



UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática



Experimentos de Creación de Contenidos Gráficos para Realidad Virtual basadas en OpenGL

Trabajo de Fin de Grado
Escuela de Ingeniería Informática

Autor: Alberto Casado Garfía

Tutores: Juan Ángel Méndez Rodríguez

Modesto Fernando Castrillón Santana

Las Palmas de Gran Canaria Noviembre de 2017

SOLICITUD DE DEFENSA DE TRABAJO DE FIN DE TÍTULO

D/D^a Alberto Casado Garfia, autor del Trabajo de Fin de Título experimentos de creación de contenidos gráficos para Realidad Virtual basados en OpenGL, correspondiente a la titulación Grado en Ingeniería informática.

SOLICITA

que se inicie el procedimiento de defensa del mismo, para lo que se adjunta la documentación requerida.

Asimismo, con respecto al registro de la propiedad intelectual/industrial del TFT, declara que:

☐ Se ha iniciado o hay intención de iniciarlo (defensa no pública).

☒ No está previsto.

Y para que así conste firma la presente.

Las Palmas de Gran Canaria, a 19 de Diciembre de 2017.

El estudiante

Fdo.: Alberto Casado Garfia

A rellenar y firmar **obligatoriamente** por el/los tutor/es

En relación a la presente solicitud, se informa:

☐ Positivamente

☐ Negativamente

(la justificación en caso de informe negativo deberá incluirse en el TFT05)

Fdo.: _____

DIRECTOR DE LA ESCUELA DE INGENIERÍA INFORMÁTICA

CONTENIDO

1. Introducción	1
2. Visión estereoscópica y estereoscopía	3
2.1 Visión estereoscópica	3
2.2 Estereoscopía y cascos de realidad virtual	5
3. Estado del arte en realidad virtual y visión 3D	13
3.1 Cascos de realidad virtual en la actualidad	14
4. Desarrollo de la aplicación Sensor Transmitter	19
4.1 Análisis de la aplicación	19
4.2 Diseño de la aplicación	20
4.3 Implementación de la aplicación	22
4.3.1 Actividad e interfaz de usuario	22
4.3.2 Sensores	25
4.3.2 Transmisores	29
4.4 Posibles mejoras	31
5 OpenGL	33
5.1 Librerías asociadas a OpenGL	37
5.1.1 GLUT	37
5.1.2 GLM	38
5.1.2 STB	38
5.1.2 Assimp	38
6 Desarrollo del demostrador en OpenGL	39
6.1 Cascos de realidad virtual utilizados	41
6.2 Partes del programa	42
6.2.1 Fullscreen	42
6.2.2 NetReceiver	44
6.2.3 StereoVision	46
6.2.4 Cámaras	53
6.2.5 Carga de modelos	59
6.2.5 Animation	60
6.2.6 Demostrador	61
7. Pruebas y resultados	63

8. Conclusiones	67
9. Anexo	69
9.1 Instalación y configuración del entorno	69
9.2 Tutorial de uso	73
9. Bibliografía	83
Imágenes y otros recursos	84

1. INTRODUCCIÓN

La realidad virtual es un sector en pleno desarrollo con aplicaciones en múltiples campos y en especial en la industria del entretenimiento. Desde 2010 la inversión en este sector ha superado los 3.500 millones de euros y cada vez es mayor el número de startups dedicadas al sector, ya que es un sector propicio para la creación de nuevos proyectos. Este crecimiento ha provocado que en los últimos años empresas punteras en tecnología como Microsoft o Sony hayan presentado proyectos relacionado con esta tecnología y similares como la realidad mixta o la realidad aumentada.

Para experimentar la realidad virtual es necesario tener unos cascos de realidad virtual. En el mercado ya existen multitud de dispositivos de diferentes características y precios cada vez más asequibles para los usuarios, con los que se puede acceder a experiencias de realidad virtual como videojuegos, simuladores o retransmisión de eventos deportivos entre otros. La creación de videojuegos que utilizan realidad virtual es un ámbito emergente dentro del sector que cuenta con cada vez un mayor número de títulos disponibles.

En el proceso de implementación de estas experiencias virtuales se deben de utilizar técnicas de estereoscopía para poder crear la realidad virtual. El mal uso de estas técnicas puede conllevar desde el malestar de usuario a la pérdida total de la inmersión, por lo que es de vital importancia conocer los principios en los que se basa estas técnicas y su aplicación en las experiencias de realidad virtual.

En este trabajo de fin de título se trata este problema, desde la explicación de las técnicas de estereoscopía usadas a su posterior implementación en un demostrador. En la ilustración 1 se muestra el esquema de las distintas partes del trabajo y cómo interactúan entre ellas. El demostrador es el encargado de generar la realidad virtual utilizando OpenGL y será ejecutado en un ordenador. La imagen se visualiza a través de unos cascos de realidad virtual usando la visión estereoscópica. Los cascos de realidad virtual utilizados usan la pantalla del teléfono móvil como pantalla interna.

Para transmitir la imagen generada por el demostrador al teléfono móvil se utiliza el software *SpaceDesk*, un software que permite expandir el escritorio de Windows a otras pantallas en la red local. El demostrador dibuja la visión estereoscópica en una ventana a pantalla completa en la

pantalla mostrada en el móvil, de manera que el software retransmite esta pantalla directamente al móvil.

Con el objetivo de mejorar la experiencia de realidad virtual se creó una aplicación que capturase los eventos de los sensores del móvil y los transmitiese, de forma que fuesen recibidos por el programa. Esta aplicación se le acabó dando el nombre de *Sensor Transmitter* y permite obtener la orientación del móvil y usarla en la demostración.

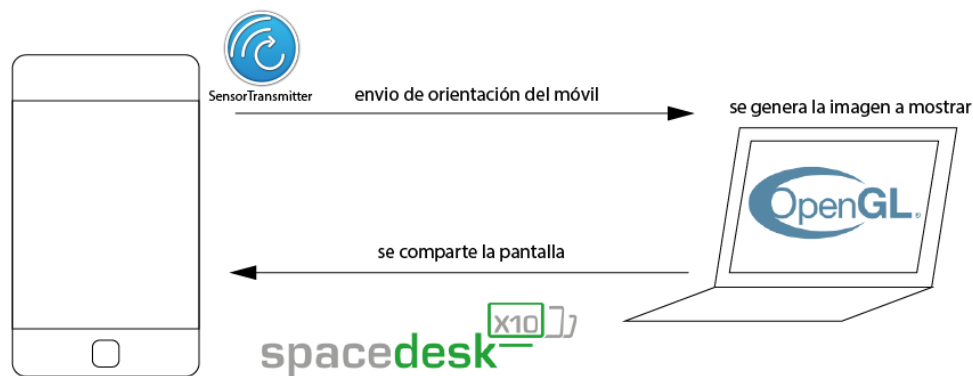


Ilustración 1 Esquema de funcionamiento del demostrador

Durante el desarrollo del TFG recopilé información sobre los métodos utilizados en realidad virtual, cuya teoría se explica en el segundo capítulo, e implementarlos para ponerlos a prueba, como se describe en el sexto capítulo junto al desarrollo del demostrador y en el séptimo junto con las observaciones.

La idea inicial del trabajo de fin de grado surgió a partir de una propuesta de una práctica de la asignatura de Creación de Interfaces de Usuario del grado de Ingeniería Informática de la Universidad de Las Palmas de Gran Canaria. En esta práctica se visualizaría un objeto 3D usando los cascos de realidad virtual, utilizando *SpaceDesk* para compartir la pantalla. Tras comentar con Juan Méndez, el profesor de la asignatura, la práctica, tomé la decisión de orientar el TFG en esta línea, partiendo de la idea de la práctica e incluyendo todos los aspectos comentados anteriormente.

2. VISIÓN ESTEREOSCÓPICA Y ESTEREOSCOPIA

2.1 Visión estereoscópica

La visión estereoscópica es una facultad que posee el ser humano que le permite ver en *3D* aquellos objetos que contempla con su visión binocular. Cada ojo capta con una perspectiva distinta del entorno una imagen que posteriormente el cerebro integra en una sola. Los objetos distantes son observados prácticamente igual por ambos ojos mientras que en los objetos cercanos cada ojo observa detalles diferentes de cada lado.

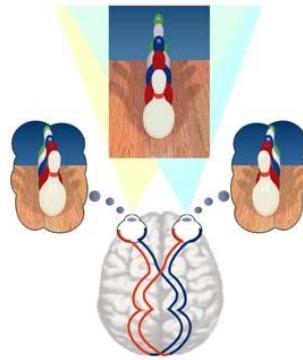


Ilustración 2 Visión estereoscópica [19]

Esta información es usada por el cerebro para calcular la distancia a la que se encuentra el objeto y crear la sensación de profundidad. Es necesario para la visión estereoscópica que al menos 2 ojos enfoquen hacia la misma dirección. Por esa razón los animales con visión lateral como los caballos carecen o tienen una menor visión estereoscópica, pero a cambio consiguen una visión periférica mayor que puede salvarlos de amenazas de depredadores. También es el motivo por el que nuestros ojos se muevan automáticamente en la misma dirección. Para procesar la información espacial más rápidamente nuestra mente usa la información observada previamente y se basa en aspectos como las sombras y la perspectiva de los objetos para determinar de forma más precisa la profundidad de estos.

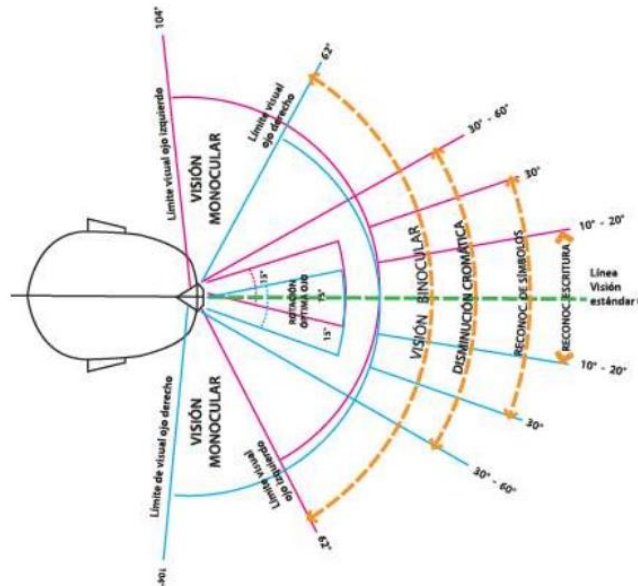


Ilustración 3 Campo visual[20]

El ser humano tiene una amplitud de campo visual de aproximadamente 180°, pero solo la región abarcada por ambos ojos puede percibir la profundidad con visión estereoscópica. En el punto donde enfocan los ojos se concentra la agudeza visual y usamos para reconocer detalles como por ejemplo en la lectura. En la ilustración 3 se pueden observar los diferentes ángulos que abarcan cada parte del campo visual, aunque los ángulos pueden variar según cada persona.

La distancia entre los dos puntos de vista, es decir la distancia interpupilar, es generalmente de 65mm. Para enfocar un objeto que está a una cierta distancia el ojo ajusta el cristalino que actúa como una lente para tener una visión clara del objetivo. La distancia mínima a la que puede enfocar el ojo humano es de 250 mm. A menor distancia se comienza a ver borroso el objeto y a menos 150 mm desaparece la sensación de visión estereoscópica y el objeto se ve doble. El ojo puede enfocar hacia el infinito, relajando el cristalino y convergiendo la mirada en el horizonte de manera que los rayos visuales sean paralelos, permitiendo ver claramente objetos muy lejanos.

2.2 Estereoscopia y cascos de realidad virtual

La estereoscopia es un conjunto de técnicas capaces de recoger información visual tridimensional y/o crear la ilusión de profundidad. Para crear una imagen virtual con una ilusión de profundidad cada ojo debe recibir una imagen diferente que el cerebro pueda combinar en una imagen tridimensional coherente gracias a la visión estereoscópica. Existen principalmente dos técnicas estereoscópicas, los cascos de realidad virtual donde una pantalla proyecta sobre cada ojo, y las pantallas 3D que proyectan una imagen que luego es filtrada para obtener una imagen diferente en cada ojo, normalmente usando sistemas como las gafas 3D. El primer método es el usado en la industria de realidad virtual mientras el segundo es utilizado en la proyección de películas 3D en cines.

Los cascos de realidad virtual captan los movimientos de la cabeza que son procesados por el software para simular el mismo movimiento dentro de la realidad virtual. Algunos cascos traen consigo mandos con los que interactuar dentro del mundo virtual con el fin aumentar la inmersión.

Cuando la experiencia de visualizar estas imágenes sea más parecida a unas observadas realmente mayor será la inmersión. Sin embargo, el mal uso de ciertos aspectos gráficos puede afectar negativamente a la experiencia y causar mal del simulador o *simulator sickness*. Es un conjunto síntomas como fatiga visual, desorientación, náuseas y mareos que es sufrido tras largos periodos de utilización de simuladores de realidad virtual mal realizados. Tanto el hardware como el software utilizado pueden afectar negativamente experiencia, por lo que un hardware defectuoso o de baja calidad puede causar este malestar al igual que un mal software que cree una mala imagen.

Uno de los aspectos que más empeoran la experiencia en los cascos de realidad virtual es la latencia o fotogramas por segundo en los que se reproduce la simulación. A partir de los 10 fotogramas por segundo el cerebro capta el cambio de las imágenes como un movimiento continuo y en los videojuegos el rango normal de fotogramas varía entre los 30 y 60 fotogramas por segundo. En los cascos de realidad virtual la latencia media es de 90 fotogramas por segundo ya que latencias menores pueden causar fatiga visual. Para ello es necesario aparte de una pantalla con una alta tasa de refresco, una gran potencia gráfica, pues la escena que contempla el usuario debe de ser dibujada 2 veces, una para cada ojo y en el tiempo entre fotogramas. El programa de realidad virtual también debe de proporcionar una latencia estable ya que también afectan negativamente a la experiencia.

En los cascos de realidad virtual es necesario que la pantalla interna se encuentre a una distancia que pueda enfocar el ojo humano o que se consiga ese efecto por medio de lentes. Otro aspecto para tener en cuenta es el campo visual que cubre la pantalla interna, cuanto mayor sea, más inmersivo será la experiencia pues el usuario verá menos el borde de la pantalla. Junto con la resolución de la pantalla, son un gran condicionante en la potencia gráfica que será necesaria para generar la realidad virtual. Un gran campo visual también puede provocar malestar debido a que la visión periférica es más sensible a la percepción del movimiento y los movimientos observados pueden entrar en conflicto con los movimientos que el usuario siente con otros sentidos. Es preferible que el usuario pueda ajustar todos estos parámetros para acomodarlos a sus necesidades.

El programa que genere la realidad virtual es el encargado de dibujar correctamente las imágenes que se proyectarán en las pantallas de los cascos de realidad virtual. Para ello debe recoger la información de los sensores del casco como la orientación de la cabeza y simularlos en la RV, para luego dibujar las escenas desde el punto de vista del usuario en la RV. Al proceso de generar una imagen o video a partir de un modelo 3D se le llama renderización.

En videojuegos y simulaciones la escena se suele dibujar usando perspectiva para conseguir unos gráficos más realistas. Los objetos más alejados se verán más pequeños que los cercanos, al igual que en la realidad. El espacio que es observado usando la perspectiva tiene forma una figura geométrica denominada *frustum*. Es una porción de una pirámide comprendida entre 2 planos, como se puede observar en la imagen 4.

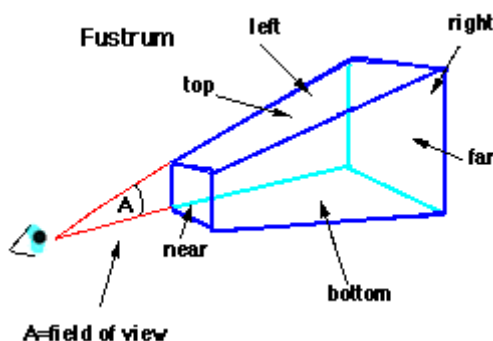


Ilustración 4 Frustum[21]

Se suele llamar en programación gráfica cámara al objeto que se usa para determinar que se va a renderizar, es decir que verá el usuario. Las cámaras en programación gráfica imitan las cámaras de la vida real, ambas capturan la realidad desde una cierta posición, enfocando en una cierta dirección y abarcando un campo de visión. Cambiando estos parámetros se modificará el frustum del espacio observado por la cámara.

Mientras que en las simulaciones en *2D* solo necesitamos una cámara, lo que se suele denominar mono, en la visión en estéreo necesitamos dos cámaras, cada una posicionada en el punto de vista de cada ojo dentro de la realidad virtual. La separación entre ambas cámaras no debe ser ni muy pequeña, pues el usuario no distinguirá la diferencia entre los puntos de vista y no apreciará la profundidad, ni muy grande que pueda causar malestar e incluso que el usuario no pueda integrar las dos imágenes y se pierda toda la inmersión. En general la separación entre las cámaras debe ser igual a la distancia interpupilar.

Al igual que en mono, en estéreo cada cámara formará un *frustum*. Existen diferentes configuraciones de cámaras que forman diferentes frustum, aunque las 2 principales son la *Toe-in* y la *Off-Axis*.

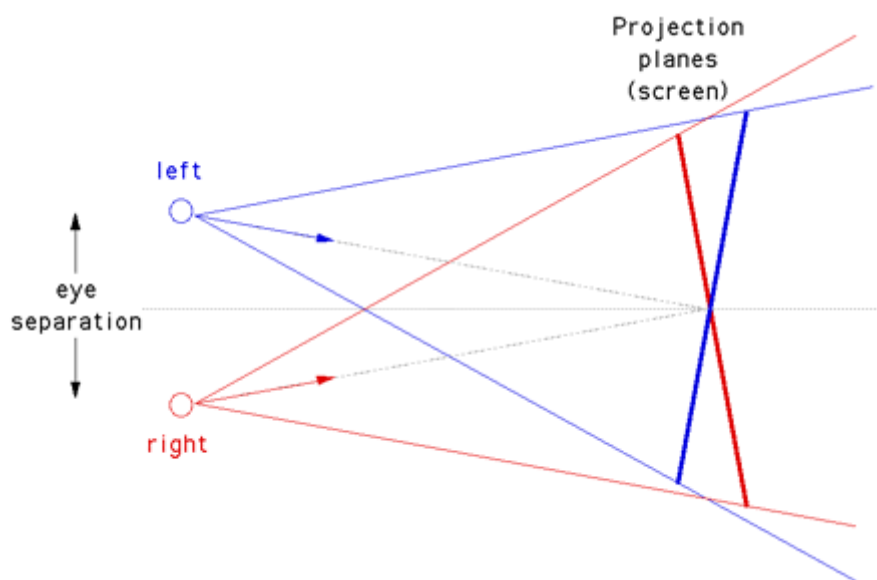


Ilustración 5 Frustums en Toe-in[22]

En la configuración *Toe-in*, mostrada en la ilustración 5, cada cámara apunta hacia un punto focal. Es una configuración sencilla pero que resulta en un efecto tridimensional que puede causar fatiga. Se puede reducir la fatiga reduciendo la separación entre las cámaras y centrando la atención en el centro de la pantalla, donde la distorsión producida es menor.

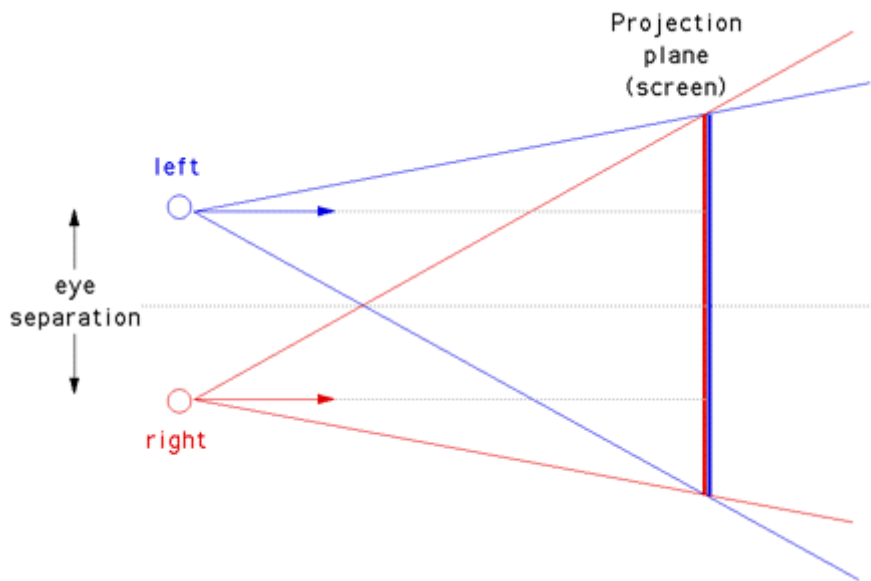


Ilustración 6 Frustums en Off-Axis[22]

El método *Off-Axis* o *skewed frustum*, mostrado en la ilustración 6, es el utilizado actualmente en los cascos de realidad virtual ya que genera un correcto efecto 3D. Sin embargo, usa frustum asimétricos, por lo que no se puede implementar en algunos frameworks de renderización antiguos.

En ambos casos, aquellos objetos que estén situados antes del punto focal o el plano de proyección para el usuario tendrán la apariencia de salirse de la pantalla, mientras que aquellos que están detrás para el usuario parecerán que están situados más alejados.

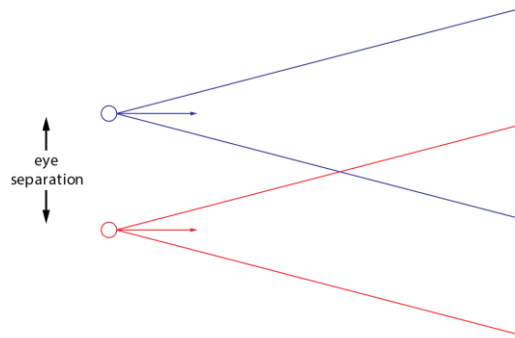


Ilustración 7 Offset frustums

Otra opción es utilizar el método *offset frustums* de la ilustración 7, en el cual se usan frustums simétricos apuntados hacia donde el usuario dirija la vista. Es semejante a poner en el método *off-axis* el plano de proyección en el infinito.

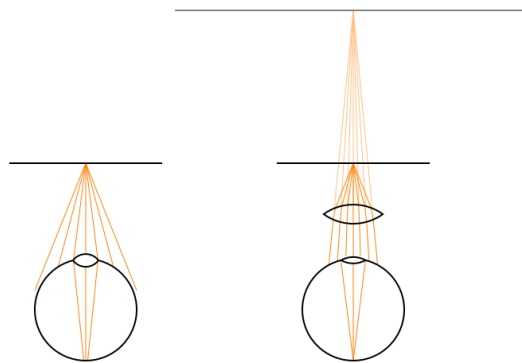


Ilustración 8 Ejemplo de la razón de uso de lentes en los cascos de realidad virtual[23]

La gran mayoría de cascos de realidad virtual utilizan lentes para proyectar la pantalla interna más lejos de donde realmente está. Sin ellas el usuario debería forzar la vista para observar la pantalla debido a su cercanía, e incluso podría darse el caso de que no lograra enfocarla, como se puede observar en la parte izquierda de la ilustración 8. En la parte derecha se puede observar el efecto

creado por la lente, las lentes refractan la luz produciendo la sensación de que la pantalla está más lejos y reduciendo la amplitud del campo visual, de manera que la pantalla parece ser más grande.

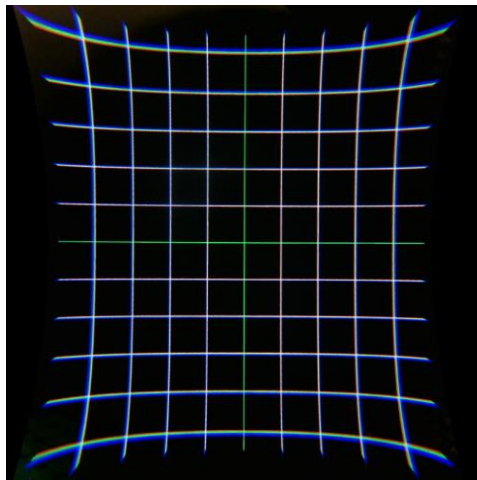


Ilustración 9 Distorsión generada por las lentes en los cascos Oculus Rift DK2[24]

La mayoría de las lentes generan una distorsión de acerico o de cojín y además pueden descomponer en menor medida la luz blanca como en un prisma, provocando el fenómeno óptico de distorsión cromática. En la imagen 9 puede observarse como la imagen de una rejilla se distorsiona por los efectos ópticos de las lentes. Estos efectos pueden ser contrarrestados distorsionando la imagen que va a ser proyectada en la pantalla. Estas distorsiones no pueden realizarse modificando los frustums de la cámara, por lo que se debe recurrir a otras técnicas.

Existen ciertos aspectos para tener en cuenta al realizar una experiencia de realidad virtual que en otras simulaciones en 2D son triviales. Los objetos planos, como partículas o interfaces deben estar paralelos al plano de proyección para que el usuario los perciba de forma cómoda. Las interfaces deben adaptarse a la realidad virtual integrándose con el entorno. El uso de mandos puede ayudar a integrar la interfaz y a hacer las acciones más naturales. También puede usarse el punto donde el usuario dirige la vista como cursor. En caso de que no pueda integrarse la interfaz en el mundo se recomienda alejar la interfaz a unos 75 cm y no fijarla a la cámara si ocupa demasiado espacio.

Las luces, sombras, texturas y sonidos en estéreo mejoran la inmersión. Los avatares son útiles en videojuegos como los *shooters*, donde el usuario puede ver sus manos, pero pueden reducir la inmersión al no reconocer su cuerpo. Se deben evitar las aceleraciones anteponiendo los cambios

de velocidad inmediatos a los progresivos, ya que las aceleraciones entran en conflicto con lo que siente el sentido del equilibrio del oído, provocando mareo. Los efectos de cámara como el zoom o el balanceo al caminar también pueden ser incómodos.

Ciertos cascos tienen un *SDK* o *API* dedicado a simplificar todos los aspectos descritos en el capítulo y una documentación con las mejores prácticas de desarrollo para ese hardware en específico. Es recomendable buscar siempre esta información para obtener los mejores resultados.

3. ESTADO DEL ARTE EN REALIDAD VIRTUAL Y VISIÓN 3D

La realidad virtual es una representación de escenas que produce en el espectador la sensación de su existencia real. Son sistemas informáticos que generan mundos artificiales que el usuario puede recorrer e interactuar. El término fue propuesto por Jaron Lanier de American VPL Researchs Inc en 1989. La realidad virtual es usada en el terreno del entretenimiento y del mundo de los videojuegos principalmente, aunque se ha extendido a otros campos como la medicina, la arqueología, la creación artística y el entrenamiento por medio de simulaciones. Una técnica usada en realidad virtual para aumentar la inmersión es la visión 3D. Existen múltiples técnicas que son capaces de crear una imagen tridimensional que han sido perfeccionadas a lo largo de los años desde que Sir Charles Wheatstone inventase la estereoscopia en la década de 1840.

Los anaglifos es una de las primeras técnicas usadas en el cine y programas de realidad virtual como los videojuegos. Fueron patentados por Louis Ducos du Hauron en el 1861 y utilizan los principios de la estereoscopia para crear una sensación de profundidad. En la pantalla se superponen dos imágenes con diferentes colores, normalmente azul y rojo, que luego serán filtrados por gafas anaglifo con filtros semitransparentes de estos colores. Los colores que sean iguales al filtro no podrán ser percibidos por el ojo, por lo que se crean 2 imágenes diferentes en cada ojo. Las gafas anaglifo pueden hacerse con cartón y papel celofán de 2 colores, por lo que siguen siendo usadas debido a su sencillez y bajos costes, sin embargo, la baja calidad del efecto tridimensional reduce la inmersión y hace que cada vez sean menos usadas.



Ilustración 10 Gafas anaglifo[25]

Las gafas anaglifo han sido sustituidas por gafas con filtros polarizados. Esta técnica se basa en proyectar en la pantalla 2 imágenes con luces de distinta polarización, que luego cada cristal de unas

gafas filtra para que cada ojo reciba la imagen correcta. En esta técnica se basan las tecnologías usadas en el cine como Real-D 3D, IMAX 3D y DOBLY 3D y algunos televisores y monitores 3D. Para esta técnica es necesario proyectores o pantallas que sean capaces de producir luz con diferente polarización y la realidad sigue estando limitada al marco de la pantalla.

En el mundo de los videojuegos se han creado múltiples consolas y periféricos enfocados a la realidad virtual desde 1995 siendo uno de los precursores la consola de Nintendo Virtual Boy. Se trataba de unas gafas que utilizaban en su interior un proyector para mostrar 3D monocromático por medio de un efecto estereoscópico. El lanzamiento resultó ser un fracaso, lo que causó que otros proyectos similares como la SEGA VR no saliesen a la luz y las experiencias de realidad virtual solo siguiesen existiendo en máquinas recreativas. Más tarde, en 2006 se desarrolló la consola Wii de Nintendo con el mando Wii Remote y en 2010 el periférico Playstation Move de PlayStation 3 de la compañía Sony, que detectaban el movimiento del mando, permitiendo a los usuarios interactuar de manera más natural con el videojuego. En 2010 Microsoft también desarrolló Kinect para la consola Xbox 360 y PC, un controlador que permite interactuar con la consola captando con una cámara los movimientos del jugador. Estos mandos y periféricos, junto con el apogeo de películas en 3D fueron los primeros pasos hacia la realidad virtual actual.

3.1 Cascos de realidad virtual en la actualidad

Durante los últimos años han llegado al mercado muchos proyectos de cascos de realidad virtual o HMD enfocados a videojuegos y experiencias inmersivas. Esta tendencia comenzó en 2012 cuando se lanzó en la plataforma de crowdfunding Kickstarter el casco de realidad virtual Oculus Rift con el objetivo de recaudar 250.000 dólares para fabricar los primeros prototipos. Al final de la campaña casi había recaudado 2.5 millones de dólares y 2 años más tarde gracias al éxito del proyecto este sería adquirido por Facebook por 2.000 millones de dólares. Oculus Rift cuenta con un mando para interactuar con la realidad virtual que utiliza sensores de posición. Existen varias versiones de los cascos, incluyendo versiones para desarrolladores. Hay un kit de desarrollo de software para ayudar a integrar Oculus Rift en sus juegos y versiones de este kit para los principales motores gráficos. En 2017 se anunció Oculus Rift Go, una nueva versión que no necesita un ordenador para renderizar la realidad virtual.



Ilustración 11 Oculus Rift Go[26]

Ese mismo 2012 Google anunció sus gafas de realidad aumentada Google Glass, un dispositivo que muestra información al usuario en las propias gafas y le permite interactuar con el dispositivo usando comandos de voz. Cuenta con una cámara y otros sensores para obtener información del entorno, procesarla y mostrar datos útiles al usuario sobre ellos. El proyecto en un principio creado para el público general fue reorientado para el uso en empresas.

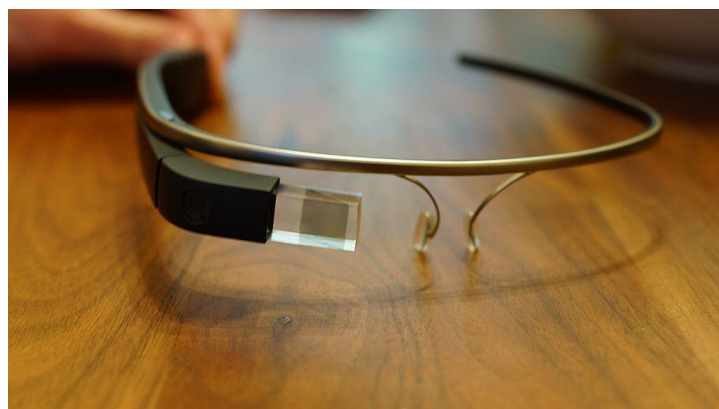


Ilustración 12 Google Glass[27]

Hololens es el nombre de las gafas de realidad mixta desarrollada por Microsoft. La realidad mixta es la combinación de realidad aumentada y realidad virtual. Fue anunciado en una conferencia de

la empresa en enero de 2015 y en la conferencia de videojuegos E3 de ese mismo año. Existen versiones funcionales del casco desde estas fechas. El proyecto llevaba en desarrollo durante 5 años, aunque fuese concebido en 2007 con la plataforma Kinect. El sistema operativo Windows 10 está preparado para llevar su interfaz a las gafas con su experiencia Windows Holographic. Las ventanas del sistema operativo pueden fijarse en posiciones del entorno y el usuario puede interactuar con ellas con sus manos. Al encender las Hololens el casco escaneará el entorno para crear un mapa sobre el que posteriormente proyectar los hologramas. También puede generar una realidad virtual como Oculus Rift, pero su diseño no se enfatiza en ese aspecto por lo que la inmersión en ese tipo de experiencias es menor.



Ilustración 13 Hololens[28]

La empresa Sony anunció en la conferencia de desarrolladores de videojuegos GDC en 2014 su nuevo visor de realidad virtual llamado PlayStation VR, que es compatible con su consola PlayStation 4. Los controles usados son el propio mando de la consola PlayStation 4 Dualshock 4, los mandos PlayStation Move y un nuevo mando con forma de pistola. Desde octubre de 2016 el dispositivo está a la venta en el mercado y cuenta con un gran catálogo de juegos compatibles.



Ilustración 14 PlayStation VR[29]

En 2015 las compañías HTC y Valve anunciaron sus prototipos del casco de realidad virtual HTC Vive. Cuenta con una cámara delantera para detectar cualquier objeto enfrente del casco y mandos para interactuar con la realidad virtual. Está a la venta desde 2016.



Ilustración 15 HTC Vive[30]

Samsung Gear VR es un casco de realidad virtual desarrollado por Samsung con la colaboración de Oculus VR. El casco está diseñado para introducir un móvil y que su pantalla sirva como pantalla interna del casco. El programa que genere la realidad virtual se ejecutará en el móvil por lo que no es necesario el uso de un ordenador. Los movimientos de la cabeza serán captados por los sensores

del teléfono. El casco solo es compatible con móviles de gama alta de la compañía como el Galaxy S6, S7, S8 y algunos modelos del Galaxy Note.

En el mercado existen cascos que utilizan el mismo método que las Samsung Gear VR pero están diseñados para ser usados en más dispositivos móviles. Google creó una versión de este concepto llamado Google CardBoard, el cual se puede fabricar a partir de diseños de internet, cartón y dos lentes. Estos tipos de cascos son los más asequibles que existen en el mercado y son los que se usarán en este TFG.

	OCULUS RIFT	SONY PLAYSTATION VR	HTC VIVE	SAMSUNG GEAR VR
RESOLUCIÓN DE PANTALLA	2160 x 1200	1920 x 1080	2560x1200	2560x1440
LATENCIA	90 fps	120 – 90 fps	90 fps	60 fps
CAMPO DE VISIÓN	110°	100°	110°	96°

Tabla 1 Diferencias técnicas de cascos de realidad virtual

Existen pantallas que pueden reproducir imágenes tridimensionales sin que el usuario tenga que utilizar ningún dispositivo especial usando la técnica de autoestereoscopia. Este método es el usado por la consola Nintendo 3Ds y algunos teléfonos móviles, y que también es utilizado en tarjetas y postales con efecto 3D.

4. DESARROLLO DE LA APLICACIÓN SENSOR TRANSMITTER

Muchas aplicaciones de realidad virtual captan el movimiento de la cabeza para simular el mismo movimiento dentro de la realidad virtual y crear una mayor inmersión. Es una técnica muy usada en videojuegos de realidad virtual o aplicaciones de realidad aumentada.

En el caso del TFG, como el renderizado se realiza en el ordenador y posteriormente se transmite al móvil, no es posible obtener la inclinación de la cabeza desde la aplicación de OpenGL. Pero sí es posible crear una aplicación que transmita los datos de la orientación del móvil a la aplicación de OpenGL. Con esta idea se desarrolló la aplicación *Sensor Transmitter*, una aplicación que puede transmitir los valores de los sensores de un móvil.

4.1 Análisis de la aplicación

Para la creación de la aplicación se eligió el kit de desarrollo de software de Android ya que permite acceder directamente a la *API* de Android y hace más sencillo trabajar con sensores y conexiones de red. Además, se utiliza el lenguaje Java para programar con el *SDK*. En otros frameworks multiplataforma, como React Native, muchas de estas características no están implementadas por defecto y es necesario importar plugins para usarlas.

A pesar de que la aplicación tiene el objetivo de transmitir la información de los sensores, para este proyecto solo será necesario transmitir la orientación del dispositivo, ya que es el objetivo prioritario de la aplicación. En la aplicación no se implementó código para otros sensores y más métodos de transferencia para ahorrar tiempo. Sin embargo, la arquitectura utilizada está diseñada para albergar más sensores y tipos de transferencia.

Para la transferencia se usará el protocolo TCP en vez de UDP para simplificar el código. El protocolo TCP permite establecer conexiones entre computadoras para enviar mensajes sin errores y en el mismo orden en que se transmitieron. En cambio, UDP permite enviar mensajes entre computadoras sin errores, pero sin realizar ningún control sobre el orden y la duplicidad de los mensajes, por lo que sería necesario implementar código para corregir esos errores. Otras formas de conexión como Bluetooth fueron descartadas debido a que no todos los computadores tienen esta característica.

Como ya se mencionó anteriormente, la arquitectura de la aplicación está diseñada para poder tener formas de transferencia que el usuario podría elegir.

4.2 Diseño de la aplicación

La aplicación está construida basándose en el patrón de software *Model View Presenter*, un patrón derivado del patrón *Model View Controller*. Este patrón está compuesto por tres capas el modelo, la vista y el presentador, y está diseñado para separar la interfaz de usuario, la lógica del programa y los datos. Esta separación permite acomodar el código al *framework* de Android, por lo que es usado en aplicaciones para esta plataforma.

El modelo es la capa que contiene todas las clases que guardan los datos y el estado del programa. También puede tratarse de datos externos como una base de datos, en cuyo caso los datos que residen en el ordenador tienen el nombre de repositorio.

El presentador actúa sobre el modelo y la vista. Recupera datos de la capa de modelo y los formatea para mostrarlos en la vista. Implementa la lógica del programa, pero es independiente de la interfaz de usuario desarrollada, lo que permite probar el comportamiento de la aplicación con tests.

La vista es una interfaz pasiva que muestra datos de la capa de modelo. Las acciones del usuario son reenviadas al presentador donde serán procesadas.

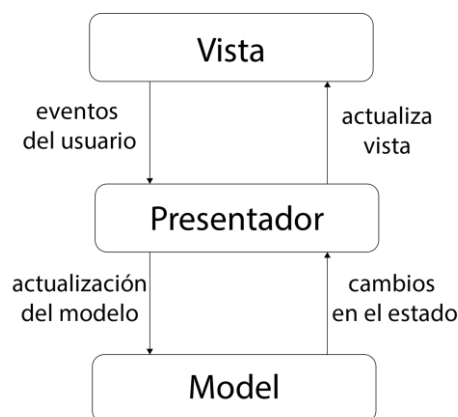


Ilustración 16 Esquema del patrón MVP

En el *framework* de Android la interfaz de usuario es creada usando una clase que hereda de la clase *Activity*. El método habitual usando *MVP* es utilizar esta clase como parte de la capa de vista y únicamente añadir código relacionado con la interfaz de usuario. También pueden pertenecer a la capa de vista clases de Android como *Fragments* o *Vistas*. Es importante apuntar que *MVP* no es un patrón rígido, existen múltiples interpretaciones del mismo.

Los sensores y transmisores no tienen un lugar específico en este esquema. Los sensores se encargan de recoger la información del dispositivo y los transmisores de enviarla a otros dispositivos. Pueden considerarse parte de la vista pues, aunque no formen parte de la interfaz gráfica de usuario, pueden considerarse interfaces de entrada y salida de información. Pero podrían ser parte del modelo, al igual que en algunas interpretaciones del patrón las conexiones a bases de datos están incluidas en esta capa. He optado por esta segunda opción debido a que al contrario que la interfaz de usuario, la aplicación depende de los sensores y transmisores, son un aspecto importante y muy relacionado con los datos que va a procesar el programa. Sin embargo, al igual que en la capa de vista, las clases relacionadas con los sensores y transmisores serán cuidadosamente separados del resto del modelo para evitar dependencias.

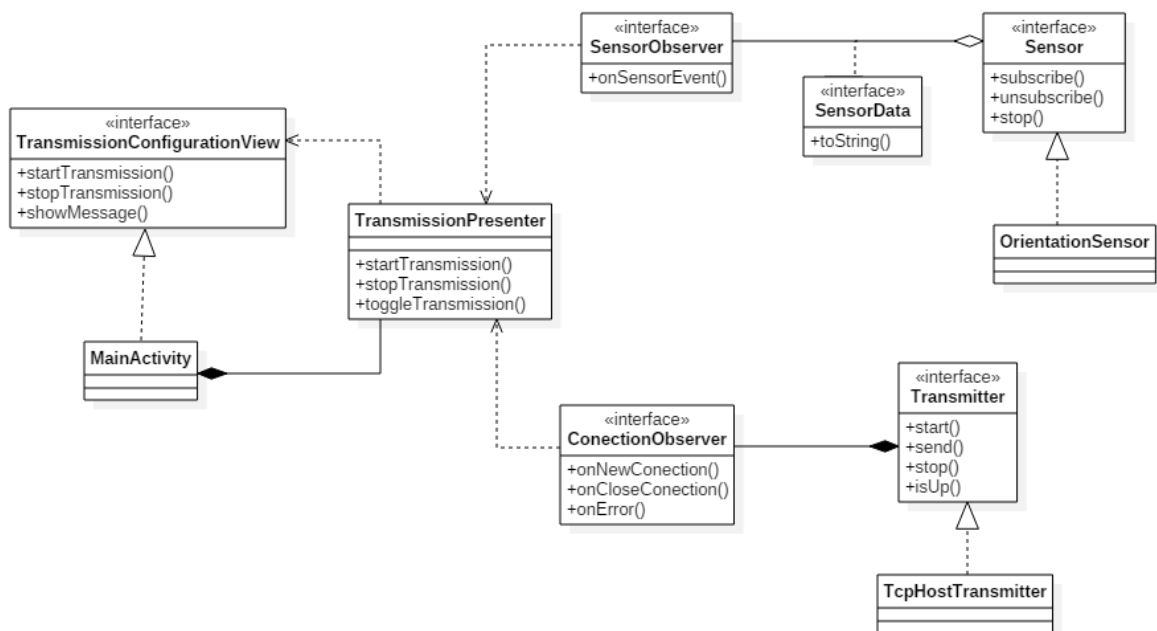


Ilustración 17 Diagrama UML de la aplicación

Los sensores heredan de la interfaz *Sensor* que define métodos para detener el sensor y suscribirse a sus eventos. Cada vez que se registre un evento el sensor enviará los datos del evento a las clases suscritas. Estas implementarán de la interfaz *SensorObserver* y recibirán el evento por el método *onSensorEvent*. *OrientationSensor* es la única clase que implementa *Sensor*, aunque es posible crear más clases que implementen el resto de sensores de Android. La interfaz *Sensor* permite crear más sensores y poder utilizarlos de una manera uniforme y además oculta los detalles de la implementación al resto del código. La interfaz *SensorObserver* mediante el patrón *Observer* evita una dependencia del presentador, quien será el responsable de interactuar con los sensores.

Los transmisores como *TcpHostTransmitter* implementan la interfaz *Transmitter* y son observados por aquellas clases que implementen la interfaz *ConnectionObserver*. El esquema es parecido al usado con los sensores exceptuando que la clase que implemente *ConnectionObserver* debe observar la conexión durante todo el periodo de vida del transmisor y este solo puede ser observado por una clase, ya que una conexión solo debe ser utilizada por una única transmisión, pero los sensores pueden ser compartidos para luego transmitir los datos por varias conexiones.

TransmissionPresenter es la clase de la capa de presentadores donde se implementa la lógica del programa. El presentador interactúa con la vista por medio de la interfaz *TransmissionConfigurationView* eliminando la dependencia de estar ligado a una cierta vista. Esta interfaz es implementada por la clase *MainActivity*, que hereda de la clase del framework de Android *Activity*, e implementa la interfaz gráfica de usuario.

4.3 Implementación de la aplicación

Para realizar la aplicación se utilizó el entorno de desarrollo Android Studio, desarrollado por Google y basado en el entorno de desarrollo IntelliJ IDEA de JetBrains. Este IDE está específicamente diseñado para el desarrollo de aplicaciones para Android y cuenta con versiones para Windows, Mac Os y GNU/Linux.

4.3.1 Actividad e interfaz de usuario

La interfaz de la aplicación está diseñada utilizando la interfaz gráfica que proporciona Android Studio. En la ilustración 18 se puede observar el diseño realizado en el entorno de desarrollo. La

interfaz es codificada en un fichero XML llamado *activity_main.xml*. Esta interfaz gráfica está asociada con la clase *MainActivity* que hereda de la clase *AppCompatActivity*, que es la clase base para las actividades que tengan barra de acción. En la clase *MainActivity* están implementados los comportamientos de la vista tales como desactivar los campos de entrada cuando el servidor está activo.

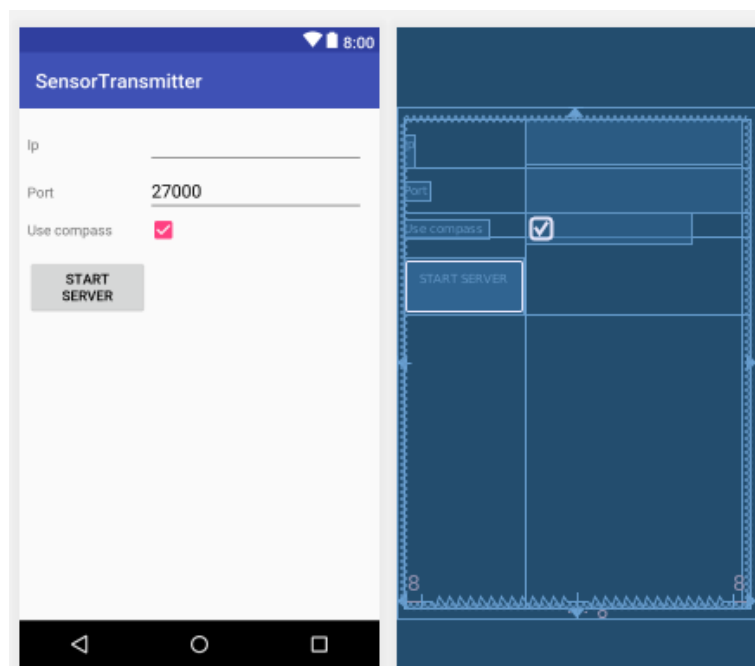


Ilustración 18 Diseño de la GUI en Android Studio

Android Studio también cuenta con un sistema para traducir los mensajes de la aplicación a distintos lenguajes. Los textos de la aplicación se guardan en archivos XML que pueden ser modificados fácilmente desde un editor integrado.


Key	Resource F...	Untranslata...	Default Value	Spanish (es) 
app_name	app/src/mair	<input type="checkbox"/>	SensorTransmi	SensorTransmi
start_server	app/src/mair	<input type="checkbox"/>	Start server	Iniciar servidor
server_started	app/src/mair	<input type="checkbox"/>	Server started	Servidor iniciad
stop_server	app/src/mair	<input type="checkbox"/>	Stop server	Detener servid
port	app/src/mair	<input type="checkbox"/>	Port	Puerto
ip	app/src/mair	<input type="checkbox"/>	Ip	Ip
connected	app/src/mair	<input type="checkbox"/>	Conected	Nueva conexi
disconnected	app/src/mair	<input type="checkbox"/>	Disconnected	Desconectado
use_compass	app/src/mair	<input type="checkbox"/>	Use compass	Usar compas
orientation_sensor	app/src/mair	<input type="checkbox"/>	Orientation Sen	Sensor de orien
accelerometer_not_found	app/src/mair	<input type="checkbox"/>	Accelerometer	Acelerómetro n
error_in	app/src/mair	<input type="checkbox"/>	Error in	Error en

Ilustración 19 Editor de traducciones de Android Studio

Los textos pueden ser accedidos desde la *API* de Android e incluso el entorno de desarrollo muestra el texto por defecto en el código.

Para usar conexiones la aplicación necesita solicitar permiso al móvil en la instalación. Estas deben de estar especificadas en el manifiesto de Android, añadiendo al fichero el código 1.

```
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
```

Código 1 Parte del manifiesto de Android de la aplicación AndroidManifest.xml

El presentador implementa clases anónimas que heredan de *SensorObserver* y *ConexionObserver*. Estas clases se conectan entre ellas para que cada vez que se reciba un dato, este sea enviado a través de los transmisores. También contiene la lógica del programa que es ejecutada cuando se inicia o detiene el servidor, actualizando la vista y controlando los sensores y transmisores del modelo.

El icono de la aplicación, mostrado en la ilustración 20, se ha diseñado en el editor de imagen vectoriales Inkscape.



Ilustración 20 Icono de la aplicación

4.3.2 Sensores

Para obtener los datos de los sensores del dispositivo se debe utilizar el framework de Android. Primero se debe obtener una instancia de *SensorManager* y posteriormente obtener el sensor identificándolo por su tipo. Esta operación se realiza en el constructor de la clase *OrientationSensor* donde se obtienen referencias a los sensores de aceleración y campo magnético.

```
public OrientationSensor(Activity activity, boolean useCompass) throws SensorException
{
    sensorManager = (SensorManager)
activity.getSystemService(Activity.SENSOR_SERVICE);
    android.hardware.Sensor rotationSensor =
sensorManager.getDefaultSensor(android.hardware.Sensor.TYPE_ACCELEROMETER);
    android.hardware.Sensor compassSensor =
sensorManager.getDefaultSensor(android.hardware.Sensor.TYPE_MAGNETIC_FIELD);

    eventListener = new SensorEventListener() {
        @Override
        public void onSensorChanged(SensorEvent event) {
            onSensorEvent(event);
        }

        @Override
        public void onAccuracyChanged(android.hardware.Sensor sensor, int
accuracy) { }
    };

    if (rotationSensor == null) {
        throw new SensorException(
            activity.getResources().getString(R.string.orientation_sensor),
activity.getResources().getString(R.string.accelerometer_not_found));
    } else {
        sensorManager.registerListener(eventListener, rotationSensor,
SENSOR_DELAY);
        if(compassSensor != null && useCompass)
            sensorManager.registerListener(eventListener, compassSensor,
SENSOR_DELAY);
    }
}
```

Código 2 Constructor de la clase OrientationSensor

Si el método *getDefaultSensor* devuelve nulo, el dispositivo no cuenta con el sensor requerido o no se disponen de los permisos necesarios para obtener información del mismo. Una clase anónima llamada *eventListener* implementa la interfaz *SensorListener* para obtener los eventos de los sensores. Cada nuevo evento será recibido en el método *onSensorChanged* que funcionará como *callback*.

La orientación del dispositivo podría haber sido implementada usando un giroscopio, un sensor específicamente diseñado para detectar rotaciones, pero debido a que carezco de un dispositivo con giroscopio era imposible probar que el código funcionaba correctamente.

Gracias al acelerómetro se puede obtener el vector gravedad y con ello calcular la orientación, excepto del eje perpendicular al suelo, pues el vector gravedad no es afectado por rotaciones sobre este eje.

Para obtener la orientación supondremos que no hay movimientos que aceleren el dispositivo y que la aceleración proviene enteramente de la fuerza de la gravedad. Conociendo el vector gravedad se puede calcular el cabeceo y el alabeo del dispositivo, como se observa en la ilustración 21.

El giro sobre el eje perpendicular al suelo, la guiñada, puede calcularse a partir del campo magnético de la tierra utilizando el sensor como una brújula. Este sensor es más ruidoso que el acelerómetro, por lo que, aunque se podría usar únicamente este sensor para calcular la orientación el ruido también se propagaría a los resultados de la orientación. También los campos magnéticos del entorno pueden interferir en los datos del sensor. Por ello se añadió una opción a la aplicación para deshabilitar el uso de este sensor y por consiguiente la detección de la guiñada.

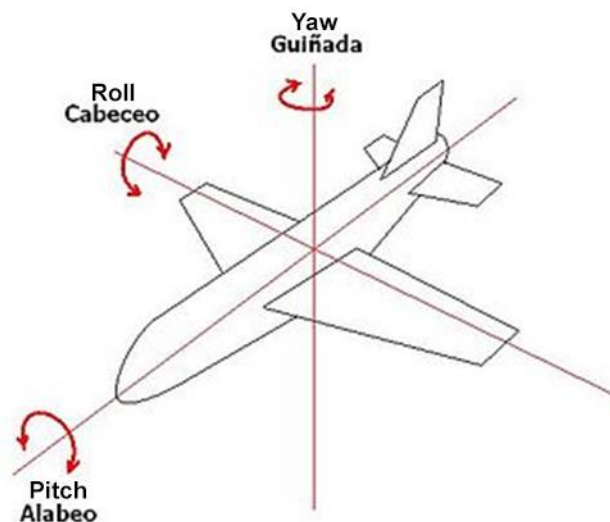


Ilustración 21 Ejes de rotación[31]

Para calcular la orientación primero se normaliza el vector gravedad dividiendo entre la aceleración de la gravedad. Para evitar errores e imprecisiones se trunca el valor del vector en cada eje. Los valores de los ángulos se obtienen utilizando la función multivaluada arco tangente, más conocida

por su abreviación *atan2*, que obtiene el ángulo en el plano formado por las componentes del vector pasadas por parámetro. Por medio de las componentes del sensor de campo magnético que son paralelas al suelo podemos obtener el ángulo de la guiñada, como si fuese una brújula.

```
if(event.sensor.getType() == android.hardware.Sensor.TYPE_ACCELEROMETER) {
    final float gravity = 9.81f;

    float yaw = lastYaw;

    float valuesClamp[] = {
        clamp(event.values[0]/gravity, -1, 1),
        clamp(event.values[1]/gravity, -1, 1),
        clamp(event.values[2]/gravity, -1, 1)
    };

    float roll = (float)Math.atan2(valuesClamp[1], valuesClamp[0]);

    float pitch = (float)-Math.atan2(valuesClamp[2], valuesClamp[0]);

    notifyObservers(new Orientation(yaw, roll, pitch));

} else if(event.sensor.getType() ==
android.hardware.Sensor.TYPE_MAGNETIC_FIELD){
    lastYaw = (float)Math.atan2(event.values[0], event.values[2]);
}
```

Código 3 Cálculo de la orientación en la clase **OrientationSensor**

El parámetro *event* tiene información sobre el evento del sensor, lo que permite identificar de que sensor proviene la información y obtener los datos. En ambos casos, los valores *x*, *y*, *z* pueden obtenerse del array *event.values* en este orden, siendo la componente *y* perpendicular al suelo y *x* la componente que describe derecha e izquierda.

Para cada orientación la aplicación transmitirá los valores de la ilustración 22 sin detectar la guiñada.

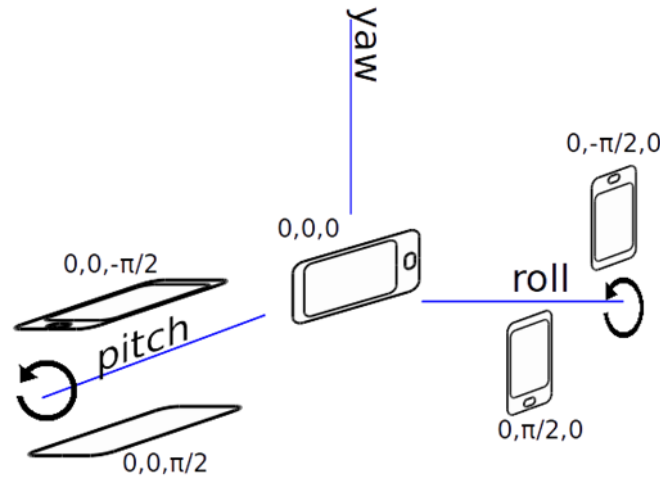


Ilustración 22 Valores transmitidos por la aplicación según la orientación del dispositivo

La orientación es calculada y transmitida usando ángulos de Euler ya que son más simples y pueden utilizarse en OpenGL de manera más sencilla. No existe el riesgo de que suceda un bloqueo de cardán o *gimbal lock* ya que la orientación es recalculada en cada evento sin tener en cuenta la orientación anterior. En el caso de usar giroscopios, los datos obtenidos por el sensor son la rotación con respecto con la última medición, por lo que en este caso sí que podría suceder un *gimbal lock*. Para evitar el error y simplificar el cálculo de la orientación esta debería de estar codificada en cuaterniones.

4.3.2 Transmisores

Para usar conexiones con el protocolo TCP se debe utilizar la API de red de Java. Algunas de las funciones de la librería son bloqueantes, por lo que también es necesario usar hilos. *AsyncTask* es un *wrapper* de Android alrededor de la clase *Thread* de Java diseñada para simplificar tareas usuales, por lo que es muy útil en estas situaciones.

Como el servidor residirá en la aplicación, debe existir un hilo que espere por nuevas conexiones. Por ello el constructor de la clase *TcpHostTransmitter* crea un *AsyncTask* y en el método *doInBackground* se ejecuta la parte de este código que es bloqueante.

```
acceptThread = new AsyncTask<Void, String, Void>() {
    @Override
    protected Void doInBackground(Void... voids) {
        try {
            server = new ServerSocket( port );
            while ( isUp() )
                addClient(server.accept());

        } catch (Exception e) {
            publishProgress("Exception", e.getMessage());
            stop();
        }
        return null;
    }

    private void addClient(Socket client) throws IOException {
        synchronized (clients) {
            clients.add(client);
        }
        client.shutdownInput();
        publishProgress("NewConection", client.getInetAddress().toString());
    }

    protected void onProgressUpdate(String... values) {
        switch (values[0]) {
            case "NewConection":
                observer.onNewConection(values[1]);
                break;
            case "Exception":
                observer.onError(values[1]);
                break;
        }
    }
};
```

Código 4 Hilo de nuevas conexiones declarado en el constructor de `TcpHostTransmitter`

Los *sockets* son un concepto que representa una conexión donde computadoras pueden intercambiar información. Se crea un nuevo *socket* de servidor donde se recibirán los mensajes y mientras el transmisor esté activo acepta nuevas conexiones y añade los *sockets* de estas conexiones a un *ArrayList*. Los accesos a este *ArrayList* están controlados por bloques sincronizados para evitar errores. El método `server.accept` es bloqueante y el hilo permanecerá dormido hasta recibir una nueva conexión. También se cierra el canal de entrada del protocolo *TCP* ya que el cliente no va a enviar ningún dato.

Para enviar un dato se debe usar el *socket* del cliente al que enviar la información en cuestión y escribir el mensaje en su flujo de salida. Los mensajes no deben ser enviados desde el hilo principal,

por lo que es necesario crear un nuevo hilo cada vez que se desee enviar datos. En el hilo se recorren todos los clientes conectados al dispositivo y se envía la información a todos ellos.

```
@Override
public void send(final String message) {

    new Thread( () ->{
        synchronized (clients) {
            ListIterator<Socket> iterator = clients.listIterator();

            while (iterator.hasNext())
                sendToSocket(message, iterator.next(), iterator);
        }
    }).start();
}

private void sendToSocket(String message, Socket client, ListIterator<Socket>
iterator) {
    try {
        if (client.isConnected() && !client.isOutputShutdown() &&
client.isBound()) {
            client.getOutputStream().write(message.getBytes());
        } else {
            observer.onCloseConnection(client.getInetAddress().toString());
            iterator.remove();
        }
    } catch (Exception e) {
        iterator.remove();
    }
}
```

Código 5 Métodos de envío de información de la clase TcpHostTransmitter

Esta clase también cuenta con un método para obtener la IP del dispositivo, para que los clientes puedan conectarse a él.

4.4 Posibles mejoras

La aplicación actualmente solo transmite la orientación captada por los sensores utilizando un servidor *TCP* que retransmitirá la información a cualquier cliente que se conecte.

Las siguientes posibles mejoras no han podido ser implementadas por falta de tiempo.

- Añadir la posibilidad de recoger la información de más sensores y otra información del móvil, como el porcentaje de carga de batería, la hora del dispositivo...

-
- Añadir más transmisores de manera de que la información pueda ser transmitida por diferentes medios.
 - Permitir al usuario elegir como se transmitirán estos datos, como texto plano, como información guardada en bytes, usando formatos como JSON o XML...
 - Añadir la opción de cifrar las conexiones.
 - Crear listas negras y blancas para controlar quien recibe la información.
 - Permitir guardar las configuraciones para que el usuario no tenga que reconfigurar la aplicación en cada uso.
 - Mejorar la interfaz para permitir utilizar todos los ajustes anteriormente comentados.
 - Mostrar la información recibida de las conexiones.

5 OPENGL

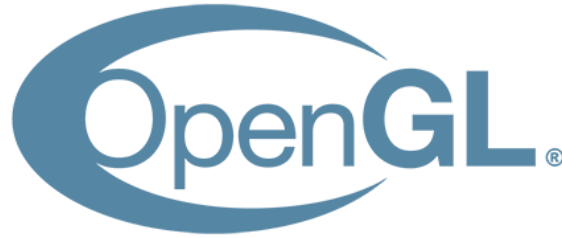


Ilustración 23 Logo de OpenGL

OpenGL es una especificación de una API multilenguaje para la creación de aplicaciones gráficas 2D y 3D. Fue creada en 1992 por Silicon Graphics Inc y actualmente está controlada por el Khronos Group, un consorcio de empresas enfocado a la creación de estándares abiertas. Se usa especialmente en aplicaciones de *CAD* (diseño asistido por ordenador), realidad virtual, representación científica, visualización de información y desarrollo de videojuegos.

OpenGL es una especificación, un documento que establece el comportamiento de un conjunto de funciones. Los fabricantes de hardware implementan librerías a partir de la especificación usando el hardware para realizar el comportamiento que dicta OpenGL. También existen implementaciones que no usan aceleración de hardware como Mesa3D.

Con el paso del tiempo se han ido añadiendo funciones a la especificación y creando nuevas versiones de la misma. Las versiones 1 y 2 destacan por estar enfocadas a tareas más comunes en la programación gráfica. Desde la versión 3.2 OpenGL se centra en dar más control al programador sobre la tarjeta gráfica, permitiendo por ejemplo ejecutar código en ella (*shaders*), pero volviendo obsoletos algunas características anteriores. Las nuevas versiones siguen la misma dirección, por lo que se les suele llamar en conjunto OpenGL moderna. La última versión es la 4.5 y fue publicada en 2014.

Las funciones de OpenGL pueden ser utilizadas desde C y desde otros lenguajes por medio de librerías. Existen variantes de OpenGL destinadas para ciertas ocasiones, como WebGL para ser ejecutado en los navegadores o OpenGL ES para dispositivos de bajo consumo como sistemas empujados o dispositivos móviles.

OpenGL está diseñado como una máquina de estados, de forma que algunas funciones están destinadas a cambiar el estado interno de la librería y por consecuencia su comportamiento. Las funciones de OpenGL van precedidas por 'gl' y pueden finalizar con una indicación del tipo de parámetro que se va a utilizar, por ejemplo, 3f se referirá a un vector de 3 *floats*.

Para renderizar un objeto debemos describirle el objeto a OpenGL indicando la posición de sus vértices y otros aspectos como su color o textura. Esta información se irá juntando y procesando en una cola de tareas llamado *pipeline* de OpenGL. Es posible cambiar el comportamiento de alguna de las tareas utilizando la máquina de estados de OpenGL. En OpenGL moderna ciertos pasos del pipeline pueden ser programados y ejecutados en la GPU. Estos programas tienen el nombre de *shaders*.

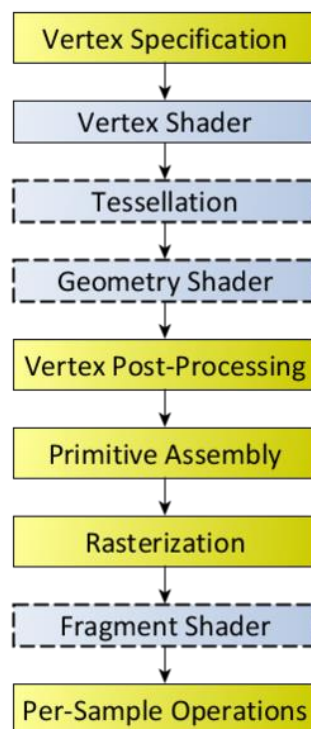


Ilustración 24 Pipeline de OpenGL

Una vez finalizado el *pipeline* de OpenGL y que el objeto haya sido renderizado se vuelca el resultado en buffers a la espera de nuevos objetos renderizados o a imprimir el resultado final en pantalla. Estos *buffers* son llamados *framebuffers* y están compuestos por 3 *buffers*.

El *buffer* de color o *color buffer* es aquel que guarda el color de cada píxel que será enviado a la pantalla. El *buffer* de profundidad o *Z buffer* almacena la profundidad a la que se encuentra cada píxel. Es utilizado principalmente al renderizar varios objetos para evitar que los objetos que se encuentren detrás se rendericen sobre los que estén delante. El *stencil buffer* permite renderizar solo sobre ciertas partes según el valor guardado para cada píxel.

OpenGL usa un sistema de coordenadas llamado *NDC* para posicionar los vértices en la pantalla.

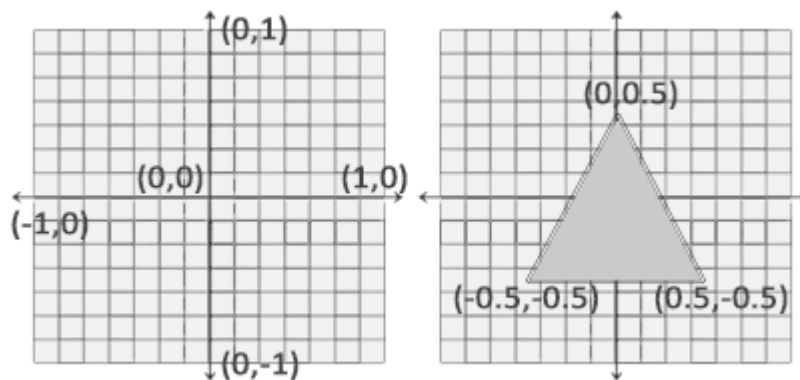


Ilustración 25 Coordenadas *NDC* de OpenGL

Si queremos usar otro sistema de coordenadas más adecuado para entornos tridimensionales para expresar donde están y como se ven los objetos debemos transformar las posiciones de un sistema de coordenadas a otro. Las transformaciones son realizadas multiplicando las posiciones, que son expresadas como vectores, con matrices.

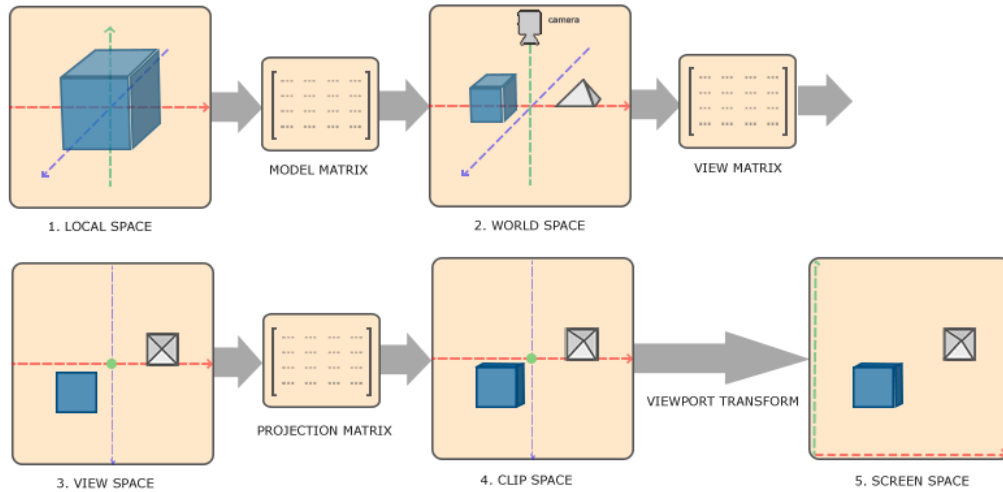


Ilustración 26 Transformaciones de OpenGL

Se usan 3 matrices para transformar posiciones entre diferentes sistemas de coordenadas. El primer sistema de coordenadas, el espacio local es en el que usualmente están codificados los modelos 3D y cada posición está determinada por una referencia que depende del modelo. Al multiplicar las posiciones por la matriz de modelo las posiciones pasan a estar determinadas por un punto de referencia en el mundo, por lo que es llamado el espacio global. Cambiando la matriz de modelo podemos mover el objeto en el mundo. La matriz de vista transforma las posiciones para que el punto de referencia sea el punto de vista desde el que se observa el mundo, es decir desde la cámara y finalmente la matriz de proyección transforma las posiciones a las coordenadas de *NDC* de OpenGL aplicándoles una proyección como la ortográfica o la perspectiva. Por último, se plasma el resultado en una región de la pantalla delimitado por el *viewport*.

Resumiendo, la multiplicación de estas matrices transforma posiciones con respecto del objeto a coordenadas *NDC*.

$$V_{clip} = M_{proyección} \cdot M_{vista} \cdot M_{modelo} \cdot V_{local}$$

Estas matrices pueden ser especificadas con funciones de OpenGL o transmitidas directamente a las shaders donde se realizará el cálculo anterior.

Un programa de *shader* está compuesto por varias partes, las principales son la *shader* de vértices y la *shader* de fragmentos. Estas *shaders* son códigos que se ejecutan en la tarjeta gráfica que realizan acciones del *pipeline* de OpenGL. La *shader* de vértices que se ejecuta una vez por vértice y permite especificar la posición final de este. La *shader* de fragmento se ejecuta por cada píxel de la forma delimitada por los vértices y especifica su color y profundidad. En el *pipeline* de OpenGL se ejecuta primero la *shader* de vértices y posteriormente la de fragmentos.

La *shader* de vértices puede transmitir parámetros a la *shader* de fragmentos de manera que estos se interpolen según la posición con los respectivos vértices. De esta forma datos como las coordenadas de la textura son interpolados a partir de los vértices y en la *shader* de fragmentos se asigna a cada píxel la coordenada correspondiente.

En OpenGL las *shaders* se programan en GLSL, un lenguaje similar a C, pero con nuevos tipos primitivos como matrices y vectores. Para enviar datos entre *shaders* se utilizan las instrucciones *in* y *out* seguidas de la declaración de la variable. Del mismo modo se pueden enviar variables a las *shaders* desde OpenGL usando *uniforms*. En la *shader* se pueden obtener utilizando la instrucción *uniform* seguida de la declaración de la variable. Desde OpenGL se envían los datos utilizando la llamada *glUniformX*, donde X es una abreviación del tipo de dato.

5.1 Librerías asociadas a OpenGL

Todas las funciones de OpenGL están destinadas a la creación de gráficos, por lo que al realizar una aplicación se necesitará de otras librerías para conseguir funcionalidades como la gestión de eventos, control de ventanas, reproducción de audio, lectura de ficheros, etc. que pueden ser necesarios en la aplicación. Para suplir esta necesidad se crearon librerías destinadas a trabajar junto con OpenGL, pero que no están soportadas por el Khronos Group.

5.1.1 GLUT

GLUT del inglés OpenGL Utility Toolkit, es una librería multiplataforma de utilidades que complementan OpenGL con funciones como control de ventanas, *timers*, interacción teclado y ratón, renderización de primitivas geométricas, renderización de textos. Está diseñada para integrarse en proyectos de OpenGL simples y facilitar el aprendizaje de OpenGL.

GLUT controla el curso del programa y permite al desarrollador añadir *callbacks* que se accionarán con determinados eventos. Las funciones *glutMouseFunc* y *glutKeyboardFunc* especifican *callbacks* para los eventos de ratón y teclado. La función *glutMainLoop* inicia el bucle principal que llama en cada iteración al *callback* especificado con la función *glutDisplayFunc*, donde se renderizarán los objetos.

5.1.2 GLM

OpenGL Mathematics, abreviado GLM, es una librería matemática para programas de gráficos basado en el lenguaje GLSL usado en las shaders de OpenGL. La librería provee de tipos para vectores y matrices de hasta tamaño 4 y un gran número de operaciones relacionadas con el ámbito gráfico, incluyendo operaciones primarias como la suma o el producto con sobrecarga de operadores. Existen funciones diseñadas para manipular matrices de transformación como rotar, trasladar, escalar, invertir. Suministra funciones para generar matrices de proyección ortográfica, perspectiva o con un determinado *frustum*. Otras funciones son conversión entre formatos de color, interpolaciones, trigonometría, generación aleatoria, ruido y otras funciones matemáticas.

5.1.2 STB

Las librerías STB proporcionan diversas funcionalidades simplemente incluyendo una cabecera. Para este proyecto se ha usado la librería *stb_image.h* que carga en memoria imágenes de tipo JPG, PNG, TGA, BMP, PSD, GIF, HDR y PIC.

5.1.2 Assimp

Assimp es una librería de lectura de modelos 3D capaz de procesar numerosos formatos. Permite además de leer la geometría del modelo obtener aspectos avanzados de los modelos 3D como mapeado de texturas, huesos, animaciones y múltiples texturas y materiales. También permite exportar modelos 3D.

6 DESARROLLO DEL DEMOSTRADOR EN OPENGL

El demostrador utiliza OpenGL para generar un entorno virtual usando la visión estereoscópica. El uso de OpenGL ha posibilitado el poder aplicar la teoría de cámaras estereoscópicas, ya que, con otras herramientas como motores gráficos, las cámaras suelen estar sujetas al código del motor y modificarlas podría ser complicado. En OpenGL los frustums de las cámaras se pueden realizar simplemente creando matrices de vista.

Como entorno de desarrollo del programa se usó Visual Studio, un IDE creado por Microsoft dedicado a la creación de aplicaciones en múltiples lenguajes. Soporta lenguajes creados por Microsoft como C#, F#, Visual Basic y otros lenguajes como Python, PHP y C++. Está disponible para Windows y Mac. Para este TFG se usó C++.

El demostrador ha sido implementado dividiendo el código en clases para facilitar la demostración y que pueden ser reutilizadas en nuevas experiencias virtuales. Las clases de demostración pueden ser reutilizadas en la asignatura de Creación de Interfaces de Usuario ya que en ambas se utiliza OpenGL. La parte específica de la demostración se ha realizado en una clase que utiliza el resto de clases para crear la demostración.

Para poder compilar el proyecto o usar las clases reutilizables es necesario configurar el entorno de desarrollo, ya que se utilizan librerías que por defecto no están incluidas.

El demostrador creado muestra al usuario los planetas y otros objetos del sistema solar como se puede observar en las imágenes 27 y 28. La cámara se mueve entre los objetos girando la cámara para que el usuario los pueda observar.

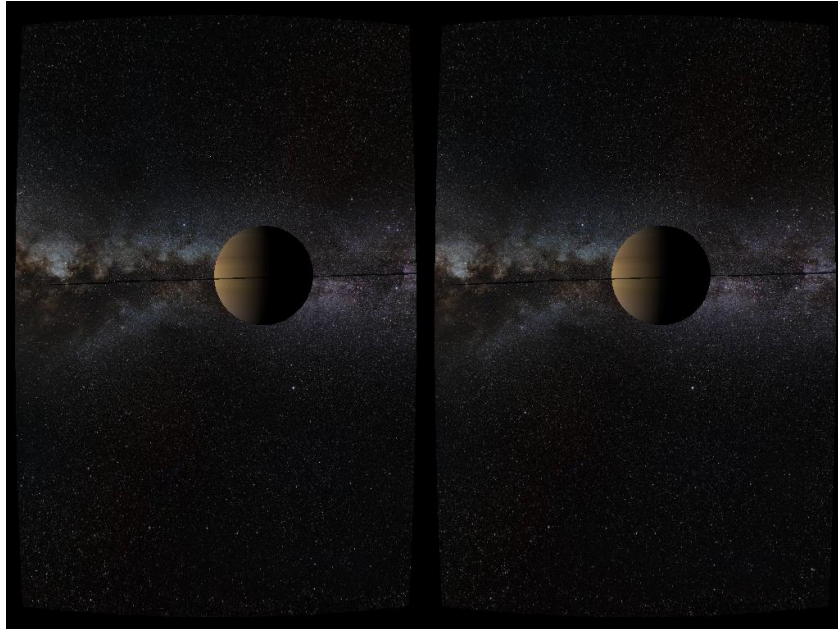


Ilustración 27 Imagen estereoscópica de Saturno generada por el demostrador

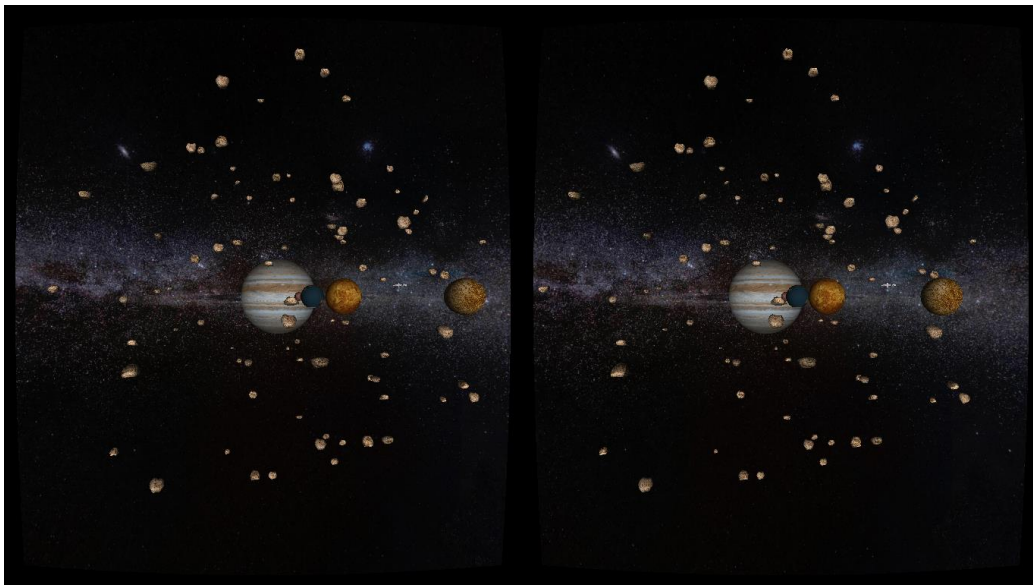


Ilustración 28 Imagen estereoscópica del sistema soar generada por el demostrador

6.1 Cascos de realidad virtual utilizados

Los cascos de realidad virtual utilizados en el trabajo funcionan introduciendo un móvil en su interior y usándolo como pantalla. Las lentes del casco pueden moverse y ajustarse por medio de los controles de la parte superior. Los cascos no traían ningún dato técnico de los mismos.



Ilustración 29 Cascos de realidad virtual utilizados vistos de frente



Ilustración 30 Cascos de realidad virtual utilizados



Ilustración 31 Cascos junto con el dispositivo móvil

6.2 Partes del programa

6.2.1 *Fullscreen*

El espacio de nombre *Fullscreen* contiene funciones para poner la aplicación en pantalla completa. Existen 2 funciones, la función *set* que llama a una función de la librería GLUT para cambiar la ventana usada a pantalla completa y la función *set_on_screen* que cambia a pantalla completa la aplicación en la pantalla indicada. A través de la librería GLUT no se puede realizar esta operación, por lo que se debe utilizar la *API* del sistema operativo para realizarlo. Otras librerías más modernas como GLFW sí implementan esta funcionalidad. Como la demostración está dirigida a Windows solo se ha implementado código para este sistema operativo. En otros sistemas operativos llamar a la función tendrá el mismo efecto que llamar a la función *set*, pero aparecerá una alerta en la compilación avisando de este comportamiento.

Para cambiar una ventana a modo pantalla completa en una pantalla concreta hay que localizar la ventana. GLUT tampoco posibilita la obtención del manejador o *handler* de ventanas, por lo que es necesario encontrar la ventana por su nombre o suponer que es la ventana activa. GLUT tampoco permite obtener el nombre de una ventana que se ha creado con la librería por lo que debe ser especificada en los parámetros de la función.


```
void FullScreen::set_on_screen(int index, const char* windowName)
{
#ifdef _WIN32

    HWND hwnd = windowName == nullptr ? GetForegroundWindow() : FindWindow(NULL,
windowName);

    struct ScreenParameter {
        RECT screenRect;
        int screenIndex;
    }screenParameter{ RECT(), index};

    EnumDisplayMonitors(NULL, NULL, [](HMONITOR hMonitor, HDC hdcMonitor, LPRECT
lprcMonitor, LPARAM dwData) -> int {
        MONITORINFOEX iMonitor;
        iMonitor.cbSize = sizeof(MONITORINFOEX);
        GetMonitorInfo(hMonitor, &iMonitor);

        ScreenParameter& info = *(ScreenParameter*)dwData;
        info.screenRect = iMonitor.rcMonitor;

        if (info.screenIndex == 0) {
            return false;
        }
        else {
            info.screenIndex--;
            return true;
        }
    }, (LPARAM)&screenParameter);
```

Código 6 Comienzo de función `set_on_screen` de la clase `FullScreen`

Como se observa en el código 6 de la función `set_on_screen`, con la función de Windows `GetForegroundWindow` se obtiene la ventana activa si no se ha especificado un nombre y con `FindWindow` la API de Windows busca una ventana con el nombre indicado. Con la función de Windows `EnumDisplayMonitors` se obtiene la información de las pantallas conectadas al equipo a través de una función que será llamada por cada pantalla. Esta forma de obtener información a través de *callbacks* es ampliamente usada en la API de Windows y las lambdas introducidas en la versión 11 de C++ pueden utilizarse como se muestra en el código.

El último argumento de la función permite pasar datos por parámetros al callback y ser utilizados como parámetros de salida. En este caso se usa `screenIndex` como parámetro de entrada y `screenRect` como parámetro de salida. En cada iteración en el callback `screenIndex` se decrementa guardando las iteraciones que faltan hasta alcanzar la iteración de la pantalla seleccionada. Al devolver falso se

deja de llamar a la función. En *screenRect* se guarda el rectángulo que ocupa la pantalla dentro del espacio virtual de pantalla, aquel formado por todas las pantallas conectadas al equipo.

```
SetWindowLong(hwnd, GWL_STYLE, WS_POPUP | WS_MAXIMIZE);
SetWindowPos(hwnd, HWND_NOTOPMOST,
    screenParameter.screenRect.left, screenParameter.screenRect.top,
    screenParameter.screenRect.right - screenParameter.screenRect.left,
    screenParameter.screenRect.bottom - screenParameter.screenRect.top,
    SWP_SHOWWINDOW);

#else
    #warn "FullScreen on a selected screen is not implemented for this OS. Use
    FullScreen::set"
    FullScreen::set();
#endif
}
```

Código 7 Final de la función *set_on_screen* de la clase *FullScreen*

Finalmente, el resto de la función, mostrada en el código 7, cambia la ventana a modo pantalla completa y ajusta su tamaño a la pantalla concreta.

6.2.2 *NetReceiver*

La clase *NetReceiver* conecta con la aplicación *Sensor Transmitter* y recibe los datos de los sensores. La clase implementa un cliente *TCP* utilizando *Winsock*, la *API* de *sockets* de Windows, por lo que la clase no está preparada para ser utilizada desde otros sistemas operativos. Actualmente no existe ninguna implementación de una extensión de red multiplataforma en la librería estándar de C++, solo borradores en proceso de estandarización. El código está basado en el ejemplo de cliente *TCP* de la documentación de Microsoft y otros ejemplos de internet.

Primero se inicializa la librería *WinSock* requiriendo la última versión 2.2 usando la macro *MAKEWORD*. En *wsaData* se obtiene la información sobre la implementación de *Winsock*. Posteriormente se construye el socket *TCP/IP* que se usará para conectarse con el servidor. *WSACleanup* siempre debe ser llamado cuando se termine de usar la librería.

```
bool NetReceiver::start()
{
    WSADATA wsaData;

    // Initialize Winsock
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) return false;

    // Create a SOCKET for connecting to server
    connectSocket = socket(AF_UNSPEC, SOCK_STREAM, IPPROTO_TCP);
    if (connectSocket == INVALID_SOCKET) {
        WSACleanup();
        return false;
    }
}
```

La estructura *sockaddr_in* contiene los aspectos de la conexión que se usarán al conectarse. La dirección ip debe estar codificada numéricamente y el puerto especificado en big-endian.

```
sockaddr_in serverAddr;
inet_pton(AF_INET, server.c_str(), &serverAddr.sin_addr);
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(port);

// Connect to server.
if (connect(connectSocket, (const sockaddr *)&serverAddr, sizeof(serverAddr)) ==
SOCKET_ERROR) {
    closesocket(connectSocket);
    connectSocket = INVALID_SOCKET;
    WSACleanup();
    return false;
}
```

Finalmente se cierra el canal de escritura, ya que el programa no enviará ningún dato y se inicia el hilo donde se recibirán los datos.

```
// shutdown the connection since no more data will be sent
if (shutdown(connectSocket, SD_SEND) == SOCKET_ERROR) {
    closesocket(connectSocket);
    WSACleanup();
    return false;
}

recvThread = std::thread(NetReceiver::recvThreadFunction, this);

return true;
}
```

6.2.3 StereoVision

StereoVision es una clase estática que permite dibujar imágenes estereoscópicas con funciones similares a las implementadas por la librería GLUT. La clase tiene tres funciones que especifican *callbacks* que serán llamados en determinados momentos del bucle principal. Las funciones *BeforeDisplay* y *AfterDisplay* especifican las funciones que serán llamadas antes y después de la renderización. *DisplayFunc* es similar a la función *glutDisplayFunc* y especifica una función que será llamada dos veces, una para la renderización por cada ojo y con un parámetro indicando cual. A esta función le daremos el nombre de función de renderización.

DisplayFunction usa internamente *glutDisplayFunc* para poner una función interna que será llamada en cada iteración del bucle principal de GLUT. En esta función se llaman a las funciones especificadas con los tres métodos y se realiza las operaciones necesarias para crear un efecto estereoscópico a partir de las imágenes renderizadas.

Para realizar el efecto estereoscópico se debe renderizar la escena dos veces y poner los resultados uno al lado del otro como en la ilustración 27. La función de renderización es la encargada de dibujar la escena, pero en vez de volcar el resultado en el *framebuffer* principal lo hará en otros dos *framebuffers* creados específicamente para este proceso. Estos *framebuffers* se inicializarán en el método *init_framebuffer* del código 8. Esta función se llamará dos veces, una por cada vista del estéreo.

Para crear un *framebuffer* se debe generar mediante la función *glGenFramebuffers* y para asignar sus propiedades o utilizarlo se debe vincular con la función *glBindFrameBuffer*. Las llamadas que se realicen a continuación y que utilicen un *framebuffer* utilizarán el último que fue vinculado, ya que OpenGL se comporta como una máquina de estados. Con *glGenTextures* se genera una textura que se usará para guardar el *buffer* de color del *framebuffer*. Como esta textura almacena la imagen que se ha renderizado tendrá las dimensiones de la mitad de la pantalla y no guardará transparencias

Posteriormente la textura se mostrará con las mismas dimensiones con las que se generó, por lo que no es necesario usar un filtro de reescalado. Por esta razón se aplica el filtro de *GL_NEAREST* que selecciona el píxel de la textura más cercano a aquel que se va a mostrar cuando se renderice la textura. Estas opciones son indicadas con la función *glTexParameterX* a la textura vinculada con

glBindTexture. Con *glFramebufferTexture2D* se vincula la textura al *framebuffer* para que se almacene los objetos renderizados en la textura.

```
void StereoVision::init_frameBuffer(unsigned int &frameBuffer, unsigned int
&textureColorBuffer)
{
    glGenFramebuffers(1, &frameBuffer);
    glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);

    // create a color attachment textureId
    glGenTextures(1, &textureColorBuffer);
    glBindTexture(GL_TEXTURE_2D, textureColorBuffer);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE,
    NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
    textureColorBuffer, 0);
    // create a renderbuffer object for depth and stencil attachment (we won't be
    sampling these)

    unsigned int rbo;
    glGenRenderbuffers(1, &rbo);
    glBindRenderbuffer(GL_RENDERBUFFER, rbo);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, width, height); // use
    a single renderbuffer object for both a depth AND stencil buffer.
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
    GL_RENDERBUFFER, rbo); // now actually attach it
}
```

Código 8 Inicialización de los framebuffers en la clase StereoVision

Cada renderización en estéreo necesitará un *buffer* de profundidad y un *stencil buffer*, por lo que se crea un *renderbuffer* con estos dos *buffers* y se vincula al *framebuffer* con *glFramebufferRenderBuffer*.

Un *renderbuffer* es similar a un *buffer* renderizado a una textura, como en el caso del *buffer* de color. Se utilizan para almacenar el *buffer* indicado, pero en vez de en forma de textura se realiza en el formato nativo de OpenGL, de forma que agiliza los procesos como guardar y copiar, pero no es posible leer directamente el buffer como se haría con una textura, por lo que generalmente son de solo escritura. Se podría optar por usar un solo *renderbuffer* compartido con cada *framebuffer* del estéreo si este es borrado al comienzo, solo habría que realizar una sola inicialización del *renderbuffer*

y vincularlo con cada *framebuffer*. Como esta clase no está dirigida a un uso específico se mantendrá uno por cada *framebuffer*.

Tras inicializar los *framebuffers* se comienza el bucle principal de GLUT y se realiza el proceso de renderización en estéreo. Cada llamada a la función especificada en *display_function* utilizará un *framebuffer* diferente y al finalizar las texturas de los *framebuffers* se renderizarán una al lado de la otra. Para ello se utilizará una proyección ortográfica y un quad sobre el que se proyectará la textura, como se muestra en la ilustración 32. Para renderizarlo se usa una *shader* que contrarresta la distorsión de las lentes.

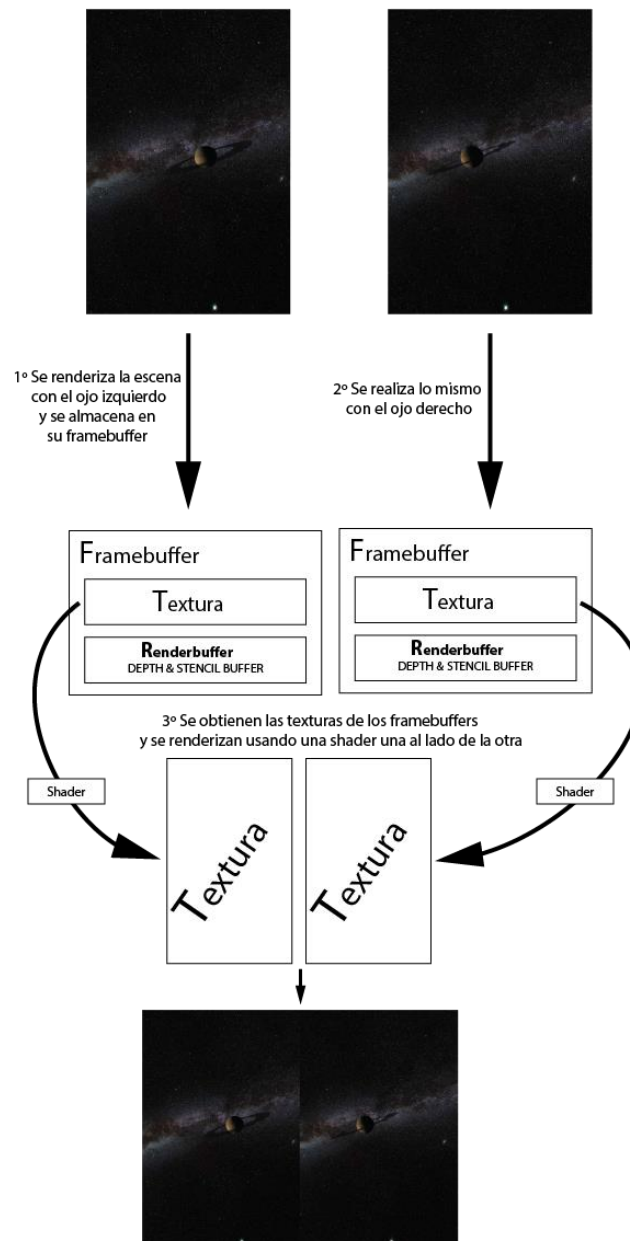


Ilustración 32 Proceso de renderización en estéreo

Antes de hacer cada llamada a la función especificada en *display_function* se debe vincular el *framebuffer* correcto con *glBindFramebuffer*. Si se llama a esta función con el identificador 0, se vinculará el *framebuffer* principal y por tanto los resultados se imprimirán en pantalla. Se vincula el *framebuffer* principal, se activa la *shader* para renderizar las texturas y se asigna la distorsión

especificada. Se aplica una proyección ortográfica llamando antes a *glPushMatrix* para evitar afectar a otras partes del código y se dibujan los quads con las texturas. Finalmente se desactiva la *shader* con *glActiveShader(0)*, se hace un *glPopMatrix* para desechar las modificaciones de la matriz de proyección y se imprime en pantalla usando el doble buffer de GLUT.

```
void StereoVision::draw_function()
{
    beforeFunc();

    glViewport(0, 0, width, height);

    glBindFramebuffer(GL_FRAMEBUFFER, rightFramebuffer);
    displayFunc(true);
    glBindFramebuffer(GL_FRAMEBUFFER, leftFramebuffer);
    displayFunc(false);

    glBindFramebuffer(GL_FRAMEBUFFER, 0);

    glClearColor(0, 0, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // borra todo lo existente en
el framebuffer

    glUseProgram(shaderId);
    glProgramUniform1f(shaderId, glGetUniformLocation(shaderId, "BarrelPower"),
barrelPower);

    set_orto_projection();

    glViewport(0, 0, glutGet(GLUT_WINDOW_WIDTH), glutGet(GLUT_WINDOW_HEIGHT));

    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, rightTextureColorbuffer);

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();

    float leftText = 0.f;
    float rightText = 1.f;

    glBegin(GL_QUADS);
    glTexCoord2f(leftText, 0);
    glVertex3f(0, -1, -1);

    glTexCoord2f(leftText, 1);
    glVertex3f(0, 1, -1);

    glTexCoord2f(rightText, 1);
    glVertex3f(1, 1, -1);

    glTexCoord2f(rightText, 0);
```



```
glVertex3f(1, -1, -1);

glEnd();

glBindTexture(GL_TEXTURE_2D, leftTextureColorbuffer);

glBegin(GL_QUADS);
glTexCoord2f(leftText, 0);
glVertex3f(-1, -1, -1);

glTexCoord2f(leftText, 1);
glVertex3f(-1, 1, -1);

glTexCoord2f(rightText, 1);
glVertex3f(0, 1, -1);

glTexCoord2f(rightText, 0);
glVertex3f(0, -1, -1);

glEnd();

glUseProgram(0);

glPopMatrix();
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glBindTexture(GL_TEXTURE_2D, 0);

glFlush();
glutSwapBuffers();
afterFunc();

}
```

Código 9 Función draw_function que dibuja las imágenes en estéreo

La *shader* realiza una distorsión de barril para contrarrestar la distorsión provocada por las lentes. Esta distorsión aleja los píxeles del centro de forma cuadrática a su distancia. Como se puede ver en la ilustración 33, la imagen de la izquierda está distorsionada para que a través de los cascos de realidad virtual se vea la imagen original, la de la izquierda.

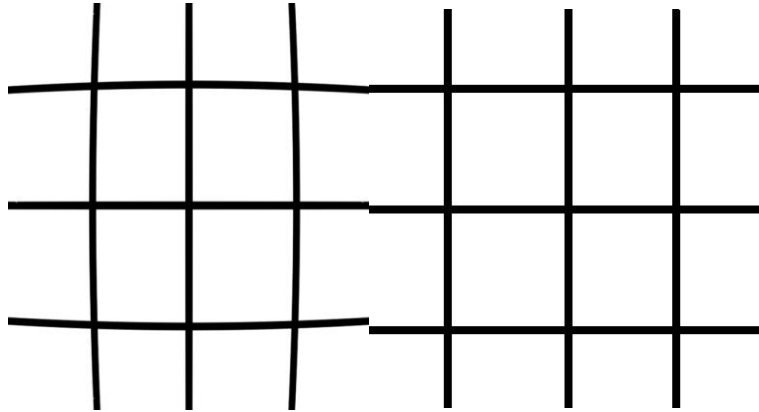


Ilustración 33 Distorsión de barril en lentes

La distorsión aplicada en la ilustración es la misma que se aplica en los cascos. Los cascos no proveían de ninguna información sobre las lentes, por lo que los valores se han ajustado a ojo. Se ha usado un método simple ya que las lentes no generan mucha distorsión. En otros cascos las lentes pueden ser más potentes y puede ser necesario usar algoritmos más complejos. Aparte el código supone que el centro de la lente enfoca al centro de la parte de la pantalla que le corresponde, cuando esto puede ser modificado por los ajustes del casco.

En la *shader* de vértices del código 10 únicamente obtiene la posición del vértice a través de *gl_Vertex* y se aplican las matrices de transformación. Se almacena la posición de la textura guardada en *gl_MultiTexCoord0* para posteriormente usarla en la *shader* de fragmentos, transmitiéndola a través de la variable *texPos*.

```
#version 120

varying out vec2 texPos;

void main() {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    texPos = gl_MultiTexCoord0.xy;
}
```

Código 10 Shader de vértices del programa que contrarresta la distorsión de las lentes

En la *shader* de fragmentos del código 11 especificaremos que píxel de la textura usar. En la función *main* usaremos el píxel de la textura indicado por la función *Distort* que realiza la distorsión de barril.

El parámetro *texPos* indica la posición de la textura que sería dibujada si no se aplicase la distorsión. A partir del parámetro en la función *Distort* se calcula la posición en coordenadas desde el centro de la textura en vez de un vértice. Este proceso debe ser restablecido al finalizar la función. A partir de estas nuevas coordenadas se calcula el radio al cuadrado y se aplica este nuevo radio según la cantidad especificada por *barrelPower*.

```
#version 120

varying in vec2 texPos;

uniform sampler2D texture;
uniform float BarrelPower;

vec2 Distort(vec2 pos)
{
    vec2 p = 2.0 * pos - 1.0;

    float r2 = p.x * p.x + p.y * p.y;
    float newRadius = (1 + BarrelPower*r2);
    p = p * newRadius;

    return (p + 1.0) / 2.f;
}

void main() {

    gl_FragColor = texture2D(texture, Distort(texPos));
}
```

Código 11 Shader de fragmentos del programa que contrarresta la distorsión de las lentes

Si no se decidiese usar una *shader* para corregir aspectos de las lentes o realizar postprocesado, se podría haber usado simplemente *glViewport* para dibujar únicamente en una parte de la pantalla y evitar usar framebuffer.

6.2.4 Cámaras

Se denomina cámara al objeto que controla el punto vista en programación gráfica. Las cámaras generan las matrices de vista y proyección a partir de parámetros indicados como el campo de visión, la posición de la cámara o la dirección enfocada. Las cámaras implementadas tienen un método para dirigir la vista hacia un punto y otro para rotar la cámara sobre el eje hacia el punto de vista, es decir su alabeo.

Para la demostración han sido creadas además de una clase para una cámara con visión monocular, cámaras en estéreo usando los diferentes métodos descritos en el capítulo de estereoscopia y cascos de realidad virtual. De la cámara con visión monocular hereda la cámara *3D* que implementa el método *Off-Axis*, el método más utilizado. De esta clase heredan cámaras que utilizan los otros dos métodos, *Toe-In* y *Offset Frustums*.

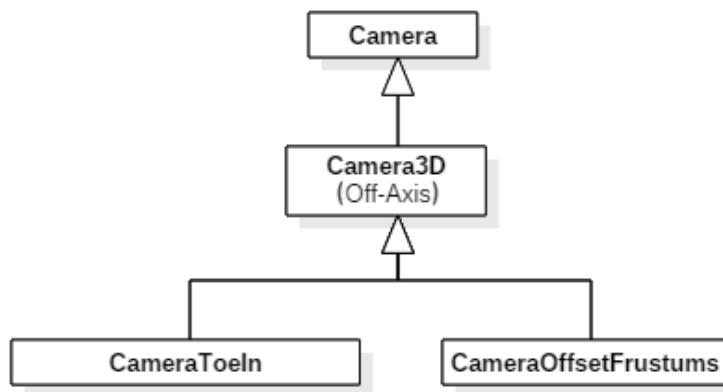


Ilustración 34 Relaciones de herencia entre las clases de cámaras

La clase *Camera* implementa todos los métodos de control de la cámara exceptuando aquellos dedicados a cámaras estereoscópicas. Para obtener la matriz de proyección con perspectiva se utiliza la función *perspective* de la librería GLM, pasando por parámetros la amplitud del campo de visión, la relación de aspecto de la pantalla donde se imprimirá la imagen y los márgenes cercanos y lejanos donde se trunca el renderizado.

```
glm::mat4x4 Camera::get_projection_matrix()
{
    return glm::perspective(fov, aspect_ratio(), _near, _far);
}
```

Código 12 Método *get_projection_matrix* de la clase *Camera*

Para obtener la matriz de vista el método se ayuda de la función *lookAt*. Esta función devuelve una matriz de vista especificando la posición de la cámara, la posición hacia donde mira la cámara y el vector hacia arriba de la cámara.

```
glm::mat4x4 Camera::get_view_matrix()
{
    return glm::lookAt(_position, _position + _front, _up);
}
```

Código 13 Método `get_view_matrix` de la clase `Camera`

El vector arriba suele tener el valor $\{0, 1, 0\}$, pero al rotar la cámara este vector cambia. Los vectores arriba y derecha se recalculan cada vez que se modifica la rotación o la dirección de la cámara. Para hallar el vector derecho se realiza el producto vectorial de la dirección de la cámara con el vector arriba $\{0, 1, 0\}$. El vector obtenido es el vector derecha sin aplicar ninguna rotación, aplicando una matriz de rotación al vector derecha se obtiene el vector derecha final. Para obtener el vector arriba simplemente se realiza el producto vectorial del vector derecha con la dirección de la cámara.

```
void Camera::recalculate_vectors()
{
    _right = glm::normalize(glm::cross(_front, glm::vec3(0, 1, 0)));
    _right = glm::vec3(glm::rotate(glm::mat4(1), _rotation, _front) *
glm::vec4(_right, 0));
    _up = glm::normalize(glm::cross(_right, _front));
}
```

Código 14 Método `recalculate_vectors` de la clase `Camera`

Las cámaras en estéreo utilizan procedimientos similares, pero crean matrices de proyección y vista para cada ojo. La clase `Camera3D` implementa el método *Off-Axis* que usa *frustums* asimétricos dirigidos hacia el frente, como se puede observar en la ilustración 6. La matriz de vista es construida de igual manera que en la clase `Camera` pero relativo al punto de vista de cada lente.

```
glm::mat4x4 Camera3D::get_view_matrix(Lens lens)
{
    const glm::vec3 lens_position = _position + offset(lens) * _right;
    return glm::lookAt(lens_position, lens_position + _front, _up);
}
```

Código 15 Método `get_view_matrix` de la clase `Camera3D`

La proyección se realiza mediante *frustums* asimétricos para que las vistas coincidan sobre el plano de proyección. Los *frustums* simétricos tienen las mismas dimensiones en los lados derecho e izquierdo, y en los lados arriba y abajo. El *frustum* de la proyección no tiene los lados derecho e izquierdo iguales. Para calcular las dimensiones de estos lados partimos de un frustum simétrico, como se puede observar en la ilustración 35, donde la longitud de cada lado se puede obtener con

la tangente del ángulo que lo forma, la mitad del campo visual, multiplicado por la distancia al margen cercano del *frustum*

$$lado_{simétrica} = \tan\left(\frac{fov}{2}\right)near$$

Para que coincidan las vistas en el plano de proyección es necesario torcer el *frustum* hacia el centro, reduciendo y aumentando la longitud de los lados una cierta cantidad. Esta cantidad es la mitad de la distancia interocular en la distancia donde se encuentra el plano de proyección y en el margen cercano del *frustum* la cantidad se escala con la razón del margen cercano con la distancia al plano de proyección.

$$lado_{asimétrico} = lado_{simétrico} \pm \frac{near}{center} offset$$

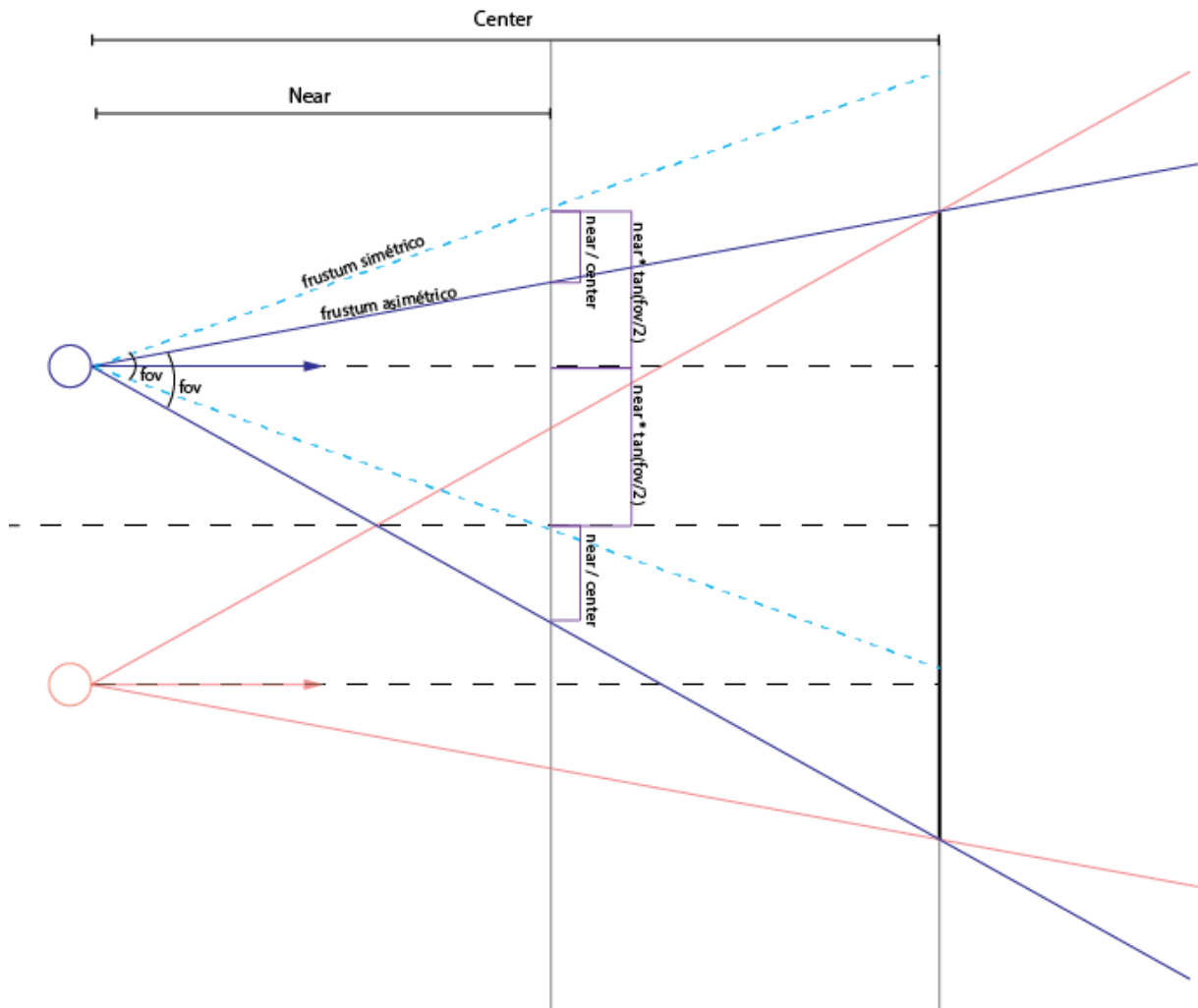


Ilustración 35 Esquema de distancias en el método Off-Axis

El código 19 implementado en la función realiza las operaciones descritas y crea el *frustum* utilizando la librería GLM.

```
glm::mat4x4 Camera3D::get_projection_matrix(Lens lens)
{
    const float wd2 = _near * tan(fov / 2.f);
    const float ndf1 = _near / center;

    const float coef = offset(lens);
    const float ratio = aspect_ratio();

    const float left = -ratio * wd2 + coef * ndf1;
    const float right = ratio * wd2 + coef * ndf1;
    const float top = wd2;
    const float bottom = -wd2;

    return glm::frustum(left, right, bottom, top, _near, _far);
}
```

Código 16 Método `get_projection_matrix` de la clase `Camera 3D`

En el método *Toe-In* los frustums están orientados a un cierto punto, como se puede ver en la imagen 5. Este punto se sitúa a una distancia medida por `center` en la dirección apuntada por la cámara.

```
glm::mat4x4 CameraToeIn::get_view_matrix(Lens lens)
{
    const glm::vec3 lens_position = _position + offset(lens) * _right;
    return glm::lookAt(lens_position, _position + _front * center + _front, _up);
}
```

Código 17 Método `get_view_matrix` de la clase `CameraToeIn`

La matriz de proyección es la misma que la matriz de proyección de la cámara con visión monocular.

La matriz de vista del método *OffsetFrustums* es igual a la de la cámara con *Off-Axis*, mientras que la matriz de proyección es la misma que en la cámara con visión monocular.

6.2.5 Carga de modelos

La carga de modelos se ha basado en las clases proporcionadas en la página web learnopengl.com¹, pero se han modificado para utilizarse versiones anteriores de OpenGL.

El código está dividido en dos clases, la clase *Mesh* que guarda la posición de los vértices y sus características de una parte del modelo y la clase *Model* que carga el modelo usando la librería Assimp y almacena las texturas y el mallado o *mesh* leídos.

Los modelos se organizan en nodos que contienen los diferentes meshes que lo forman. Para cargar un modelo en la clase *Model* se crea un importador de Assimp y utilizarlo para leer el archivo del modelo. Luego se recorren los nodos procesando cada *mesh*. Los *meshes* contienen la información de las posiciones de los vértices, normales, índices, las texturas coordenadas y las texturas utilizadas. Las texturas son guardadas en el modelo para evitar cargar varias veces la misma textura y el resto de datos son guardados en una clase *Mesh* que a su vez es guardada en la clase *Model*.

Para renderizar el modelo se deben renderizar por separado cada *mesh*. En OpenGL moderno y tal como está explicado en Learnopengl se pueden guardar los datos del mesh en la GPU con *buffers* de vértices y posteriormente renderizarlos refiriéndose a ellos mediante un id. Debido a incompatibilidades entre versiones he usado un método anterior en la que se transfieren los datos en cada fotograma.

Primero se habilitan la capacidad de transferencia de posiciones de vértices, normales y coordenadas de texturas con la función *glEnableClientState*, y se activan las texturas necesarias. Luego se transmiten los datos y se dibujan con la función *glDrawElements* que forma triángulos utilizando los índices. Finalmente se deshabilitan las capacidades de transferencia y se desvinculan las texturas.

¹ <https://learnopengl.com/#!Model-Loading/Assimp>

```
void render(Model &model) {
    for (auto& mesh : model.meshes) {
        glEnableClientState(GL_VERTEX_ARRAY);
        glEnableClientState(GL_NORMAL_ARRAY);
        glEnableClientState(GL_TEXTURE_COORD_ARRAY);

        unsigned int numTextures = 0;
        for (auto& texture : mesh.textures)
        {
            glActiveTexture(GL_TEXTURE0 + numTextures);
            glBindTexture(GL_TEXTURE_2D, texture.get_Id());
            numTextures++;
        }

        glTexCoordPointer(2, GL_FLOAT, 0, &mesh.texCoords[0]);
        glVertexPointer(3, GL_FLOAT, 0, &mesh.positions[0]);
        glNormalPointer(GL_FLOAT, 0, &mesh.normals[0]);
        glDrawElements(GL_TRIANGLES, mesh.indices.size(), GL_UNSIGNED_INT,
&mesh.indices[0]);

        glDisableClientState(GL_VERTEX_ARRAY);
        glDisableClientState(GL_NORMAL_ARRAY);
        glDisableClientState(GL_TEXTURE_COORD_ARRAY);

        while (numTextures > 0) {
            numTextures--;
            glActiveTexture(GL_TEXTURE0 + numTextures);
            glBindTexture(GL_TEXTURE_2D, 0);
        }
    }
}
```

Código 18 Método render de renderización de modelos

Assimp carga los modelos de manera más rápida al compilarse con optimizaciones en el modo *Release*, en cambio en el modo *Debug* tarda más tiempo.

6.2.5 Animation

Esta plantilla sirve para crear transiciones, como mover la cámara de un punto a otro, dirigir la vista, mover un objeto. Guarda un *array* de *Frames* o fotogramas claves, que almacenan las posiciones a interpolar y el tiempo que tarda en llegar al siguiente punto. Las transiciones se realizan utilizando la función de *glm::catmullRom* que devuelve interpolación de un punto dentro de una curva. También se puede obtener una interpolación lineal con el método *glm::mix*. La plantilla puede ser parametrizada para utilizar diferentes tipos que sean compatibles con la librería GLM. Por defecto utiliza *glm::vec3*.

6.2.6 Demostrador

El demostrador está encapsulado en una gran clase *SpaceDemo*. Esta clase utiliza las demás para crear una representación del sistema solar y mover al usuario entre los planetas. Estos planetas pueden tener anillos semitransparentes, por lo que deben de renderizarse en un cierto orden según la distancia a la cámara. La cámara se mueve entre los planetas usando la plantilla *Animation*.

Presionando la barra espaciadora la cámara se para y al pulsar de nuevo sigue su camino. A través de las teclas ‘1’, ‘2’, ‘3’ se puede cambiar el tipo de cámara por *Camera3D*, *CameraToeIn* y *CameraOffsetFrustums* y con las teclas ‘o’ y ‘l’ se aumenta y disminuye la distorsión que contrarresta la lente. Con la tecla ‘q’ se cierra la demostración.

7. PRUEBAS Y RESULTADOS

Tras realizar la demostración y probar las diferentes cámaras estereoscópicas con los cascos de realidad virtual he llegado a las siguientes conclusiones. Algunas de las conclusiones pueden ser subjetivas ya que algunas cuestiones como la inmersión o la sensación de tridimensionalidad son experimentadas por el usuario. Hubiera sido conveniente encuestar a un grupo de personas sobre estos temas para obtener una visión más amplia, pero finalmente no fue posible. Los resultados finales son descritos a partir de mi punto de vista, por lo que pueden variar ligeramente para otras personas.

De los tres métodos usados en las cámaras estereoscópicas el más cómodo al ojo es el método *Off-Axis* con frustums asimétricos.

El método *Offset Frustums* con las cámaras separadas y apuntando paralelamente hacia adelante también obtiene buenos resultados, aunque los objetos mostrados parecen estar ligeramente más cerca que en con el método *Off-Axis*. Esto es debido a que los objetos mostrados utilizando este método tendrán la apariencia de situarse delante de la pantalla y no detrás de ella, por lo que la experiencia usando este método depende de los cascos utilizados. En el caso de los cascos utilizados la sensación no es incómoda, pero en otros cascos los resultados podrían ser peores.

El método *Toe-In* es el que peores resultados logra, pues se tiene que forzar la vista para ver el efecto tridimensional. La sensación mejora cuando la pantalla de proyección está más alejada del punto de vista, ya que las diferencias entre las distintas orientaciones son menores y el método se asemeja al *Offset Frustums*.

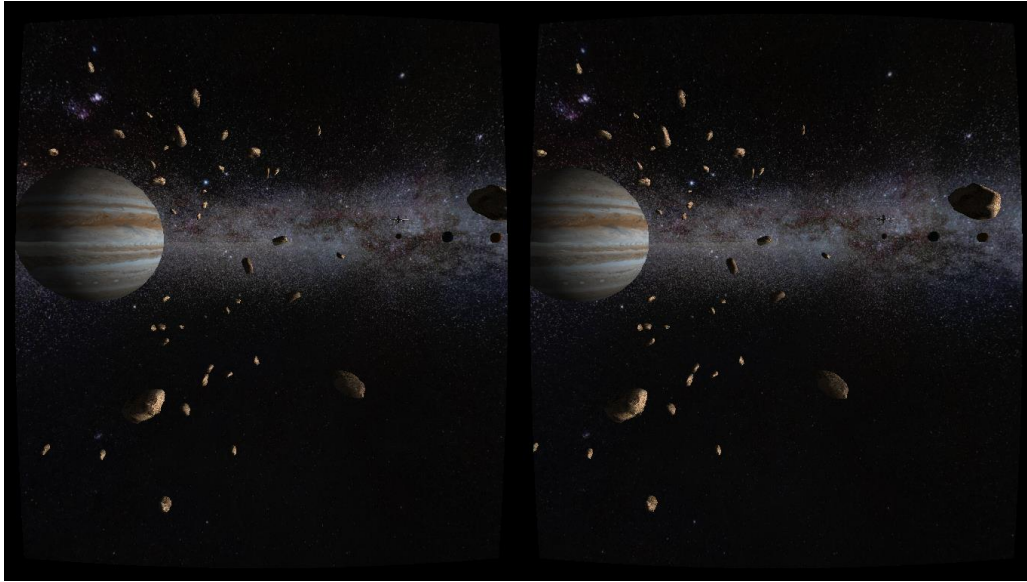


Ilustración 36 Júpiter en estéreo usando el método Off-Axis

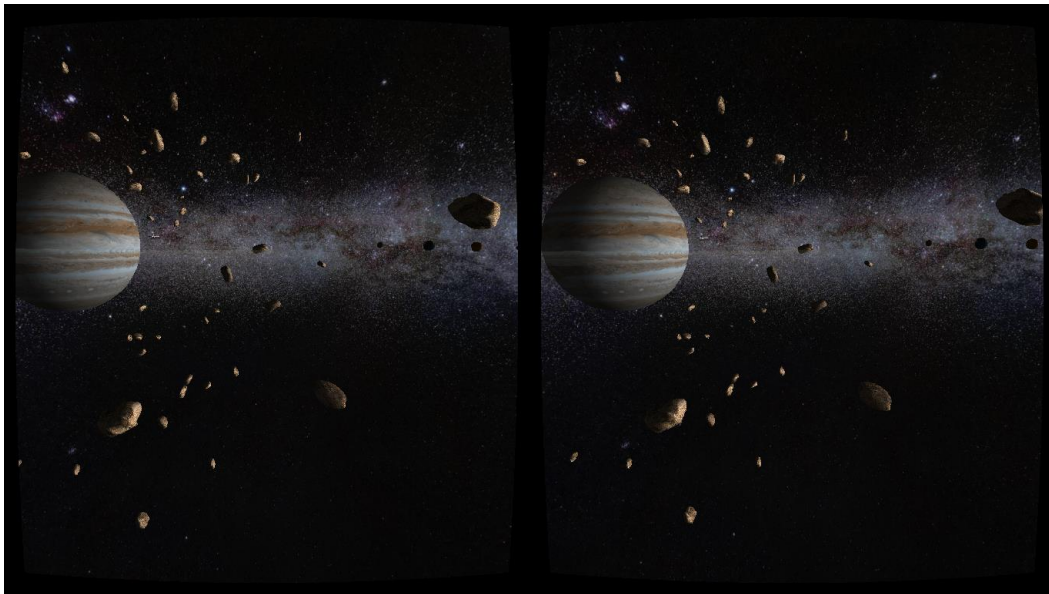


Ilustración 37 Ilustración 35 Júpiter en estéreo usando el método Toe-In

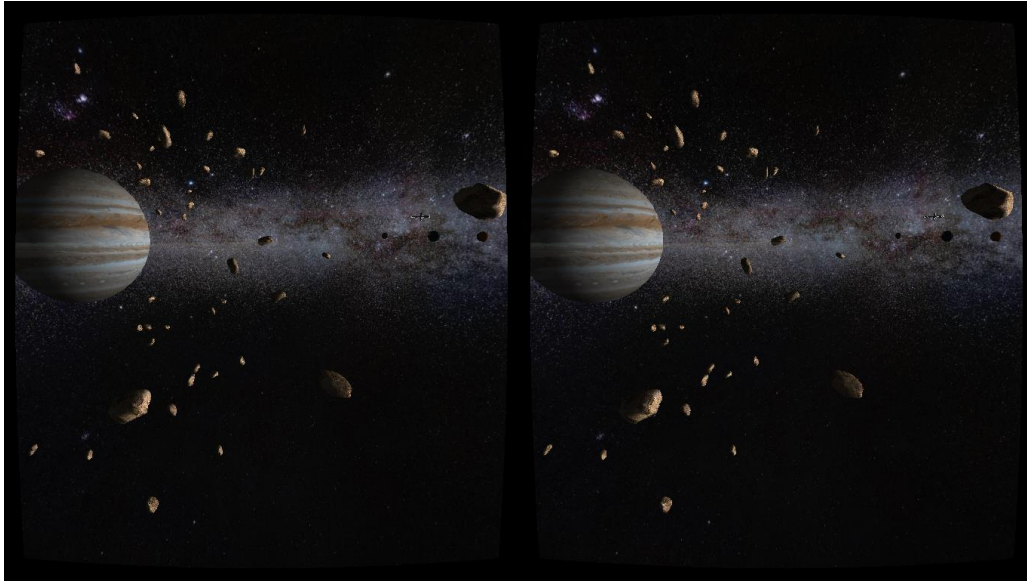


Ilustración 38 Ilustración 35 Júpiter en estéreo usando el método Offset Frustums

A pesar de las diferencias los tres métodos logran que el usuario perciba un efecto tridimensional. Este efecto puede desaparecer si las cámaras están mal configuradas.

El campo de vista de la cámara no afecta notablemente al efecto tridimensional, excepto en los casos extremos en donde supere en gran medida al campo de vista del casco. Por el contrario, sí afecta a la inmersión, pues la amplitud del campo de vista puede no ser realista. Se debe evitar los cambios en el campo de vista y en general en todos los aspectos de la cámara, ya que el ojo se acostumbra a ellos y cambiarlos aumenta la sensación de vista cansada.

La distancia de la pantalla de proyección debe equiparar a la distancia percibida en la pantalla del casco. En distancias muy cortas hay que forzar la vista y en distancias largas, objetos que están en el infinito, como el cielo, parecen estar más cerca.

La separación de las lentes es el aspecto que más influye en la sensación de profundidad. Cuanto mayor es la separación, más se aprecia la profundidad de los objetos hasta que la separación es tal que las imágenes no convergen en una sola y el usuario ve doble. Esta separación crítica es diferente para cada método, siendo el método *Off-Axis* el que mayor separación soporta. En el método *Toe-*

In la separación aumenta la diferencia de los planos proyectores y en el *Offset Frustum* el punto donde ambos frustums se juntan se sitúa más lejos separando en mayor medida las imágenes.

Los objetos con texturas detalladas hacen que se distinga mejor la profundidad. Es aconsejable que el fondo también tenga textura y no esté compuesto por un solo color, para que el usuario a través de los detalles sepa dónde está el infinito y le ayude a situar los demás objetos de la escena. También es de ayuda como punto de referencia que este a la vista. Se deben evitar los movimientos de cámara extraños que desorienten al usuario.

Cuantos más objetos haya en la escena, más fácil será para el usuario percibir la sensación de profundidad, pues tendrá más puntos de referencia para comparar su profundidad. Si uno de estos objetos no se ve con una perspectiva coherente con el resto de la escena, el usuario intentará enfocarlos, desenfocando el resto de la escena, lo que lleva a la pérdida del efecto tridimensional.

La demostración se ejecuta en el ordenador con una tasa de entre 40 y 60 FPS. Sin embargo, la pantalla es compartida a través de la aplicación SpaceDesk Beta, por lo que los fotogramas por segundo son mucho menores y dependen de la velocidad de la red.

En las experiencias de realidad virtual pequeños retardos pueden marear al usuario por el desfase entre las acciones del usuario, como mover la cabeza y su efecto en el entorno virtual. En el método utilizado, en el que la aplicación capta el movimiento, es enviado al ordenador donde se realiza el procesamiento y posteriormente se transmite al teléfono a usando la aplicación SpaceDesk, el retardo es considerable, debido a las transmisiones de datos por internet. El retardo es tan grande que el usuario puede notar los movimientos de cabeza y sus efectos en la realidad virtual como actos diferentes, reduciendo la inmersión, pero disminuyendo la sensación de mareo.

8. CONCLUSIONES

La realidad virtual es una tecnología que debe ser calibrada para obtener los mejores resultados y prevenir malestares en el usuario y pérdidas de inmersión. Las experiencias de menor coste como los cascos de realidad virtual que utilizan un dispositivo móvil pueden funcionar si no son extensas, pero en experiencias de larga duración es preferible el uso de cascos de realidad virtual más avanzados y de programas de realidad virtual que usen correctamente las técnicas de estereoscopia.

El uso de herramientas de bajo nivel como OpenGL o DirectX pueden ayudar a lidiar con los requisitos gráficos y de rendimiento que son necesarios en las experiencias de realidad virtual de muchos cascos. Las técnicas de estereoscopia pueden aplicarse en cualquier herramienta, y aunque algunos cascos de realidad virtual hayan creado librerías que ayudan al desarrollador a crear experiencias de realidad virtual, siempre es de gran ayuda conocer los principios que hay detrás de estas técnicas.

En el ámbito personal me ha encantado realizar este trabajo y poder experimentar con la realidad virtual. Tenía mucha curiosidad sobre este tema, como funcionaba la visión estereoscópica y he podido probar muchas de las técnicas que he encontrado en internet. La aplicación desarrollada puede ser la base de otros proyectos que tengo en mente y muchas de las clases que he creado pueden ser reutilizadas en futuros proyectos. También espero que este TFG sirva de ayuda a aquellos que quieran adentrarse en este mundo y que puedan disfrutar tanto de la realidad virtual como yo lo he hecho en estos últimos meses.

9. ANEXO

9.1 Instalación y configuración del entorno

El proyecto ya viene configurado para ser compilado y ejecutado en Visual Studio 2017. En caso de querer crear un nuevo proyecto de OpenGL que utilice las clases desarrolladas en este TFG se debe crear un nuevo proyecto vacío de C++.

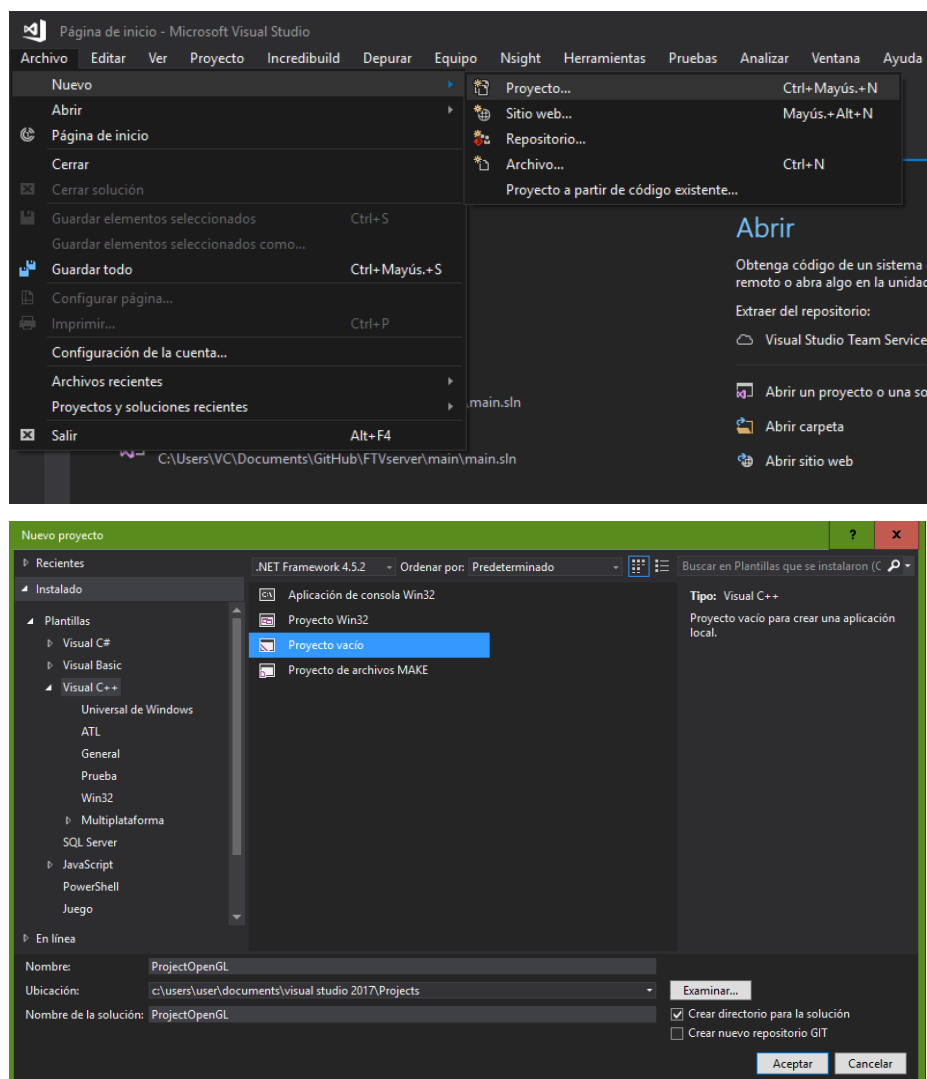


Ilustración 39 Creación de un nuevo proyecto vacío de C++ en Visual Studio

Las configuraciones pueden aplicarse simplemente añadiendo la hoja de propiedades *libraries.props* desde el administrador de propiedades. Este panel puede accederse desde *Ver/Otras ventanas/ Administrador de propiedades (View/Other Windows/Property Manager)*.

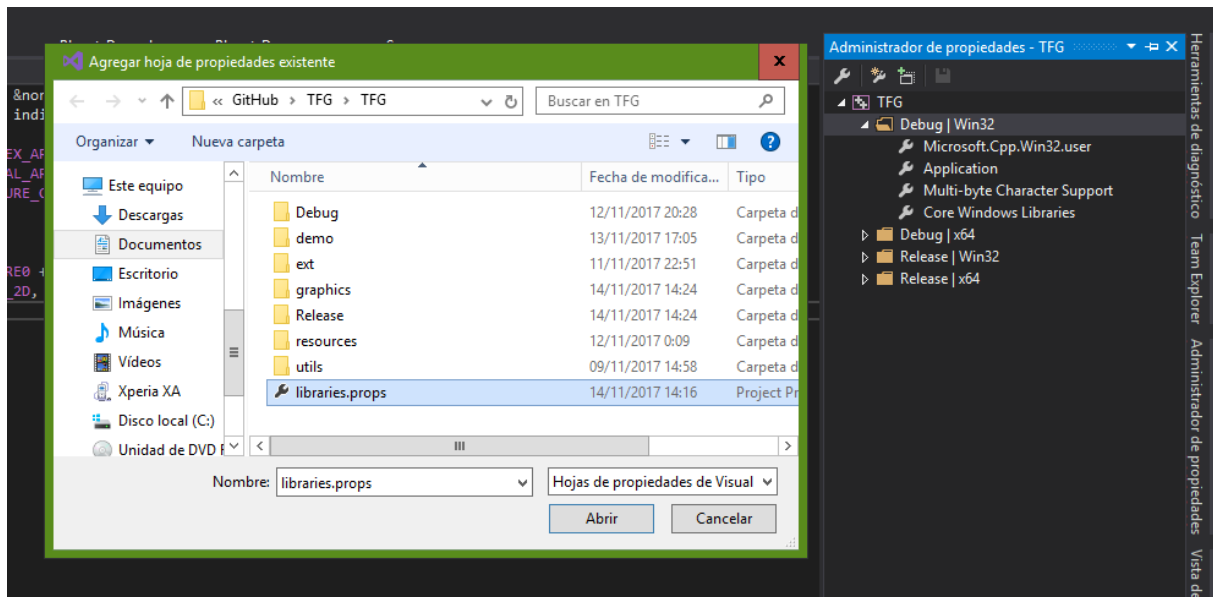


Ilustración 40 Configuración de propiedades del proyecto con una hoja de propiedades

Realizando estos pasos el proyecto solo es necesario añadir las carpetas *graphics*, *utils* y *resources* con las clases reutilizables que se deseen utilizar.

En caso de que no se pueda aplicar la hoja de propiedades, se pueden asignar la configuración manualmente. Se deben de incluir las librerías que se van a utilizar, que se encuentran en la carpeta *ext* del proyecto. Dentro de esta carpeta se encuentran las cabeceras en la carpeta *include*, las librerías en la carpeta *lib* y las bibliotecas de enlace dinámico o dll en la carpeta *bin*.

Los archivos dll de la carpeta *bin* deben ser copiados a las carpetas donde se cree el ejecutable, generalmente *Debug* o *Release* o se producirá un error en ejecución.

Accediendo al menú de *Proyecto (Project)*, y haciendo clic en *Propiedades del proyecto (Properties)* se abre una ventana donde cambiar las propiedades del proyecto para una determinada configuración, *Debug* o *Release*.

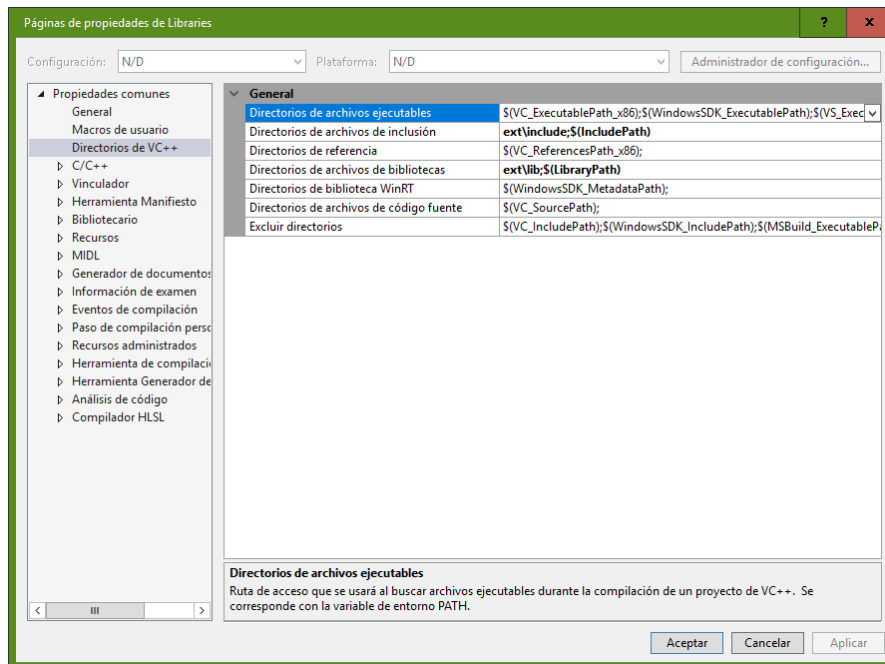


Ilustración 41 Configuración de directorios de incusión y bibliotecas

En la pestaña *Directorios de VC++ (VC Directories)* se debe especificar la carpeta de cabeceras en el campo *Directorios de inclusión (Include Directories)* y en el campo *Directorios de archivos de bibliotecas (Library Directories)* la carpeta de librerías.

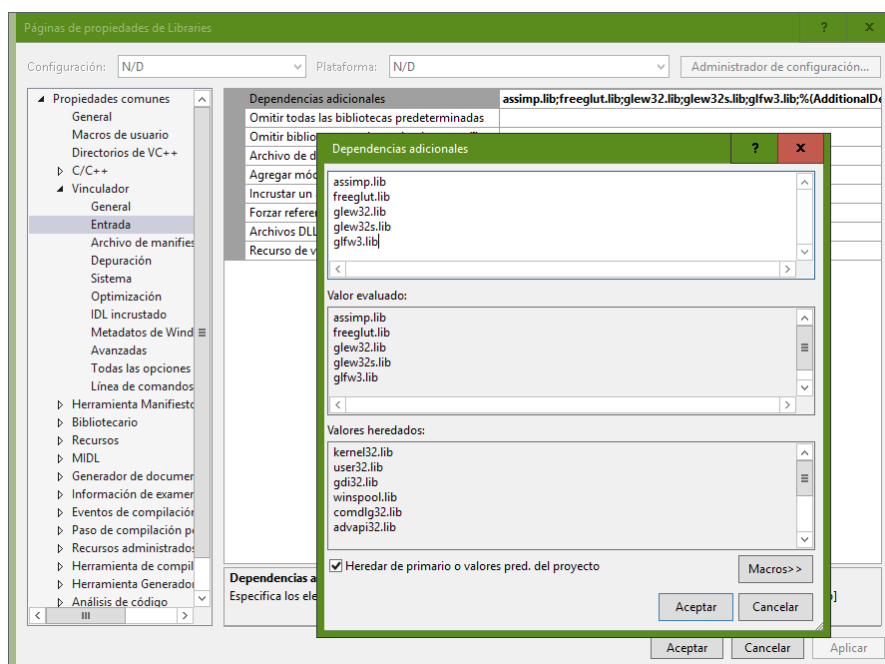


Ilustración 42 Configuración de librerías usadas por el vinculador

Además, en la pestaña de *Vinculador (Linker)*, *Entrada(Input)*, en el campo de *dependencias adicionales (Executable Directories)* se debe listar todas las librerías que se desean usar, en este caso son.

- `assimp.lib`
- `freeglut.lib`
- `glew32.lib`
- `glew32s.lib`
- `glfw3.lib`

Para compilar el código con otros compiladores se deben igualmente linquear las librerías usadas. Para sistemas operativos diferentes a Windows se deben buscar las versiones de las librerías apropiadas para el sistema operativo. Todas cuentan con versiones para otros sistemas operativos y las librerías GLM y *stb_image* son multiplataforma al solo contar con cabeceras. Las clases *FullScreen* y *NetReceiver* implementadas solo funcionan en Windows.

Para proyectar la imagen estereoscópica generada en el ordenador debe estar instalado el driver de SpaceDesk y en el dispositivo móvil la aplicación de SpaceDesk. En caso de que se quiera recibir los datos de la orientación del dispositivo también debe instalarse la aplicación *Sensor Transmitter*.

Primero se debe iniciar la aplicación *Sensor Transmitter*, iniciar el servidor usando el puerto deseado e iniciar la aplicación de SpaceDesk y conectar con el ordenador. Si el driver del ordenador esta activado se compartirá automáticamente la pantalla.

En caso de que no se reciban datos de SpaceDesk o la aplicación *Sensor Transmitter* se debe revisar que el ordenador y el dispositivo se encuentran en la misma red y que el cortafuegos permite la conexión.

9.2 Tutorial de uso

Para explicar cómo utilizar la visión estereoscópica que proporcionan las clases del TFG partiremos del ejemplo básico del triángulo de OpenGL del código 19 que mostrará como resultado la imagen 43.

```
#include <GL/glew.h>
#include <stdio.h>
#include <GL/glew.h>
#include <GL/freeglut.h>

void display() {

    glClearColor(0, 0, 0.5, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glBegin(GL_TRIANGLES);

    glColor3f(1, 0, 0);
    glVertex3f(-1.0f, -0.5f, 0.0f);
    glVertex3f(1.0f, -0.5f, 0.0f);
    glVertex3f(0.0f, 0.5f, 0.0f);

    glEnd();

    glFlush();
    glutSwapBuffers();

}

int main(int argc, char *argv[]) {

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);

    glutCreateWindow("Test");

    glewInit();

    glutDisplayFunc(display);

    glutMainLoop();

}
```

Código 19 Triángulo en OpenGL usando GLUT

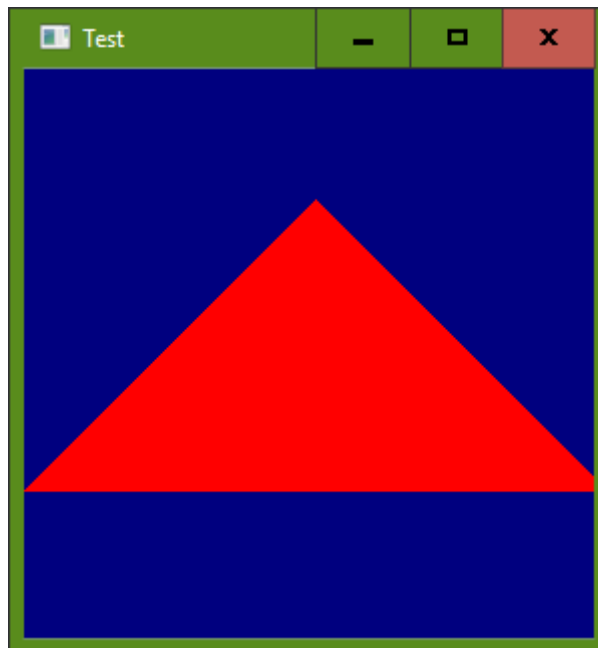


Ilustración 43 Triángulo dibujado por OpenGL

Para convertir esta imagen en una imagen estereoscópica que pueda verse a través del móvil utilizando SpaceDesk debemos usar las clases *StereoVision*, *Camera3D* y *FullScreen* como se ve en el código 20, además de tener activado la pantalla remota de SpaceDesk.

En la función *main* después de crear la ventana, esta se debe de poner en modo pantalla completa en la pantalla remota de SpaceDesk usando la función *FullScreen::set_on_screen*. El primer parámetro de la función indica el índice de la pantalla. Las pantallas se numeran en orden, siendo 0 la principal, 1 la secundaria... etc. Como la pantalla de SpaceDesk es la última en conectarse suele tener el mayor índice.

Después se procede a configurar la cámara 3D, especificando la separación de los puntos de vista del estéreo con *lens_separation*, la distancia dónde se sitúa la pantalla virtual con *set_center* y otras configuraciones más usuales como *focus_point*, que apunta la cámara a un cierto punto.

La función *glutDisplayFunc* ha sido reemplazada por *StereoVision::display_func* para usar la visión en estéreo. La función *display* ahora debe de tener un argumento booleano que es verdadero cuando se renderiza la parte derecha del estéreo y falsa cuando es la izquierda.

Al comienzo de la función se obtienen las matrices de proyección y vista de la cámara necesarias para aplicar la perspectiva de la cámara. Las matrices se devuelven usando la clase *glm::mat4* de la librería GLM, que contiene la matriz en un array de 16 elementos.

Para aplicar la matriz se cambia el modo de matriz de OpenGL y se cargan las matrices. Finalmente se renderiza el triángulo.

Finalmente en la función *main* se utiliza la función *StereoVision::after_display* y se indica que se llame a *glutPostRedisplay* para que se vuelva a dibujar el triángulo después de haberse dibujado.

```
#include <GL/glew.h>
#include "graphics\Camera.h"
#include "graphics\StereoVision.h"
#include "utils\FullScreen.h"

#include <stdio.h>
#include <GL/glew.h>
#include <GL/freeglut.h>

Camera3D camera;

void display(bool isRight) {

    Camera3D::Lens lens = camera.lens(isRight);
    glm::mat4 projection = camera.get_projection_matrix(lens);
    glm::mat4 view = camera.get_view_matrix(lens);

    glClearColor(0, 0, 0.5, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_PROJECTION);
    glLoadMatrixf((float*)&projection[0]);

    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf((float*)&view[0]);

    glBegin(GL_TRIANGLES);

    glColor3f(1, 0, 0);
    glVertex3f(-1.0f, -0.5f, 0.0f);
    glVertex3f(1.0f, -0.5f, 0.0f);
    glVertex3f(0.0f, 0.5f, 0.0f);

    glEnd();

}
```

```
int main(int argc, char *argv[]) {

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);

    glutCreateWindow("Test");
    FullScreen::set_on_screen(1, "Test");

    glewInit();

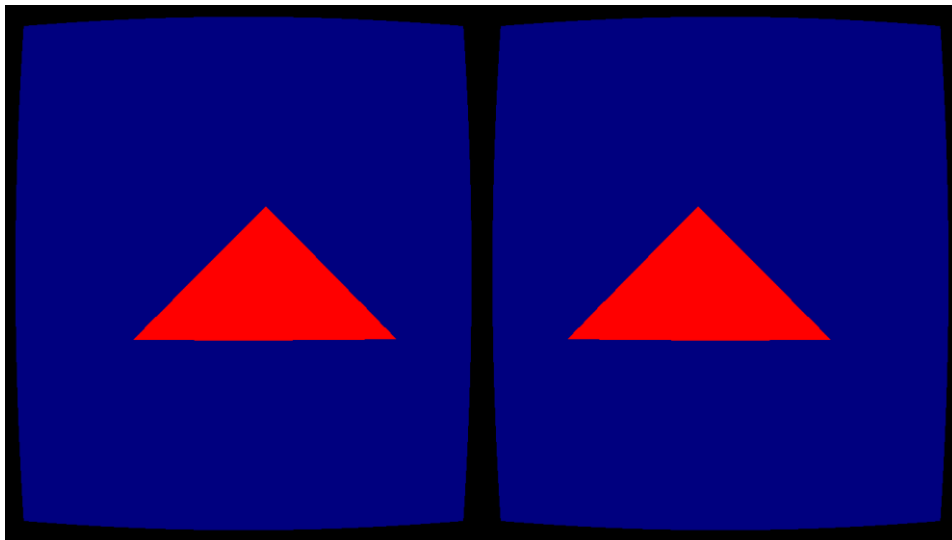
    camera.lens_separation(0.2f);
    camera.field_of_view(glm::radians(90.f));
    camera.set_center(3);
    camera.position({ 0, 0, -2 });
    camera.focus_point({ 0, 0, 0 });
    camera.screen_dimensions({ glutGet(GLUT_WINDOW_WIDTH) / 2.f,
    glutGet(GLUT_WINDOW_HEIGHT) });

    StereoVision::display_func(display, glutGet(GLUT_WINDOW_WIDTH),
    glutGet(GLUT_WINDOW_HEIGHT));
    StereoVision::after_display(glutPostRedisplay);

    glutMainLoop();

}
```

Código 20 Triángulo en estéreo en OpenGL



Código 21 Triángulo en estéreo

La clase *NetReceiver* permite obtener la orientación del dispositivo enviado a través de la red por la aplicación *Sensor Transmitter*. El código 22 es el anteriormente usado modificado para que la cámara se mueva según los movimientos del dispositivo. Para que los movimientos del dispositivo muevan

la cámara debemos crear una instancia de *NetReceiver* con la ip del dispositivo y el puerto indicado en la aplicación. En la función *main* debemos iniciar el receptor. Utilizando la función *StereoVision::before_display* hacemos que se llame antes de cada renderización la función *before_render* de nuestro código, donde obtendremos la última orientación transmitida y aplicaremos su rotación a la cámara.

```
#include <GL/glew.h>
#include "graphics\Camera.h"
#include "graphics\StereoVision.h"
#include "utils\FullScreen.h"
#include "utils\NetReceiver.h"

#include <stdio.h>
#include <GL/glew.h>
#include <GL/freeglut.h>

Camera3D camera;
NetReceiver receiver("192.168.1.100", 27000);

void display(bool isRight) {

    Camera3D::Lens lens = camera.lens(isRight);
    glm::mat4 projection = camera.get_projection_matrix(lens);
    glm::mat4 view = camera.get_view_matrix(lens);

    glClearColor(0, 0, 0.5, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_PROJECTION);
    glLoadMatrixf((float*)&projection[0]);

    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf((float*)&view[0]);

    glBegin(GL_TRIANGLES);

    glColor3f(1, 0, 0);
    glVertex3f(-1.0f, -0.5f, 0.0f);
    glVertex3f(1.0f, -0.5f, 0.0f);
    glVertex3f(0.0f, 0.5f, 0.0f);

    glEnd();

}

void before_display() {
    NetReceiver::Orientation orientation = receiver.get_orientation();
    camera.focus_point({ 0, 0, 0 });
    camera.front(camera.front() + glm::vec3{ 0, sin(orientation.z), 0 });
    camera.rotate(orientation.y);
}
```

```
}  
  
int main(int argc, char *argv[]) {  
  
    glutInit(&argc, argv);  
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);  
  
    glutCreateWindow("Test");  
    FullScreen::set_on_screen(2, "Test");  
  
    receiver.start();  
  
    glewInit();  
  
    camera.lens_separation(0.2f);  
    camera.field_of_view(glm::radians(90.f));  
    camera.set_center(3);  
    camera.position({ 0, 0, -2 });  
    camera.focus_point({ 0, 0, 0 });  
    camera.screen_dimensions({ glutGet(GLUT_WINDOW_WIDTH) / 2.f,  
glutGet(GLUT_WINDOW_HEIGHT) });  
  
    StereoVision::display_func(display, glutGet(GLUT_WINDOW_WIDTH),  
glutGet(GLUT_WINDOW_HEIGHT));  
    StereoVision::after_display( glutPostRedisplay );  
  
    StereoVision::before_display(before_display);  
  
    glutMainLoop();  
}
```

Código 22 Triángulo en OpenGL en estéreo y usando NetReceiver

Por último, en el código 23 cambiamos el triángulo por un modelo que carga la clase *Model* en la función *main*. La ruta al modelo es indicada en el constructor.

Posteriormente inicializamos una animación que moverá el modelo verticalmente como si estuviese flotando utilizando la plantilla *Animation*. Esta plantilla interpola entre un conjunto puntos dados. El parámetro de la plantilla indica el tipo que se va a interpolar, por defecto un *glm::vec3*.

Primero se necesita crear una lista con los fotogramas claves de la animación, los puntos por donde pasará el modelo, en este caso el punto superior y el inferior. Los fotogramas claves del vector pueden especificarse usando *aggregate initialization*, con {}, especificando primero el valor del punto y luego el tiempo que transcurre hasta moverse al siguiente. Finalmente se inicia la animación con el método *start*.

En la función *display* a través del método de *Model render* se renderiza el modelo en la escena. Previamente debemos transformar el modelo, pues por defecto viene tumbado y es demasiado grande. A través de la matriz de modelo podemos aplicar estas transformaciones y además aplicar la traslación realizada por la animación. La posición de la animación se obtiene a través del método *get_frame*.

Las funciones de transformación de GLM tienen la misma funcionalidad que sus análogas de OpenGL pero se aplican sobre la matriz indicada. El resultado final es el de la imagen 44.

```
#include <GL/glew.h>
#include "graphics\Camera.h"
#include "graphics\Model.h"
#include "graphics\StereoVision.h"
#include "utils\FullScreen.h"
#include "utils\NetReceiver.h"
#include "utils\Animation.h"

#include <stdio.h>
#include <GL\glew.h>
#include <GL\freeglut.h>

Camera3D camera;
NetReceiver receiver("192.168.1.100", 27000);
Model* model;
Animation<> animation;

void display(bool isRight) {

    Camera3D::Lens lens = camera.lens(isRight);
    glm::mat4 projection = camera.get_projection_matrix(lens);
    glm::mat4 view = camera.get_view_matrix(lens);

    glClearColor(0, 0, 0.5, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_PROJECTION);
    glLoadMatrixf((float*)&projection[0]);

    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf((float*)&view[0]);

    float time = glutGet(GLUT_ELAPSED_TIME) / 1000.f;
```

```

    glm::mat4 modelMatrix = glm::translate(glm::mat4(),
animation.get_frame(time).point);
    modelMatrix = glm::rotate(modelMatrix, -glm::half_pi<float>(), glm::vec3(1,0,0));
    modelMatrix = glm::rotate(modelMatrix, glm::pi<float>(), glm::vec3(0, 0, 1));
    modelMatrix = glm::scale(modelMatrix, glm::vec3(0.1f));

    glMultMatrixf((float*)&modelMatrix[0]);

    model->render();
}

void before_display() {
    NetReceiver::Orientation orientation = receiver.get_orientation();
    camera.focus_point({ 0, 0, 0 });
    camera.front(camera.front() + glm::vec3{ 0, sin(orientation.z), 0 });
    camera.rotate(orientation.y);
}

int main(int argc, char *argv[]) {

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);

    glutCreateWindow("Test");
    FullScreen::set_on_screen(2, "Test");

    receiver.start();

    glewInit();
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    camera.lens_separation(0.2f);
    camera.field_of_view(glm::radians(90.f));
    camera.set_center(3);
    camera.position({ 0, 0, -2 });
    camera.focus_point({ 0, 0, 0 });
    camera.screen_dimensions({ glutGet(GLUT_WINDOW_WIDTH) / 2.f,
glutGet(GLUT_WINDOW_HEIGHT) });

    model = new Model("resources/models/Nanosuit/nanosuit2.3ds");

    float time = glutGet(GLUT_ELAPSED_TIME) / 1000.f;

    std::vector<Animation<>::Frame> frames = { {{ 0, 0, 0 }, 2},{ { 0, -2, 0 }, 2 } };
    animation.set_frames(frames);
    animation.start(time);

    StereoVision::display_func(display, glutGet(GLUT_WINDOW_WIDTH),
glutGet(GLUT_WINDOW_HEIGHT));
    StereoVision::after_display( glutPostRedisplay );

    StereoVision::before_display(before_display);

```

```
glutMainLoop();  
  
delete model;  
}
```

Código 23 Renderización de modelo en estéreo con animación

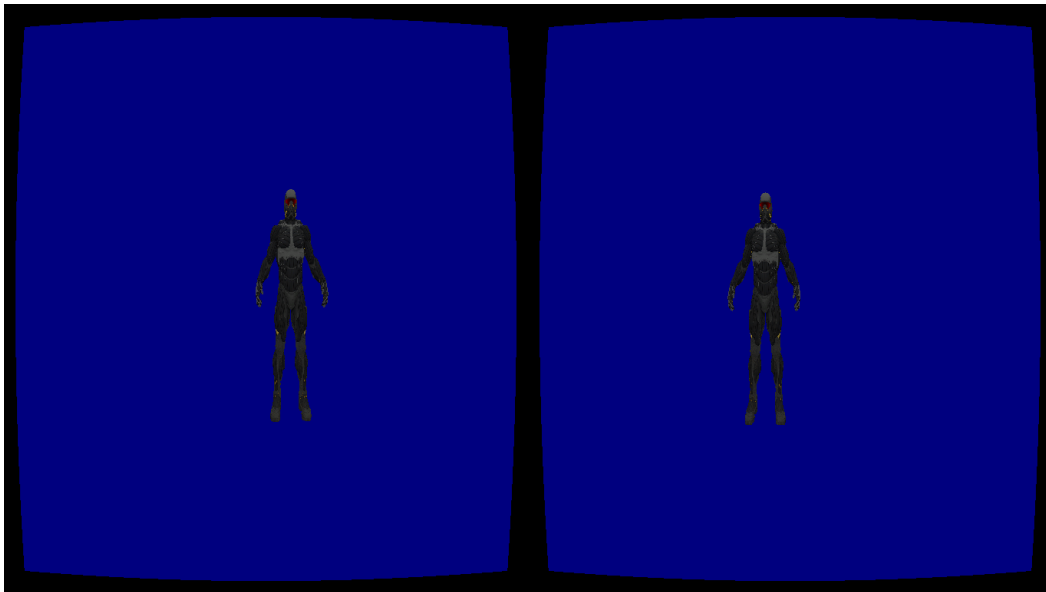


Ilustración 44 Modelo renderizado en estéreo

9. BIBLIOGRAFÍA

- [1] **Vozpopuli** De la realidad virtual a la realidad española: las startups que lideran el sector
http://www.vozpopuli.com/economia-y-finanzas/startups/espana-realidad-virtual-sectores-startups_0_1032797194.html
- [2] **Startup Xplore** l negocio de la realidad virtual
<https://startupxplore.com/es/blog/realidad-virtual/>
- [3] **NovaVision Blog** Stereoscopic Vision – How Does It Work
<http://blog.novavision.com/stereoscopic-vision-how-does-it-work>
- [4] **Documento de cátedra de la Universidad Nacional de San Juan** Vision Estereoscopica
<ftp://ftp.unsj.edu.ar/agrimensura/Fotogrametria/Unidad4/VISION-ESTEREO.pdf>
- [5] **Nvidia** Implementing Stereoscopic 3D in Your Application
https://www.nvidia.com/content/GTC-2010/pdfs/2010_GTC2010.pdf
- [6] **Autor: Paul Bourke** Calculating Stereo Pairs
<http://paulbourke.net/stereographics/stereorender/>
- [7] **Autor: Raymond C. H. Lo** Rendering Stereoscopic 3D Models using OpenGL
<https://www.packtpub.com/books/content/rendering-stereoscopic-3d-models-using-opengl>
- [8] **LearnOpenGL** Autor: Joey de Vries
<https://learnopengl.com>
- [9] **Doc-Ok.org** Good stereo vs. bad stereo
<http://doc-ok.org/?p=77>
- [10] **Ocularis** La visión tridimensional
<https://ocularis.es/la-vision-tridimensional/>
- [11] **Autor: Alfredo Flores** Campo Visual
<https://es.slideshare.net/alfredoDG/campo-visual-14689484>
- [12] **IGN** Autor: José L. Ortega Un repaso a la historia de la realidad virtual
<http://es.ign.com/realidad-virtual/109691/feature/un-repaso-a-la-historia-de-la-realidad-virtual>
- [13] **Wikipedia** Autoestereoscopia
<https://es.wikipedia.org/wiki/Autoestereoscopia>

-
- [14] **Autor Antonio Leiva** MVP for Android: how to organize the presentation layer
<https://antonioleiva.com/mvp-android/>
- [15] **MSDN** EnumDisplayMonitors function
[https://msdn.microsoft.com/en-us/library/windows/desktop/dd162610\(v=vs.85\).aspx?f=255&MSPPErr=-2147217396](https://msdn.microsoft.com/en-us/library/windows/desktop/dd162610(v=vs.85).aspx?f=255&MSPPErr=-2147217396)
- [16] **MSDN** Complete Winsock Client Code
[https://msdn.microsoft.com/en-us/library/windows/desktop/ms737591\(v=vs.85\).aspx?f=255&MSPPErr=-2147217396](https://msdn.microsoft.com/en-us/library/windows/desktop/ms737591(v=vs.85).aspx?f=255&MSPPErr=-2147217396)
- [17] **Binary Tides** Winsock tutorial – Socket programming in C on Windows
<http://www.binarytides.com/winsock-socket-programming-tutorial/>
- [18] **Repositorio OSVR en Github** Shader de distorsión
<https://github.com/OSVR/distortionizer/blob/master/vizard/ShaderTest.frag>

9.1 Imágenes y otros recursos

- [19] Ilustración 2 Visión estereoscópica
<http://www.tech-faq.com/stereoscopic-vision.html>
- [20] Ilustración 3 Campo visual
<https://es.slideshare.net/alfredoDG/campo-visual-14689484>
- [21] Ilustración 4 Frustum
https://www.cosc.brocku.ca/Offerings/3P98/course/lectures/3d_perspective/
- [22] Ilustración 5 Frustums en Toe-in e Ilustración 6 Frustums en Off-Axis
<http://paulbourke.net/stereographics/stereorender/>
- [23] Ilustración 8 Ejemplo de la razón de uso de lentes en los cascos de realidad virtual
<http://doc-ok.org/?p=1360>
- [24] Ilustración 9 Distorsión generada por las lentes en los cascos Oculus Rift DK2
<http://doc-ok.org/?p=1414>
- [25] Ilustración 10 Gafas anaglifo
https://es.wikipedia.org/wiki/Gafas_anaglifo

[26] Ilustración 11 Oculus Rift Go

<https://www.oculus.com/blog/pioneering-the-frontier-of-vr-introducing-oculus-go-plus-santa-cruz-updates/>

[27] Ilustración 12 Google Glass

https://es.wikipedia.org/wiki/Google_Glass

[28] Ilustración 13 Hololens

<http://time.com/4240446/microsoft-hololens-release-date-2016/>

[29] Ilustración 14 PlayStation VR

<https://www.playstation.com/es-es/explore/playstation-vr/>

[30] Ilustración 15 HTC Vive

<https://www.playscope.com/htc-vive/htc-vive-images/>

[31] Ilustración 21 Ejes de rotación

https://es.wikipedia.org/wiki/Ejes_del_avión

[32] Ilustración 23 Logo de OpenGL

<https://www.opengl.org>

[33] Ilustración 24 Pipeline de OpenGL

https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

[34] Ilustración 25 Coordenadas *NDC* de OpenGL e Ilustración 26 Transformaciones de OpenGL

<https://learnopengl.com/#!Getting-started/Coordinate-Systems>

[35] Modelo Halcón Milenario

<https://free3d.com/3d-model/millennium-falcon-82947.html>

[36] Modelo Nanosuit 2

<https://free3d.com/3d-model/crysis-2-nanosuit-2-97837.html>

[37] Modelo Asteroide

<https://learnopengl.com/data/models/rock.rar>

[38] Texturas del sistema solar

a. <https://github.com/MattLoftus/threejs-space-simulations/tree/master/images>

