo.

Los Cortex cuentan con una arquitectura Harvard, tamaño de palabra de 32 bits y un bus de datos de 32 bits. Debido a su arquitectura Harvard, la CPU tiene una cache L1 de instrucciones y otra de datos separada.

En la cache se instrucciones solo se almacenan las instrucciones para ser leídas mientras que en la cache de datos la CPU realizará además de lecturas, escrituras que más adelante serán enviadas a caches de mayor nivel y memoria principal para mantener la coherencia.

El procesador cuenta con un repertorio de instrucciones RISC (ReducedInstruction Set Computing), lo cual significa que cuenta con pocas instrucciones y con estructura regular, pero muy rápidas y fáciles de

implementar al contrario de lo que pasa en las instrucciones de tipo CISC (Complex Instruction Set Computing) que cuentan con una gran cantidad de instrucciones con propósitos muy específicos, lentas y costosas.

Exynox cuenta con un procesador gráfico llamado ARM Mali-T628, cuyo objetivo es la aceleración hardware para la visualización de imagen. Su arquitectura está estructurada alrededor de grandes bloques de procesamiento en paralelo. Mali-T628 cuenta con soporte para OpenGL ES 3.0.

El co-proceador ARM Neon SIMD (Single instruction, multiple data) es usado para diferentes procesos multimedia y señales digitales. Acelera la velocidad de los algoritmos de procesado de señal y aumenta el rendimiento en comparación con un procesador de propósito general, ya que en una misma instrucción puede gestionar conjuntos de datos en paralelo.

En cuanto a componentes externos, la placa tiene una interfaz de RAM que soporta doble canal DDR2, LPDDR2, LPDDR3, DDR3L SDRAM.

Soporta una resolución de 2560x1600 pixeles mediante una interfaz microHDMI y cuenta con un módulo que soporta GPS.

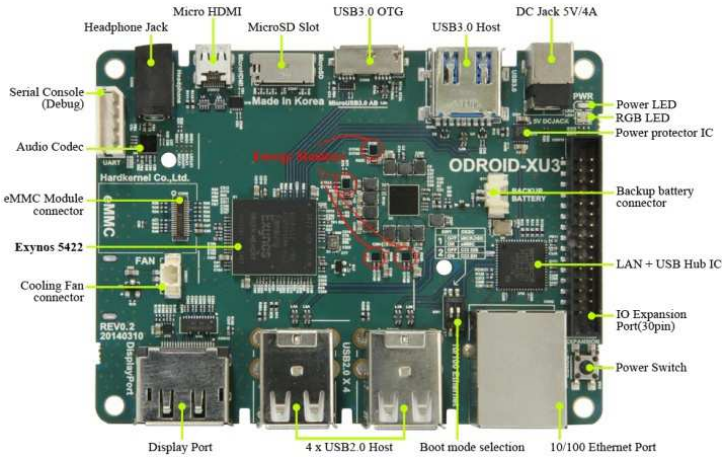


Ilustración 18 - Odroid XU3

3.3.2 Desarrollo de aplicaciones en Odroid

3.3.2.1 Ubuntu

El desarrollo de aplicaciones en Odroid bajo Ubuntu es similar al desarrollo en PC con el mismo sistema operativo. La distribución de Ubuntu para Odroid incluye el compilador gcc, que compila directamente para ARM, por lo que podemos compilar directamente en la placa.

Lo único que debemos tener en cuenta son las dependencias entre librerías, puesto que algunas requieren librerías del sistema que no son compatibles con el hardware o no están bien optimizadas.

Como cualquier distribución de Ubuntu, esta cuenta con su repositorio con aplicaciones ya compiladas que resuelven sus dependencias al ser instaladas y a las cuales se accede con apt-get install.

3.3.2.2 Android

Android está limitado en cuanto al desarrollo sobre el sistema directamente, por lo que no queda más remedio que hacer una cross-compilación, es decir, compilar en PC y después ejecutar en la Odroid. Para ello Google facilita el SDK de Android y un IDE con todas las herramientas integradas, Android Studio.

Además el sistema Android cuenta con la opción de realizar un debugging mediante USB y el SDK incluye el programa adb (Android Debug Bridge) que permite obtener logs del sistema y enviar aplicaciones para que sean ejecutadas directamente.

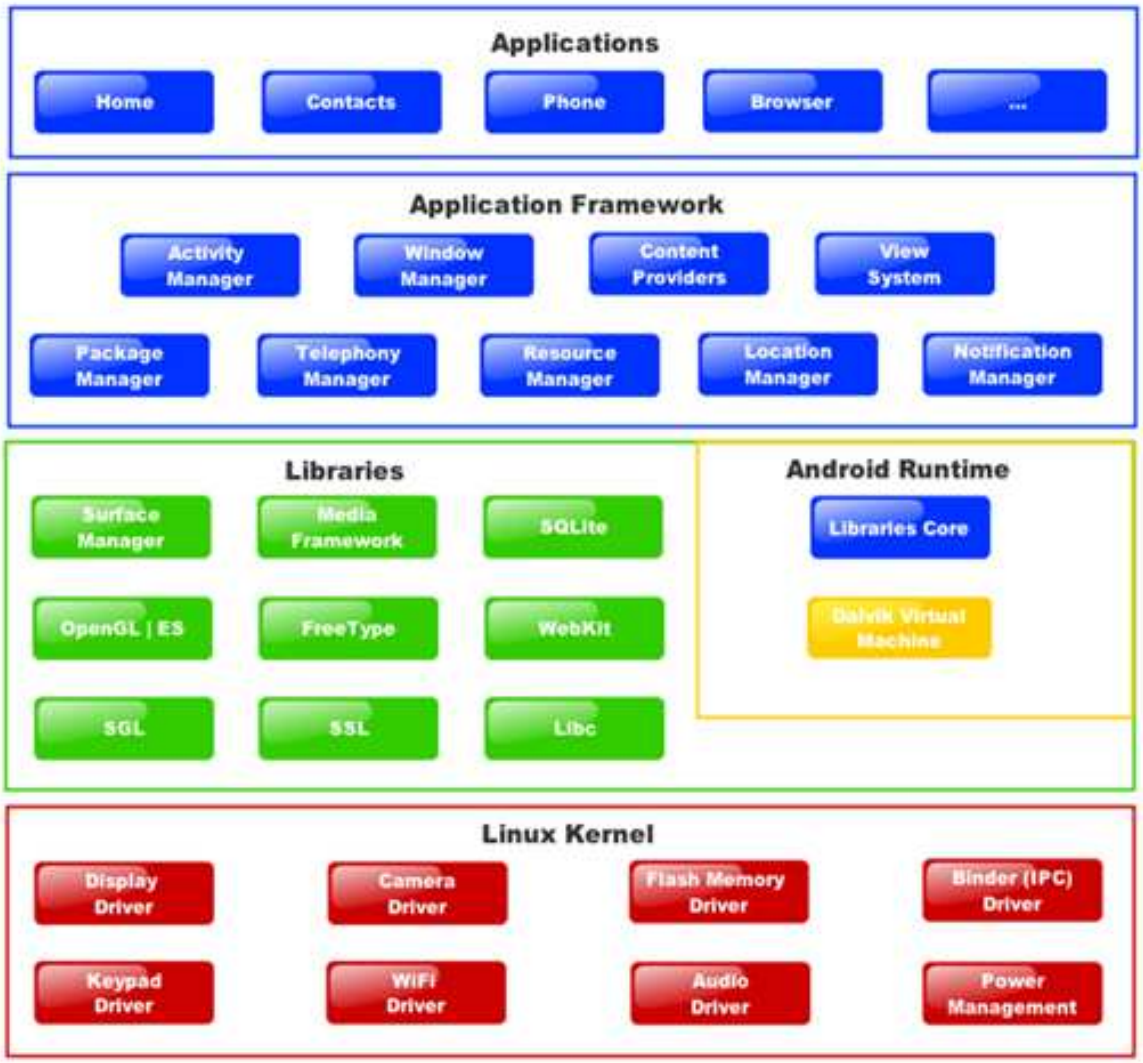


Ilustración 19 - Capas de Android

Otra limitación cuando se utiliza el sistema Android es que las aplicaciones solo pueden programarse en Java. No obstante esas aplicaciones pueden utilizar librerías (.so) desarrolladas en C o C++ que pueden acceder directamente al sistema.

3.4 UNITY

Unity es un motor de videojuego multiplataforma creado por Unity Technologies. Unity está disponible como plataforma de desarrollo para Microsoft Windows, OS X y Linux, y permite crear juegos para Windows, OS X, Linux, Xbox 360, PlayStation 3, Playstation Vita, Wii, Wii U, iPad, iPhone, Android y Windows Phone. Gracias al plugin web de Unity, también se pueden desarrollar videojuegos de navegador para Windows y Mac. Unity tiene dos versiones: Unity Professional y Unity Personal

Ilustración 20 - Logo de Unity

Unity puede usarse junto con aplicaciones que crean objetos gráficos, como 3ds Max, Maya, Softimage, Blender, Modo, ZBrush, Cinema 4D, Cheetah3D, Adobe Photoshop, Adobe Fireworks y Allegorithmic Substance. Los cambios realizados a los objetos creados con estos productos se actualizan automáticamente en todas las instancias de ese objeto durante todo el proyecto sin necesidad de volver a importarlos.

El motor gráfico utiliza Direct3D (en Windows), OpenGL (en Mac y Linux), OpenGL ES (en Android y iOS), e interfaces propietarias (en Wii). Tiene soporte para mapeado de relieve, reflexión de mapeado, mapeado por paralaje, pantalla de espacio oclusión ambiental (SSAO), sombras dinámicas utilizando mapas de sombras, render a textura y efectos de post-procesamiento de pantalla completa.

Ilustración 21 - Vista de Unity

Se usa ShaderLab, language para el uso de shaders, que soporta tanto la programación declarativa de los programas de función fija de tuberías y shader GLSL o escritas en Cg. Un shader puede incluir múltiples variantes y una especificación declarativa de reserva, lo que permite a Unity detectar la mejor variante para la tarjeta de vídeo actual y si no son compatibles, recurrir a un shader alternativo que puede sacrificar características para una mayor compatibilidad.

El soporte integrado para Nvidia

(antes Ageia), el motor de física PhysX, (a partir de Unity 3.0) con soporte en tiempo real para mallas arbitrarias y sin piel, ray casts gruesos, y las capas de colisión.

Los script que precisa se basan en Mono, la implementación de código abierto de .NET Framework. Los programadores pueden utilizar UnityScript (un lenguaje personalizado inspirado en la sintaxis ECMAScript), C# o Boo (que tiene una sintaxis inspirada en Python). A partir de la versión 3.0 añade una versión personalizada de MonoDevelop para la depuración de scripts.

Unity también incluye Unity Asset Server - una solución de control de versiones para todos los assets de juego y scripts, utilizando PostgreSQL como backend, un sistema de audio construido con la librería FMOD, con capacidad para reproducir audio comprimido Ogg Vorbis, reproducción de vídeo con códec Theora, un motor de terreno y vegetación , con árboles con soporte de billboarding, determinación de cara oculta con Umbra, una función de iluminación lightmapping y global con Beast, redes multijugador RakNet y una función de búsqueda de caminos en mallas de navegación.

3.5 GOOGLE CARDBOARD

Una posible alternativa al uso de Oculus Rift es el uso de Cardboard, que únicamente son dos lentes que se ajustan al dispositivo móvil directamente. Debido a que la móvil ya cuenta con sensores que permiten el posicionamiento, no es necesario ningún tipo de complemento. Únicamente debemos comprobar si el dispositivo es compatible con las aplicaciones de cardboard, y utilizar el SDK para la programación de la aplicación.

Ilustración 22 - Google Cardboard

4 DISEÑO DEL SISTEMA

En la primera parte de este trabajo se han planteado unos objetivos y unos requisitos a verificar por parte del sistema a diseñar. A partir de un análisis del estado del arte y de los elementos disponibles, se han seleccionado una serie de dispositivos, herramientas y librerías para desarrollar la aplicación.

Una vez elegidas, se ha procedido a la instalación de las herramientas y librerías seleccionadas. La mayoría de ellas se han podido obtener mediante el gestor de paquetes apt-get de Ubuntu. Otras han sido instaladas manualmente compilando su código fuente. Toda la aplicación será desarrollada en C++, con programación orientada a objetos.

Además se ha escogido como entorno de desarrollo (IDE) la herramienta Eclipse CDT, ya que facilita la gestión del proyecto software y la generación de archivos de compilación (makefile).

Como ya se explico anteriormente, la aplicación a desarrollar utilizará las siguientes librerías:

- Ogre3D: Para la generación de escenas en 3D
- V4L: Para poder controlar las cámaras
- libOVR: Para la adquisición de datos de Oculus y generar la distorsión

La configuración de Eclipse incluye todas rutas de las cabeceras de Ogre, V4L y OVR, asi como todas las librerías que se van a requerir para compilar el sistema. Una vez configurado correctamente Eclipse, se puede empezar a desarrollar la aplicación de realidad aumentada ARES.

4.1 INTRODUCCIÓN Y ESQUEMA GENERAL

Antes de empezar a desarrollar ARES, es necesario tener claro todos los módulos que van a ser desarrollados y la función de cada uno.

Se ha decidido dividir la aplicación en los siguientes bloques:

- Un modulo que se encargue de toda la inicialización de Ogre y OVR, y la conexión entre ambas librerías.
- Un modulo que gestione las cámaras.
- Un modulo que cree menús y mapas (GUI).
- Un modulo que se encargue de crear una conexión de entrada para permitir un control remoto.
- Por último, un modulo que conecte todos los demás y los gestione.

Además de decidir los módulos que se van a crear, se ha de decidir la jerarquía de cada uno de ellos, las dependencias, las propiedades y los métodos que necesitarán para realizar su función.

El siguiente esquema es un resumen de las principales clases que tendrá la aplicación, sus métodos y propiedades principales.

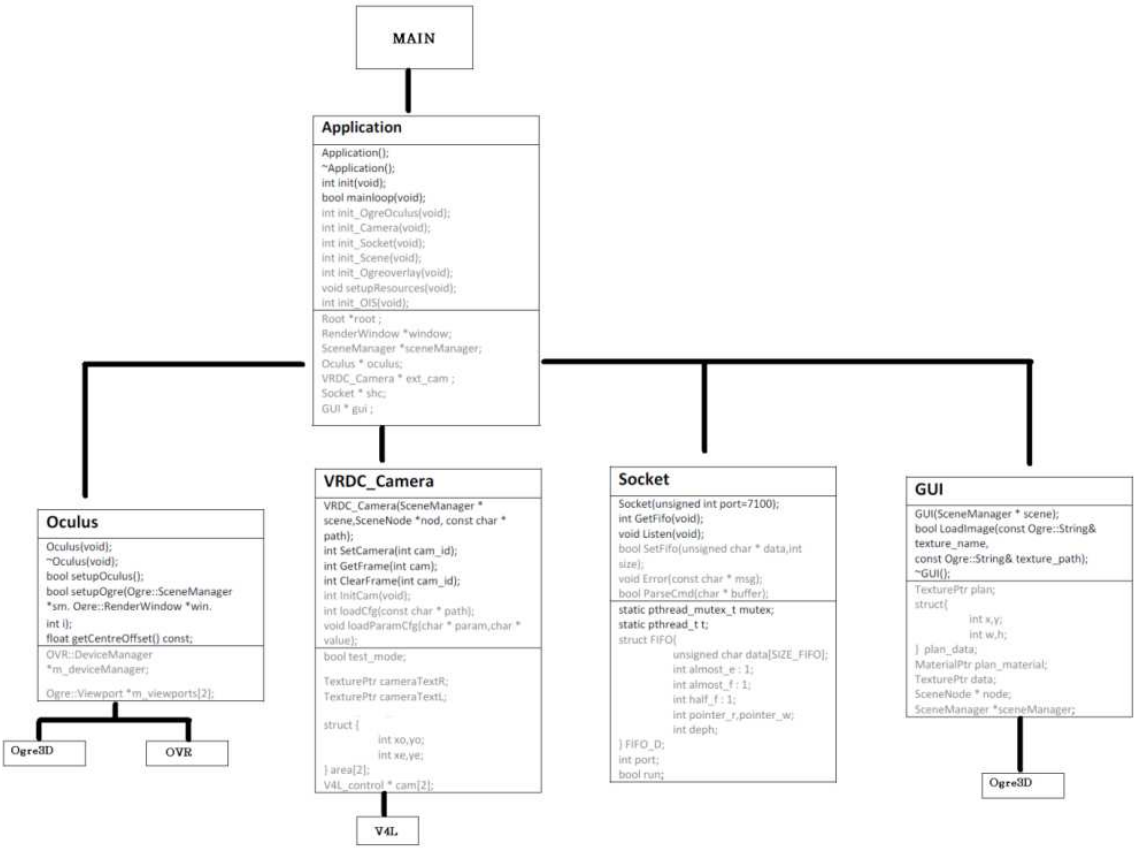


Ilustración 23 - Jerarquía de clases de ARES con sus clases y propiedades más importantes. En gris se marcan las propiedades privadas, en negro las públicas.

Se puede resumir todo el sistema a una jerarquía de dos niveles y 5 clases:

- Programa principal - Application: Es la clase principal que inicializa el resto y las interconecta entre si
- Adaptación de imagen y control de sensores - Oculus: Es la clase encargada de comunicarse con la librería OVR, adquirir información de sensores y deformar la imagen para que pueda ser vista por las Oculus. También conecta la posición de los sensores de giro con la cámara de Ogre
- Captura de imagen de cámaras - VRDC_Camera: Esta clase se comunica con la librería de video de Linux y codifica la imagen de las cámaras en texturas para el Ogre
- Acceso remoto - Socket: Esta clase se comunica con el servidor para el posicionamiento de los objetos
- Interfaz gráfico de usuario - GUI: Esta clase agrega una interfaz para el usuario

Además de la jerarquía interna de la aplicación son necesarios archivos externos con scripts, configuraciones y tablas de ajuste

- cfg: En esta carpeta están todos los archivos de configuración y rutas con plugins y dependencias.
- yml: En esta carpeta se guardan los parámetros de ajuste de las cámaras para alinearlas correctamente y corregir sus desviaciones
- plugin: En esta carpeta están los plugins requeridos por Ogre
- lib: En esta carpeta están las librerías dinámicas de Ogre
- media: Contiene los modelos y texturas requeridos por la aplicación
- src: Es el código fuente de la aplicación
- calibration: Contiene una aplicación para la generación de yml usando un patrón de ajuste

ARES, como la mayoría de aplicaciones, empezará desde la función main, que está situada en el archivo **core.cpp.**

En main será instanciado el objeto Application, que una vez inicializado entra en el bucle principal.

Como en todas las aplicaciones con interfaz de usuario, el bucle principal solo finaliza cuando el usuario lo indica. Por ello el bucle principal será infinito, y solo finalizará si es pulsada la tecla de finalización o el programa encuentra un fallo crítico. También el sistema operativo puede finalizar la aplicación si la CPU o el gestor de memoria detectan un fallo, tal como una división entre cero o un error de segmentación.

Cuando este bucle se rompe, se destruye todas las instancias de clase, se libera la memoria y el programa finaliza. Todos los mensajes de error e información son registrados en un archivo log (Ogre.log) por lo que si ocurre algún error, puede saberse su procedencia revisando ese archivo.

4.2 PROGRAMA PRINCIPAL - APPLICATION

Application

Application();
~Application();
int init(void);
bool mainloop(void);
int init_OgreOculus(void);
int init_Camera(void);
int init_Socket(void);
int init_Scene(void);
int init_Ogreoverlay(void);
void setupResources(void);
int init_OIS(void);

Root *root ; RenderWindow *window; SceneManager *sceneManager; Oculus * oculus; VRDC_Camera * ext_cam ; Socket * shc; GUI * gui ; Ogre::Euler g_bodyOrientation; Vector3 * g_bodyPosition; Quaternion * g_headOrientation ; bool g_flying ; Ogre::Euler g_sinbadLook; Light *light ; SceneNode *node; Entity *ent ; Ogre::Bone *g_sinbadHead ; OIS::InputManager *g_inputManager ; OIS::Keyboard *g_keyboard ; OIS::Mouse *g_mouse ; OIS::ParamList pl; std::ostringstream windowHndStr;

Ilustración 24 - Propiedades y métodos de Application

Es la clase principal de ARES. Se crea en main y en su constructor se crea el resto de clases y se las inicializa. Esta clase está definida en **Application.cpp.**

Application se encarga en primer lugar de inicializar, en segundo lugar crea y mantiene el bucle principal, y por ultimo finaliza.

Podemos dividir Application en 3 estados:

- Inicialización (Init)
- Bucle principal (mainloop)
- Finalización (End)

Para inicializar todo, se deberá llamar al método init. Esto se hará desde main y comprobará si se ha producido algún error.

En el caso de que todo se inicialice correctamente, se llamará a un método llamado mainloop que contiene el bucle principal del programa. Este calculará los tiempos de cada iteración para llevar un buen ratio de frames a la vez que refrescará la pantalla.

Además, contiene propiedades de la librería OIS, la cual se encarga de comunicarse con el ratón y el teclado para poderse mover dentro del escenario.

Esta funcionalidad puede deshabilitarse mediante una directiva de compilación, ya que ARES está pensado para que el movimiento se haga mediante geolocalización o triangulación local.

Las propiedades principales de Ogre son inicializadas aquí, y únicamente se pasará lo requerido por referencia al resto de clases necesarias.

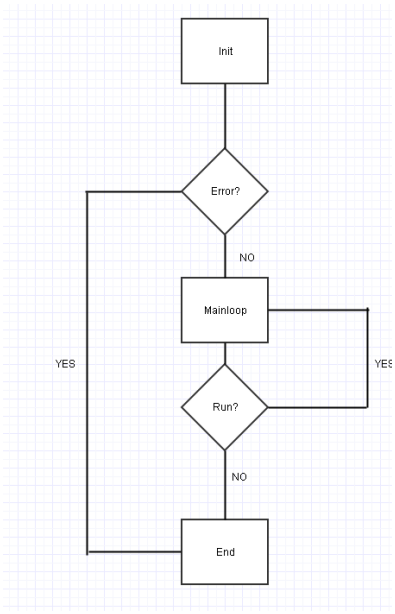


Ilustración 25 - Diagrama de Application

4.2.1 Inicialización

Inicializa el resto de clases, y prepara Ogre3D y OIS si está activo.

En el caso de que existiese algún error, se registraría un mensaje de log y el programa finalizaría.

La forma en la que se inicializan el resto de clases se explicará a continuación

4.2.1.1 Inicializando Ogre y Oculus

El primer método para inicializar Ogre es el constructor de la clase Root, que ha sido explicado con anterioridad. Como argumentos se deben pasar las rutas de todos los archivos de configuración.

En este caso, necesitamos 3 archivos: Rutas de los plugins a cargar (reflejados en el archivo **plugins.cfg)**, configuración gráfica (**config.cfg)** y archivo de logs **(ogre.log)**.

Los plugins definidos en plugins.cfg son imprescindibles, ya que sin ellos no contaremos con ninguna característica gráfica. La configuración grafica no es necesaria, ya que Ogre nos muestra una ventana con las diferentes opciones a seleccionas, como la librería gráfica (OpenGL en Linux DirectX en Windows), resolución y demás opciones graficas.

En ARES si encuentra el archivo de configuración, no mostrara la ventana de configuración y continuara la ejecución con la configuración que se encuentre en el archivo.

Por eso, si se desea cambiar algo o se ha cambiado la plataforma, la mejor opción es eliminar el archivo de configuración **config.cfg.**

Después de crear root, se creara la clase SceneManager y Windows. SceneManager gestiona todos los elementos dentro de la escena y Windows se encarga de la comunicación con el escritorio para mostrar por pantalla la imagen renderizada.

Tal como se explico en el apartado 3.1 de este trabajo ambas clases se encuentran dentro de root, y para su gestión se obtendrá un puntero a ambas que será enviada a la clase Oculus.

Además, se cargarán todos los recursos necesarios (Texturas, modelos, imágenes y scripts) para su posterior uso. Para ello se llamará al método `setupResources ()` y se le indicará el archivo con las rutas de los diferentes recursos que usará la aplicación, **resource.cfg.**

En esta versión, la carga de recursos es estática ya que hay pocos, pero en versiones posteriores posiblemente sea necesaria una carga a demanda para no saturar la memoria.

4.2.1.2 Inicializando OIS

La librería OIS es la encargada de capturar las teclas pulsadas por el usuario, el movimiento del ratón y los clicks del ratón.

Para usarla, lo único que debemos hacer es vincular la instancia window con el handle de OIS. Una vez hecho, se creara un inputManager que a su vez creará el objeto de entrada de teclado (`OIS::Keyboard`) y de ratón (`OIS::Mouse`), que podremos usar para comprobar si hay pulsaciones del teclado o el ratón.

4.2.1.3 Inicializando la escena

La escena de la aplicación desarrollada tendrá una luz, un cubo y un modelo de ejemplo con animación. En este método se crearán dichos objetos y se posicionarán en la escena.

Para crear la luz, usaremos `sceneManager->createLight("light")` mientras que para crear el cubo y el modelo se usará `sceneManager->createEntity("Cube","Cube.mesh")`.

Tal como se explico anteriormente, todos los elementos de la escena deben crearse desde el sceneManager.

En una versión futura, se crearán objetos de forma dinámica, por lo que no deberán crearse objetos directamente en el código fuente.

4.2.1.4 Inicializando Overlay y GUI

Este método instanciará la interfaz de usuario, que será la encargada de crear imágenes, botones y demás elementos sobre la escena.

4.2.1.5 Inicializando la cámara

Se instanciará Camera, que será la clase que se encargue de comunicarse con las cámaras y capturar imágenes. ClearFrame limpia el buffer de cada una de las cámaras, para evitar que durante la inicialización aparezca ruido blanco.

4.2.1.6 Inicializando Socket

Socket creará un thread que se mantendrá a la escucha en el puerto 7100. Dicho elemento se identifica por referencia en la propiedad shc de Oculus.

4.2.2 Bucle principal

El bucle principal se ejecutará mientras la ventana de la aplicación esté abierta. En dicho bucle se realizan las siguientes tareas:

- Detección de si el usuario ha pulsado alguna tecla. Estos eventos se evaluaran en las instancias creadas con anterioridad de teclado y ratón, con los métodos `g_keyboard->isKeyDown()` y `g_mouse->getMouseState()`

- Obtener un paquete de la FIFO del Socket y comprobar si hay que realizar algún cambio. Esto se hará con `type_c = oculus->shc->GetFifo();` y se comprobará el valor de type_c. Si type_c corresponde a desplazamiento, tendrá el mismo efecto que pulsar la tecla correspondiente del teclado.
- Leer los sensores de Oculus y cambiar orientación de las cámaras que observan el mundo virtual. Se podrá hacer con `oculus->getOrientation()`
- Capturar imágenes de las cámaras físicas y pasarlas a las texturas asignadas. Se obtendrá el frame con `ext_cam->GetFrame(0)` y se copiara a la textura con `ext_cam->SetCamera(0)`

Después de finalizar el bucle, habrá una pequeña pausa para mantener constante la frecuencia de frame.

4.2.3 Finalización

Al estado de finalización se llega si el usuario solicita salir de la aplicación o se detecta algún error. Únicamente en este punto se procede a liberar la memoria de las diferentes clases y objetos y a salir de la misma.

4.3 ADAPTACIÓN DE IMAGEN Y CONTROL DE SENSORES - OCULUS

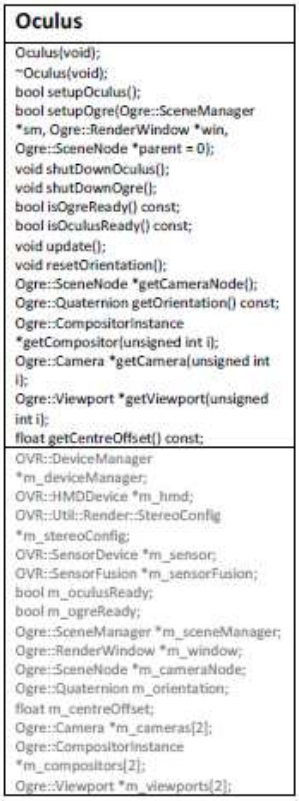


Ilustración 26 - Clase Oculus

Esta clase será la encargada de leer los sensores, y la distorsionar de la imagen para que pueda ser vista correctamente con las gafas. La clase se encuentra definida en **OgreOculus.cpp.**

Las fases de inicialización son las siguientes

- Inicialización de Oculus: Se comunicará con la librería OVR para inicializar los sensores, obtener todos los parámetros requeridos y las referencias
- Inicialización de Ogre3D: Creará las cámaras gracias a gestor de escena, y las vistas a partir de la referencia de la ventana de aplicación
- Conexión entre ambos: Se hará dentro de la inicialización de Ogre. Se asignarán los parámetros de la configuración estéreo de OVR a las cámaras y a las vistas.

4.3.1 Inicialización de Oculus

En esta etapa se inicializaran y configuraran los sensores. Los objetos que utilizaremos serán los siguientes, todos ellos dentro de la librería OVR.

- DeviceManager: Controla y facilita el acceso a dispositivos soportados por OVR, tales como HMDs y sensores. Una sola instancia de DeviceManager es normalmente creada al iniciar el programa, permitiendo que dispositivos sean numerados y creados.
- StereoConfig: Controla una escena en estéreo y permite conmutar entre diferentes modos de renderizado estéreo. Para soportarlo StereoConfig mantiene un seguimiento de las variables HMD como son el tamaño de la pantalla, la distancia entre el ojo y la pantalla y la distorsión.
- HMD: Representa un dispositivo Oculus HMD. Una instancia de esta clase es normalmente creada desde el DeviceManager. Después de que el dispositivo sea creada, es posible obtener el objeto del sensor
- Sensor: Es la interfaz del sensor

- SensorFusion: Acumula los datos del sensor y opera con ellos para mantener un seguimiento de la orientación

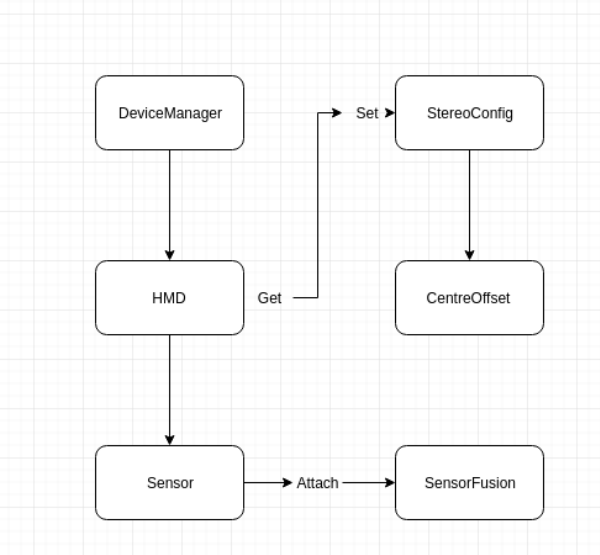


Ilustración 27 - Esquema de los objetos de DeviceManager

Para ello lo primero que se hará es crear un DeviceManager y se obtendrá StereoConfig. De DeviceManager crearemos HMD y se vinculará a StereoCofig. También obtendremos CentreOffset que nos servirá más adelante para la imagen. De HMD obtendremos los sensores y crearemos la clase SensorFusion que se encargara de interpretar los sensores y darnos unos valores de orientación con los que podremos operar.

Entre operación y operación, se comprobará si hay algún error (como por ejemplo que las Oculus no estén conectadas) y, si lo detecta, se desactivará el control de las Oculus y el programa continuara con su ejecución dando los parámetros por defecto.

4.3.2 Inicialización de Ogre3D

Para la inicialización pasaremos 2 argumentos creados en la clase Application:

- SceneManager – Gestor de escena
- RenderWindow – Referencia a la ventana donde se verá la aplicación

Los objetos más importantes de este método son los siguientes:

- Camera: Serán dos cámaras y se crearán dentro de la escena.
- CameraNode: Servirá de unión entre la escena y las cámaras
- ViewPort: Son las vistas dentro de la ventana de la aplicación. En la mayoría de las aplicaciones solo existe una que ocupa toda la ventana, a la que se le asigna una cámara. Pero en esta aplicación se necesitarán dos vistas con una cámara cada una y se repartirán la ventana.
- Material: Incluye todo tipo de parámetros que modifican la forma en la que se verá en pantalla. Parte de los parámetros del material se obtendrán en tiempo de ejecución de la configuración estéreo de OVR. El resto se obtendrán de un script material que se ha incluido como recurso. Habrá dos materials que se asignaran a cada una de las vistas
- Compositor: Muy similares a los materials. Incluye más parámetros que serán requeridos para una correcta visualización. Se cargaran del script compositor incluido como recurso.

Se crearan las cámaras usando el método `createCamera()` del SceneManager. Se necesitarán dos cámaras, la derecha e izquierda, y dos vistas que contendrán cada una de las cámaras.

También se definirán dos materiales para cada una de las vistas, aunque se creará el primero con "`Ogre::MaterialManager::getSingleton().getByName("Ogre/Compositor/Oculus")`" y el segundo se clonara del primero con "`matLeft->clone("Ogre/Compositor/Oculus/Right")`".

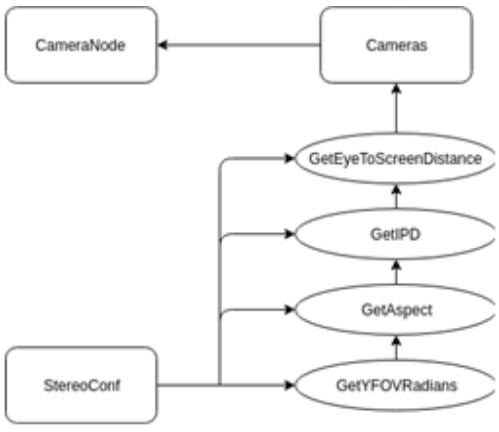


Ilustración 28 - Parámetros de cámara

Después de la creación de las cámaras y los materiales, será necesaria una distorsión para que pueda verse de forma adecuada en las Oculus.

Para ello se declarara un vector de 4 componentes y se le asignará cada valor que se obtendrá de la configuración estéreo de la librería OVR. En el caso de que OVR no haya podido obtener el vector de distorsión se le asignará uno por defecto.

Para asignarle este vector de distorsión a las cámaras, se deberá primero asignar el mismo a los materiales que luego serán asignados a las vistas, que es lo que el usuario verá en pantalla.

Se hará obteniendo el GpuProgramParametersSharedPtr que es una referencia compartida a un conjunto de parámetros referente a la GPU. Se obtendrá mediante "`matRight->getTechnique(0)->getPass(0)->getFragmentProgramParameters()`".

A esta referencia le asignaremos por un lado los parámetros de distorsión que hemos obtenido de OVR mediante "`m_stereoConfig->GetDistortionK(0)`" y por otro lado el centro de la que se obtendrá con "`m_stereoConfig->GetProjectionCenterOffset()`".

El resto de parámetros, al ser fijos, se cargarán mediante un script compositor y otro script material que hemos incluido como recurso. Obtendremos el script compositor mediante"`Ogre::CompositorManager::getSingleton().getByName("Oculus")`" y el script material mediante"`comp->getTechnique(0)->getOutputTargetPass()->getPass(0)>setMaterialName("Ogre/Compositor/Oculus")`"

Después de asignar todos los parámetros a los materiales y compositors, será necesario configurar las cámaras a partir de los parámetros obtenidos por la configuración estéreo de OVR, asignar las cámaras a un nodo y por último a los dos puntos de vista para que puedan ser vistas por el usuario.

Las vistas se obtendrán de la ventana con "`win->addViewport(m_cameras[i], i, 0.5f*i, 0, 0.5f, 1.0f)`" dando como primer parámetro la cámara que usará, el numero de vista y la posición en la ventana.

A cada vista se le asignará un MaterialScheme diferente que se usará más adelante para poder asignar texturas diferentes en un mismo objeto según la vista que use. Eso será útil para que cada cámara se vea por una vista distinta y conseguir así una visión estéreo.

Una vez terminado todo, se asignará al flag `m_ogreReady` el valor true y se mostrará por log un mensaje que confirmará que la carga ha sido exitosa.

4.4 CAPTURA DE IMAGEN DE CÁMARAS - VRCD_CAMERA

Ogre no tiene soporte nativo para streaming, por lo que se debe gestionar mediante otra librería de video como OpenCV o a bajo nivel en la propia aplicación. En este caso se utilizará V4L, una librería de bajo nivel que accede directamente al hardware usando los drivers del USB.

Esto se hace así ya que OpenCV es una librería muy pesada e incluye muchas opciones que no necesitara la aplicación.

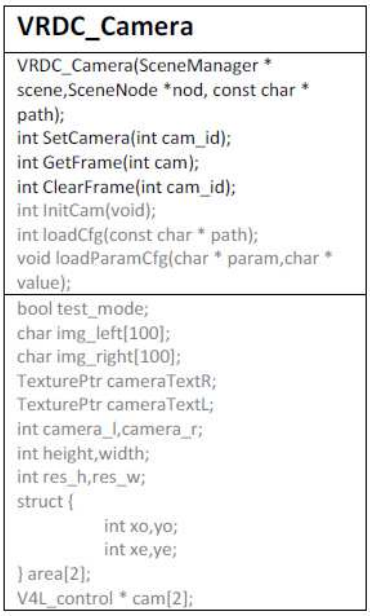


Ilustración 29 - Clase Camera

Esta clase se ocupará de la comunicación entre la librería v4l y Ogre3D. Esta clase está definida en **VRCD_Camera.cpp**

Para ello crea dos clases V4L_control definidas en **V4L_control.cpp** (Una por cada cámara) y dos texturas (Izquierda y derecha). La idea es que cada iteración del bucle principal se obtendrá una imagen de cada cámara y se copiara al buffer de cada textura.

Estas texturas estarán declaradas en Application como textura de fondo a distancia infinita y fija, por lo que todos los elementos del espacio estarán sobre la imagen de la cámara.

Cada textura estará asignada a cada uno de los materiales de las dos vistas, haciendo que cada textura solo sea visible en una vista.

La clase consta de las siguientes etapas

- Carga de configuración (Cfg)
- Inicialización de las cámaras
- Creación de texturas y materiales
- Obtención de frames
- Copiado de imagen en texturas

Hay que tener en cuenta además, que el formato de imagen que entrega la cámara puede no coincidir con el esperado por Ogre, por lo que es necesario decodificarlo.

Hay cámaras que entregan directamente un formato jpeg o mpeg, por lo que se necesitará usar librerías adicionales y optimizadas para su decodificación.

Otra cosa que se deberá tener en cuenta, es que puede que las cámaras no estén totalmente alineadas, por lo que se necesitarán parámetros de ajustes que es posible obtener mediante una calibración.

4.4.1 Calibración de las cámaras y generación de archivo de configuración

La calibración se hará previamente con una aplicación externa que generará el archivo de configuración que ARES utilizará a la hora de colocar las imágenes de las dos cámaras, **camera.cfg**.

Esta calibración es necesaria para que la imagen capturada sea mostrada correctamente en el visor.

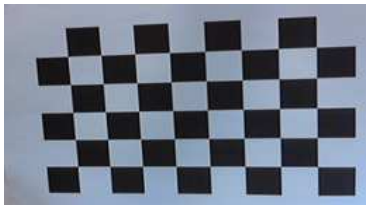


Ilustración 30 - Patrón utilizado para la calibración

Para ello usaremos un patrón de cuadrados negros y blancos que será capturado por ambas cámaras a diferentes distancias y giros.

Es necesario darle a la aplicación de calibración el tamaño de cuadricula y las rutas donde guardará los resultados de los cálculos.

Una vez finalizada, la aplicación de calibración calculará el área de interés de cada una de las cámaras para que estén en línea y lo guardará en el archivo de configuración que utilizará ARES.

4.4.2 Carga de configuración

En esta primera etapa se leerá el archivo de configuración de las cámaras que ha generado la aplicación de calibración y se cargaran siguientes parámetros:

- size_o: Resolución de la cámara
- size_c: Área de interés
- camera0_x: Primer punto x de la primera cámara
- camera0_y: Primer punto y de la primera cámara
- camera0_xe: Segundo punto x de la primera cámara
- camera0_ye: Segundo punto y de la primera cámara
- camera1_x: Primer punto x de la segunda cámara
- camera1_y: Primer punto y de la segunda cámara
- camera1_xe: Segundo punto x de la segunda cámara
- camera1_ye: Segundo punto y de la segunda cámara
- camera_l: Identificación de dispositivo de la cámara de la izquierda
- camera_r: Identificación de dispositivo de la cámara de la derecha
- test_mode: Cuando el test mode está activo, se carga una imagen por defecto en lugar de la cámara
- test_img_r: Nombre de la imagen de test para el lado derecho
- test_img_l: Nombre de la imagen de test para el lado izquierdo

En el archivo de configuración se incluye un modo test por si no se desea utilizar las cámaras en ARES, ya que quizás en un momento determinado no se disponga de ellas. En el modo test se utilizaran unas imágenes fijas de prueba.

4.4.3 Inicialización de las cámaras

Para inicializar las cámaras, crearemos dos instancias de V4L Control dando el ID de dispositivo. Esta clase se comunicará con la librería de Linux y accederá a bajo nivel al dispositivo.

4.4.4 Creación de texturas

Se crearán dos texturas de forma manual, con las dimensiones que han sido cargadas en la configuración de las cámaras. Estas dos texturas se vincularan a cada uno de los materiales de las dos vistas.

En los materiales se desactivará la profundidad y la luz para que estas texturas estén siempre en el fondo y así sean un background de la escena. Por último se creará un objeto plano que ocupará toda la pantalla al que se le asignará los materiales con las texturas y estará en el infinito.

Además este plano estará fijado a las cámaras, por lo que se moverá a la vez que ellas.

4.4.5 Obtención de frames

Dentro de V4L_control hay un buffer que cada vez que se haga una llamada a getFrame se refrescará con el contenido del buffer interno de la cámara. Esto deberá hacerse antes de copiar la imagen dentro de la textura de Ogre.

4.4.6 Copiado de imagen en el buffer de texturas

Para copiar la imagen en la textura, obtendremos el puntero del buffer de la textura (PixelBox), lo bloquearemos para que otros thread propios de Ogre no lo puedan sobrescribir y copiaremos la imagen directamente sobre la referencia del buffer de la textura.

Para ello usaremos `copyRGB(pDest,size)`,un método de V4L_control que se encargará de comparar el tamaño de la región de interés con la resolución de la cámara y su formato. Hay que tener muy en cuenta el formato que entrega la cámara, ya que debemos adaptar el formato al admitido por la textura (RGB).

En el caso que una cámara entregue JPG o MPG necesitaremos librerías adicionales para decodificarlas. Las cámaras usadas en la implementación entregan la imagen en formato YUV (Luminancia y crominancia) por lo que hay que transfórmalo en formato RGB. Para ello debemos calcular cada componente mediante la matriz de conversión siguiente:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1,14 \\ 1 & -0,396 & -0,581 \\ 1 & 2,029 & 0 \end{bmatrix} \begin{bmatrix} Y' \\ U \\ V \end{bmatrix}$$

R = Y + 1.14*V;

G = Y - 0.396*U- 0.58*V;

B = Y + 2.029*U

El formato de la imagen de textura es RGB con un canal Alpha (transparencia), con una profundidad de 32 bits, por lo cual cada color (canal) tiene una anchura de 8 bits, proporcionando un rango entre 0 y 255.

Podemos acceder al puntero de salida como un array de enteros de 32 bits, y asignar a cada iteración un pixel de la siguiente forma:

```
pix[j++] = (((int)r)<<0) | (((int)g)<<8) | ((((int)b)<<16));
```

Puesto que no existe transparencia en la imagen de una cámara, el canal alpha siempre tiene valor 0.

4.5 ACCESO REMOTO - SOCKET

En esta aplicación se requiere que su control no precise interacción de teclado o ratón por parte del usuario. Por eso la mejor, forma para controlar la aplicación es abrir una conexión que se mantendrá a la escucha de información que se le transfiera desde el exterior.

Se definirán una serie de instrucciones que serán enviadas como paquetes con una serie de argumentos. Por ejemplo, se han definido instrucciones que piden a la aplicación que la cámara se mueva en una dirección o que se posicionen en un punto en concreto.

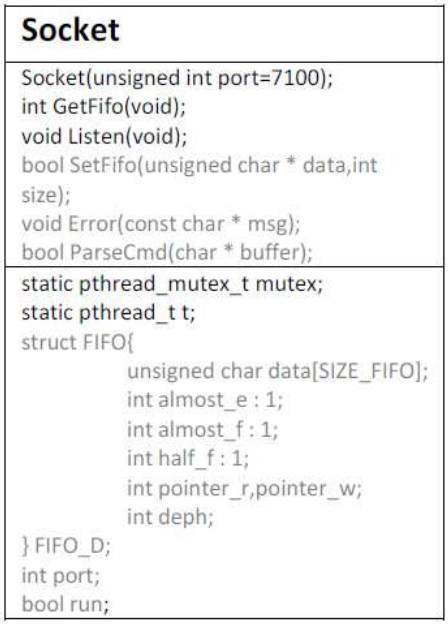


Ilustración 31 - Clase Socket

Todo ello será definido en la clase Socket. Esta clase se encargará de establecer una conexión de escucha. Esta clase está definida en **Socket.cpp.**

En la clase Socket se crea otro hilo que se quedará bloqueado con la conexión y, cada vez que reciba algo, lo guardará en una FIFO.

Esta FIFO será leída por el bucle general y, si contiene acciones, serán ejecutadas por la aplicación. La FIFO está protegida por un mutex para permitir al thread del socket y al bucle principal acceder de forma concurrente al recurso compartido sin problemas.

Una vez se termina de leer la FIFO, se retorna la instrucción leída al bucle general que, en función de su valor, modifica los parámetros del objeto que se requiera.

4.5.1 Creación de la conexión

Para crear la conexión se creará en primer lugar un thread mediante `pthread_create(&t, NULL, Listen_f,(void *)this)`. Dentro de Listen, se creará la conexión con `socket(AF_INET, SOCK_STREAM, 0)`.

4.5.2 Escucha y almacenamiento de la instrucción en la FIFO

Después de crearse la conexión se entrará en un bucle infinito que se quedará en escucha mediante la instrucción:
"`ewsockfd = accept(sockfd,(structsockaddr *) &cli_addr,&clilen)`"

Cuando recibe algo lo procesa con **`Socket::ParseCmd`** y lo guarda dentro de la FIFO.

4.6 INTERFAZ GRÁFICA DE USUARIO - GUI

Esta clase se encarga de la generación de los mapas y distintos elementos que aparecerán por pantalla.

Ilustración 32- Clase GUI

Para ello se utilizarán texturas que se colocaran fijas en la pantalla y de fondo.

Es importante calcular las coordenadas y la diferencia entre ambas cámaras virtuales, ya que eso nos dará la sensación de posicionamiento en el espacio.

De la misma manera que se hizo al crear la textura de las cámaras, se creara aquí también un plano que estará en el infinito.

La diferencia es que solo se creará una textura, pero dos materiales y dos planos.

Ambos utilizaran la misma imagen (Un recorte de un plano de Santander) pero se encontrarán en posiciones diferentes en cada vista.

Según la posición que se de los planos, nos dará la sensación de que están más cerca del usuario o más lejos. En esta primera versión se ha implementado una interfaz sencilla, pero en versiones posteriores sería posible hacer planos dinámicos que diesen el posicionamiento por GPS. También se podrá implementar diferentes cuadros con información y distintos overlays.

5 EVALUACIÓN DEL SISTEMA

5.1 EVALUACIÓN DEL SISTEMA EN PC

A continuación se va a evaluar la aplicación y para determinar su rendimiento en un PC. La siguiente imagen muestra las imágenes generadas por ARES en el monitor del visor de realidad virtual.

Ilustración 33 - Vista de ARES

5.1.1 Sistema de cómputo

Las características del PC aparecen en la siguiente tabla. Se puede observar que se trata de un portátil de gama media-baja cuya GPU no tiene altas prestaciones.

Equipo	Asus Aspire E 14
GPU	AMD Radeon R4
CPU	AMD A6-6310
RAM	6 GB (4 GB aprovechables)
Sistema Operativo	Ubuntu 14.02
Ogre	1.10
RenderSystem	Open GL
API grafica	Open GL 2.0
Cámaras	Logitech
Oculus Rift	DK1
USB	3.0

5.1.2 Medidas

A continuación se evalúan las prestaciones en dicho sistema.

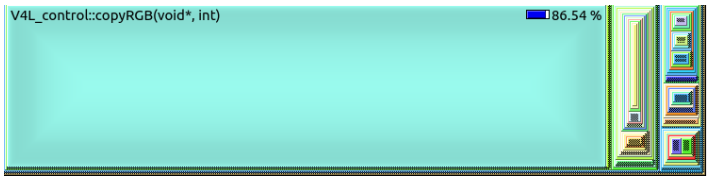
FPSpk	41.7201
FPSmin	38.5847
FPSavg	40.3538
FSAA	8
RAM utilizada	102 MB
CPU load	47%
Tamaño de aplicación	2.0 MB
Tamaño total (App+res+plugins)	108 MB
Resolución	1366x768

Después de realizar un primer análisis vemos que los FPS aunque no llegan a los 60 recomendados quedan por encima de un límite asumible. El sobre muestreo para anti-aliasing (FSAA) es de 8.La RAM usada por la aplicación es de 102 MB que se mantiene estable, por lo que se pueden descartar fugas de memoria.

5.1.3 Análisis de prestaciones

Se quiere mejorar los FPS para poder alcanzar los 60. Para ello necesitaremos analizar con profundidad la ejecución de la aplicación.

Para analizar el rendimiento de ARES, hemos utilizado callgrid, una herramienta de depuración que nos da información sobre las llamadas y el tiempo que consume la aplicación en cada una de ellas.



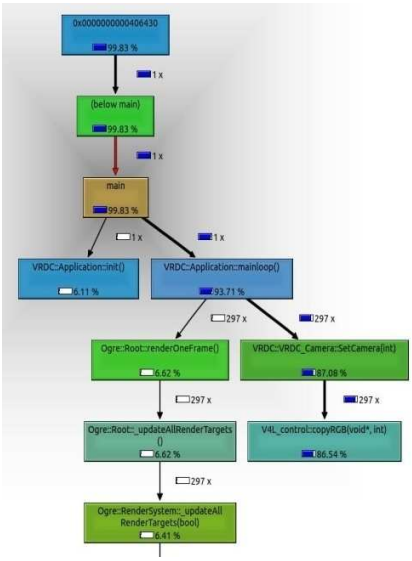
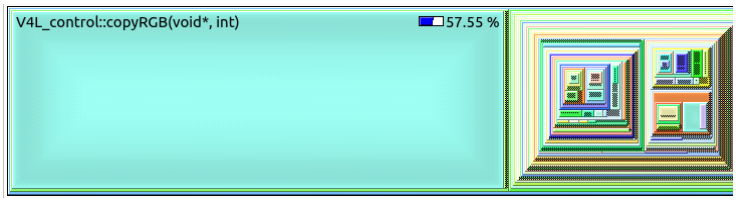


Ilustración 34 - Parte del árbol de llamadas obtenido por callgrid

Vemos que la aplicación permanece un 86.54% del tiempo en el método V4L_control::copyRGB, por lo que es necesaria una optimización del algoritmo, ya que parece la parte critica del sistema.

De todas formas, es necesario tener en cuenta que la escena de la aplicación contiene pocos elementos por lo que Ogre3d tiene baja carga de trabajo.

Para ver cómo afecta el número de elementos, se ha creado una escena con 8000 elementos, y se han analizado sus prestaciones.



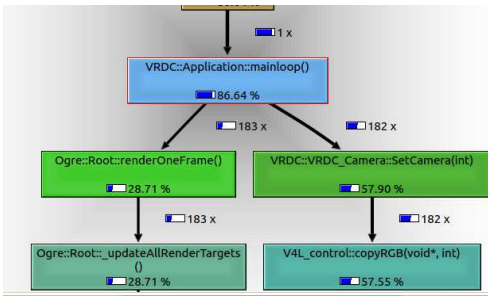


Ilustración 35 - Árbol de llamadas obtenido en la segunda prueba

Vemos que tal como se había presupuesto, la carga se reparte algo más entre el render y el algoritmo de conversión del color de la imagen de la cámara. No obstante la carga del convertidor de color sigue siendo muy significativa, por lo que realmente es necesario optimizar ese algoritmo.

5.1.4 Optimización del algoritmo para aumentar el número de imágenes por segundo

El algoritmo que se debe optimizar es el siguiente

Tal como se explico anteriormente, se trata de un bucle que lee valores de luminancia y crominancia y lo transforma a valores RGB.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1,14 \\ 1 & -0,396 & -0,581 \\ 1 & 2,029 & 0 \end{bmatrix} \begin{bmatrix} Y' \\ U \\ V \end{bmatrix}$$

R = Y + 1.14*V;

G = Y - 0.396*U- 0.58*V;

B = Y + 2.029*U

Debido a que las operaciones con flotantes suelen ser costosas computacionalmente y que algunos sistemas embebidos no cuentan con un co-procesador para operaciones con números flotantes, se ha decidido cambiar esta operación por una alternativa aproximada utilizando enteros.

```
R = (( 298 * Y           + 409 * V + 128) >> 8);
G = (( 298 * Y - 100 * U - 208 * V + 128) >> 8);
B = (( 298 * Y + 516 * U           + 128) >> 8);
```

A continuación se comparan las medidas obtenidas antes y después de este cambio

	FPSavg	FPSmax	FPSmin
Sin optimización	40.7352	41.9512	38.8061
Con optimización	46.7984	49.1159	31.8408

Se ha conseguido un aumento de velocidad de x1.15

5.1.5 Conclusiones del análisis del algoritmo en PC

Optimizando este algoritmo es posible aumentar el framerate de la aplicación y por tanto mejorar notablemente el rendimiento, permitiendo gestionar escenas con muchos elementos virtuales sin que el programa vaya excesivamente lento.

5.2 EVALUACIÓN DEL SISTEMA EN LA PLATAFORMA EMBEBIDA ODROID

A continuación se portará la aplicación a la plataforma embebida Odroid XU3. Para una correcta portabilidad será necesario que todas las librerías necesarias puedan ser portadas y funcionen adecuadamente, verificando todas las dependencias entre ellas.

5.2.1 Sistema

Placa	Odroid XU3
SOC	Exynox 5422
GPU	Mali T628 MP6
RAM	2 GB LPDDR3 RAM at 933MHz
Sistema Operativo	Ubuntu 15.02
Ogre	1.10
RenderSystem	GLES2/ Open GL
API grafica	Open GL ES 3.0
Cámaras	Logitech
Oculus Rift	DK1
USB	3.0

Hay que tener muy en cuenta que la GPU Mali solo soporta OpenGL ES 3.0/2.0/1.1, que es una alternativa a OpenGL especialmente pensada para plataformas móviles.

No obstante, es posible que la aplicación utilice OpenGL, pero en este caso el motor no podrá utilizar la GPU por lo que no conseguirá aceleración hardware y su rendimiento será muy bajo.

5.2.2 Medidas

	GLES2	OpenGL
FPSpk	42.3034	4.1045
FPSmin	23.3213	0.0164
FPSavg	31.0232	2.4823
FSAA	8	8
RAM utilizada	150MB	210MB
CPU load	67%	89%
Tamaño de aplicación	2.4MB	
Tamaño total (App+res+plugins)	108MB	
Resolución	1366x768	1366x768

Comparando resultados entre ambos RenderSystem se puede ver claramente el impacto que supone no utilizar aceleración hardware.

Cuando se usa de RenderSystem GLES2 se observa un rendimiento un poco menor que en el PC utilizado para este trabajo, pero perfectamente asumible. En el caso de utilizar OpenGL el rendimiento cae, y no es en absoluto asumible.

Aunque la elección parece clara, cuando se ha portado la aplicación a Odroid se han encontrado diversos problemas que serán comentados más tarde.

5.2.3 Portabilidad del código del PC a plataforma embebida

Para portar la aplicación es necesario que todas las dependencias con y entre librerías sean cumplidas, por lo que será necesario portar todas las librerías de las que dependa. En muchos casos, es posible encontrar la librería con la versión requerida en el repositorio correspondiente. En otros casos, como libOVR, las librerías deben ser compiladas.

5.2.3.1 Bug en el RenderSystem GLES2

Debido a un bug que no ha podido ser corregido ni justificado, cuando se utiliza el RenderSystem GLES2 junto a la aplicación, esta no muestra ningún elemento en pantalla y solo puede verse el color de fondo.

No ocurre lo mismo cuando se utiliza el RenderSystem Open GL. Como la GPU Mali solo soporta Open GL ES, la implementación de Open GL se ejecuta directamente sobre la CPU, por lo que no hay aceleración hardware de ningún tipo y la aplicación es excesivamente lenta.

5.2.3.2 Falta de soporte y de dependencias para el plugin compositor

El plugin compositor requiere unas librerías no disponibles para la GPU Mali, por lo que no es posible usar este plugin que se encarga de ejecutar el script compositor para Oculus que genera la distorsión en cada una de las vistas. Este problema no es crítico ya que puede proveerse esta funcionalidad mediante código específico.

5.2.3.3 Depuradores inestables

Para tratar de solventar los problemas de rendering, se ha tratado de sacar trazas y árboles de llamada de la misma manera que el PC. No obstante los depuradores embebidos se muestran muy inestables y nunca llegan a terminar ninguna ejecución, a pesar de estar días ejecutándose, por lo que resulta casi imposible obtener alguna idea de lo que puede estar ocasionando el fallo.

6 IMPLEMENTACIÓN DEL SISTEMA UTILIZANDO UNITY

Debido a que no ha sido posible implementar correctamente la aplicación en el sistema elegido, se han explorado otras alternativas.

6.1 UNITY 5, VR Y ANDROID

Una posible solución podría ser utilizar un entorno diferente de Ogre3D o/y un caso de realidad virtual nuevo. Por ejemplo:

- Unity 5.x: A partir de Unity 5, la versión gratuita ofrece todas las características necesarias para crear nuestra aplicación. Además ofrece un soporte nativo a Realidad Virtual y es posible compilar directamente para el sistema Android.
- Oculus Rift DK2 o Gear VR: Las Oculus Rift DK1 están obsoletas, por lo que las últimas versiones del SDK o de los motores gráficos con soporte nativo no pueden no proporcionar soporte para DK1 y solucionar problemas.
- Plataforma con soporte VR: No todas las plataformas móviles dan soporte a VR. Por ello hay que elegir una que lo proporcione. Por ejemplo Samsung Note 4
- Sistema Android 4.4 o superior: Gear VR requiere esa versión o superior.

Para crear una aplicación Android VR primero se deberá activar el soporte en la configuración de jugador

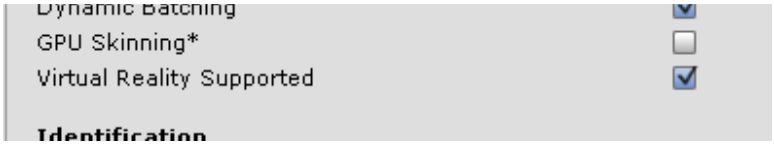


Ilustración 36 - Opción de soporte para realidad virtual en Unity

Para poder compilar para un sistema Android, es necesario el Android SDK, que podemos obtener descargando el Android studio. Además, para ejecutar directamente sobre el sistema Android, es necesario conectar nuestro equipo por USB al dispositivo, y activar el modo de depuración USB en él.

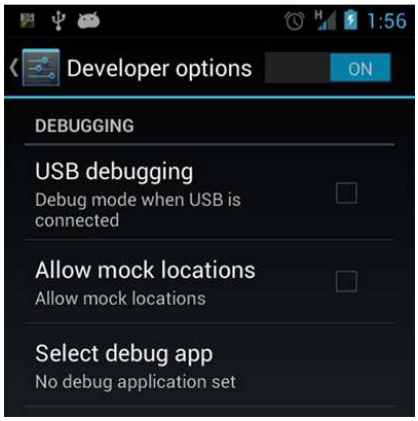


Ilustración 37 - Opción de depuración por USB en un dispositivo Android

En Unity, en build settings, se seleccionará el sistema Android

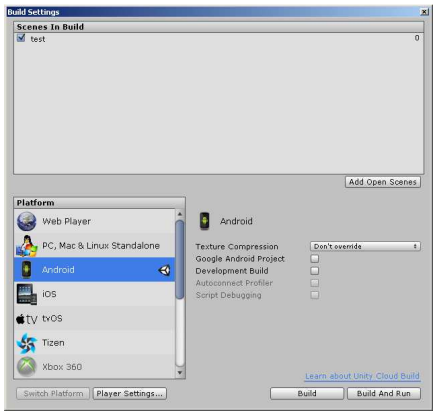


Ilustración 38 - Ventana de Build Settings de Unity

Y se ejecutará un “build & ron”. Si el dispositivo es compatible, después de compilar se ejecutará la aplicación en él. Podemos obtener logs del dispositivo mediante el comando de consola “adb logcat”.

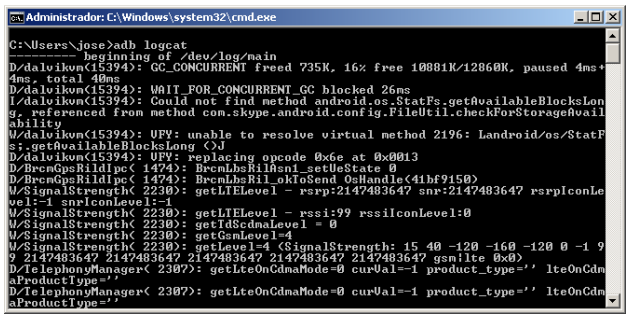


Ilustración 39 - Resultado de logcat

7 CONCLUSIONES Y TRABAJO FUTURO

En este proyecto se ha desarrollado una aplicación de realidad aumentada que puede ser utilizada como punto de partida o ejemplo para generar nuevos sistemas que usen esta tecnología.

El sistema utiliza visor o casco de realidad virtual (Oculus Rift DK1) para mostrar al usuario la realidad mixta (real combinada con virtual). Dichas gafas también proporcionan información sobre la posición y movimiento de la cabeza del usuario del sistema.

Además, el sistema integra un par de cámaras que capturan el entorno real que rodea al usuario.

En el proyecto se han desarrollado una aplicación en C++ que utiliza el motor gráfico Ogre3D para generar el entorno virtual e insertar en el mismo las imágenes capturadas por las cámaras. Para poder capturar la imagen de las cámaras con baja latencia ha sido necesario utilizar el interfaz V4L (Video For Linux). La aplicación genera las imágenes de la realidad aumentada y las transfiere al casco de realidad virtaul mediante la librería libOVR. La misma librería es utilizada por la aplicación para determinar la posición del usuario en el mundo virtual.

La aplicación desarrollada gestiona el mundo virtual teniendo en cuenta los datos de los sensores del visor Oculus Rift, lo que aumenta la sensación de inmersión en mundo aumentado.

La aplicación puede ser controlada de forma remota, no precisando utilizar el ratón y teclado. También puede acceder a recursos remotos.

Todo el sistema ha sido desarrollado en C++ y utilizando librerías de código abierto, lo que facilita la portabilidad y adaptación del código.

La aplicación desarrollada tiene unas prestaciones adecuadas (genera alrededor de 30fps) tanto cuando se ejecuta en PC como en la plataforma embebida, aunque en el último caso problemas con las librerías parece que están afectando al resultado final, impidiendo que se genere correctamente la imagen.

Como trabajo futuro se propone actualizar las últimas versiones de cascos de realidad virtual (al menos Oculus Rift DK2), utilizar una plataforma embebida que tenga un buen soporte de las librerías necesarias para VR, identificar un motor gráfico que haga un uso eficiente de los recursos hardware de la plataforma (máximo uso de la aceleración hardware) y optimizar la implementación del conversor de color.

8 ANEXOS

8.1 ACCESO AL REPOSITORIO CON EL CÓDIGO

http://wsvn.bsro.es/listing.php?repname=ARES

Usuario: TFG

Contraseña: tfg_2016

8.2 FICHEROS CON EL CÓDIGO FUENTE

- core.cpp
- Application.cpp
- Application.h
- OgreOculus.cpp
- OgreOculus.h
- Euler.cpp
- Euler.h
- GUI.cpp
- GUI.h
- Socket.cpp
- Socket.h
- V4L_control.cpp
- V4L_control.h
- VRDC_Camera.cpp
- V4DC_Camera.h
- conf.h

8.3 ARCHIVOS DE CONFIGURACIÓN

- camera.cfg
- config.cfg
- plugins.cfg
- resource.cfg
- server.cfg

9 BIBLIOGRAFÍA Y DOCUMENTACIÓN

[1] Sitio oficial de Ogre - http://www.ogre3d.org

[2] Documentación de Ogre - http://www.ogre3d.org/docs/manual/

[3] Sitio oficial de Oculus - https://www.oculus.com

[4] Documentación de Oculus Rift - https://developer.oculus.com/documentation/pcsdk/latest/concepts/book-gsg/

[5] Documentación de Odroid - http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127

[6] Unity - http://docs.unity3d.com/es/current/Manual/index.html

Ver también:

- Pro OGRE 3D Programming : Gregory Junker
- Computer Vision Principes and Practice