

MÁSTER UNIVERSITARIO EN COMPUTACIÓN GRÁFICA Y SIMULACIÓN

2017/2018

Trabajo de Final de Máster

Investigación, evaluación e implementación de
métodos que simulen seis grados de libertad
en fotos y vídeo para Realidad Virtual

Autor: Gregorio Iniesta Ovejero
Tutor: Diego Bezares Sánchez

Índice general

1. Resumen	1
2. Introducción	3
2.1. Grados de libertad	3
2.2. Foto y vídeo en realidad virtual	4
2.3. Qué aporta este proyecto	5
3. Planteamiento del problema	7
4. Objetivos	9
5. Estado del Arte	11
5.1. Marco teórico	11
5.1.1. Fotografía y vídeo	11
5.1.2. Mapas de profundidad	19
5.1.3. Fotogrametría	21
5.2. Project Sidewinder	22
5.3. Nube de puntos	22
5.4. Cámara y herramientas de realidad virtual de OTOY y Facebook	23
5.5. Facebook: Realidad virtual y Fotogrametría	24
5.6. "Welcome to Lightfields"	26
6. Desarrollo	29
6.1. El entorno de desarrollo	30
6.1.1. Shaders Gráficos	30
6.1.2. Shaders de cómputo	31
6.2. Metodología	31
6.3. Generación de imágenes con las que trabajar	32
6.4. Primera aproximación: Paralaje sencillo	35
6.4.1. Qué es el paralaje	36
6.4.2. Implementación	36
6.4.3. Resultado	38

6.5.	Segunda aproximación: Paralaje sencillo	38
6.5.1.	Implementación	39
6.5.2.	Resultado	41
6.6.	Tercera aproximación: Paralaje sencillo con <i>shaders</i> de cómputo	42
6.6.1.	Implementación	42
6.6.2.	Resultado	44
6.7.	Cuarta aproximación: Resolviendo la condición de carrera	45
6.7.1.	Implementación	45
6.7.2.	Resultado	47
6.8.	Quinta aproximación: Resolviendo la condición de carrera 2	48
6.8.1.	Resultado	50
6.9.	Aproximación final: Pasando a espacio de mundo	50
6.9.1.	Implementación	51
6.9.2.	Resultado	54
6.10.	Otras pruebas	55
6.10.1.	Evitar borrar el <i>buffer</i> de color	55
6.10.2.	Desplazamiento en espacio de mundo con <i>shaders</i> gráficos	57
6.10.3.	Completar con el ojo opuesto	58
7.	Conclusiones	59
Bibliografía		63

1. Resumen

El vídeo y la fotografía es, hoy en día, una parte imprescindible del entretenimiento en realidad virtual, pero tiene grandes desventajas frente a otras tecnologías, como las experiencias en entornos tridimensionales, debido a la falta de interactividad.

Desde hace unos años empresas como Disney, Facebook y Google han establecido un precedente en la industria tecnológica, buscando establecer los llamados seis grados de libertad al vídeo y la fotografía 360º para mejorar la sensación de inmersión. Existen numerosas maneras de conseguir este objetivo y cada una tiene sus ventajas y sus inconvenientes.

El propósito del trabajo descrito en esta memoria es implementar, de la manera más realista y eficiente posible, un sistema que proporcione seis grados de libertad en un rango de movimiento limitado. Para ello se va a recurrir a un algoritmo basado en desplazamientos pixel a pixel, teniendo capacidad de manipulación de la información al nivel mas detallado disponible, haciendo uso de todos los recursos disponibles en los dispositivos incluyendo hardware gráfico con propósito general.

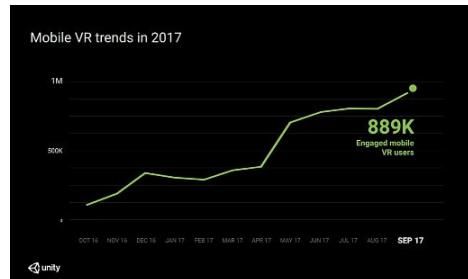
La forma de conseguir este objetivo será evaluar muchas implementaciones para, más tarde, seleccionar los mejores resultados y trabajar sobre ellos con la finalidad de hacerlos más eficientes y realistas. Todo esto sin dejar de contemplar más opciones como las que han implementado otros desarrolladores.

2. Introducción

La realidad virtual es una tecnología que desde hace unos años ha estado intentando hacerse un hueco en la industria del entretenimiento. Unity expuso estadísticas de consumo de realidad virtual durante el pasado año, proporcionando cifras como un 134 % de crecimiento en sobremesa y consola y un 295 % de crecimiento en las instalaciones de realidad virtual en móvil (2.1a), lo que hace casi un millón de usuarios activos en aplicaciones hechas con Unity a finales de 2017 (2.1b). Por último las estadísticas de uso en multimedia superan el 50 % del tiempo de uso llegando a más del 80 % según Facebook [10].



(a) Crecimiento en las instalaciones de realidad virtual.



(b) Usuarios activos en RV para móvil en Unity a finales de 2017.

Figura 2.1: Imágenes obtenidas de la Unite de Unity de octubre de 2017

2.1. Grados de libertad

Un factor clave a tener en cuenta cuando hablamos de realidad virtual son los grados de libertad, que definen la capacidad de movimiento a la hora

de interactuar. Generalmente se habla de tres y seis grados de libertad. En el caso de tres grados de libertad, hace referencia a los giros sobre el eje principal (viraje o *yaw*, inclinación o *pitch* y cabeceo o *roll*), mientras que cuando se amplia a seis grados de libertad, hace referencia al desplazamiento en los tres ejes. Esto, aplicado a las gafas de realidad virtual, informa de la capacidad del dispositivo de captar los giros de la cabeza en todas las direcciones (tres grados de libertad) o si además capta el desplazamiento por la sala (seis grados de libertad).

El mercado de dispositivos de realidad virtual se está enfocando hacia seis grados de libertad. La alta gama ya dispone de ellos desde el principio, mientras la gama media y baja ya está adaptándose como demuestra Oculus Santa Cruz o Vive Focus.

2.2. Foto y vídeo en realidad virtual

El contenido que se genera todavía está basado en gran parte en técnicas ya conocidas como reproducción de vídeo y fotos cuya máxima adaptación consiste simplemente en poner una imagen ligeramente diferente en cada ojo.

La característica principal de este tipo de contenido es que cada imagen está tomada desde un punto fijo en el espacio. Esto crea una problemática que consiste en que el usuario únicamente tiene tres grados de libertad a la hora de visualizarlo en unas gafas de realidad virtual. Además de esta restricción debido a las técnicas de grabación que se usan, el cabeceo o *roll* provoca ver imágenes duplicadas y puede provocar incomodidad o incluso mareo.

Debido a las estadísticas de consumo del contenido multimedia grandes empresas como Google, Facebook y Disney entre otras, están trabajando en mejorar los sistemas de visualización de vídeo y fotos mediante vídeo volumétrico, campos de luz o fotogrametría.

2.3. Qué aporta este proyecto

Este trabajo toma la demo “Welcome to lightfields” como referencia pero creando un proyecto de código libre. Trata de conseguir una sensación real de tres dimensiones habilitando seis grados de libertad en un espacio reducido de desplazamiento, a partir de un vídeo estereoscópico plano en dos dimensiones y su mapa de profundidad. Para ello se hará un paralaje en tres dimensiones píxel a píxel en función del mapa de profundidad y la posición del usuario.

Durante el desarrollo del proyecto se llevan a cabo pruebas con diferentes técnicas y se evalúa la viabilidad en diferentes dispositivos (plataformas móviles y de sobremesa), el realismo del resultado así como la escalabilidad de los métodos.

3. Planteamiento del problema

La fotografía y el vídeo para realidad virtual actualmente tienen muchas limitaciones, tanto para producirlo como para visualizarlo. Una de estas limitaciones y del cual este proyecto se ocupa es la falta de libertad movimiento.

La foto y el vídeo solo tienen tres grados de libertad y de ellos cabeceo o *roll* (inclinar la cabeza sobre los hombros) no funciona como cabría esperar y puede provocar desde mareos hasta ver las imágenes duplicadas. La captura de imágenes, tanto reales como virtuales, se hace con cámaras que irremediablemente están en un punto concreto del espacio y eso en principio limita el movimiento del usuario.

En mayo de 2017, Google dio una charla aportando que cerca del 50 % del tiempo pasado en Daydream se centra en experiencias de vídeo. Un año después Facebook en el F8 [10], hablando sobre el estudio para el diseño de Oculus Go, puso de manifiesto que el 99 % de los usuarios consumen vídeo y que el 83 % de tiempo utilizado se destina a multimedia, llegando a la conclusión que es uno de los casos de uso principales.

Por ello, este trabajo se centra en mejorar la visualización del vídeo con un sistema que permita al usuario desplazarse físicamente dentro de un área, reaccionando el vídeo a ese posicionamiento en tiempo real.

4. Objetivos

Este proyecto ha sido creado con el objetivo de investigar, evaluar e implementar software para proporcionar seis grados de libertad en la visualización de contenidos a partir de imágenes planas y su mapa de profundidad aplicado en tecnologías de realidad virtual.

5. Estado del Arte

La acepción de realidad virtual que va a ser tratado en este trabajo es un mundo generado por ordenador. Se puede interactuar con este mundo por medio diferentes dispositivos como los hápticos o los olfativos, pero la forma más común actualmente y en la cual se va a centrar este documento es en la representación de imágenes mediante un casco o gafas de realidad virtual.

5.1. Marco teórico

5.1.1. Fotografía y vídeo

La fotografía y el vídeo en realidad virtual es un campo muy amplio en el que se pueden encontrar diferentes maneras guardar y reproducir la información. A continuación se comentan algunos conceptos importantes para entender el resto del documento.

Nota: Están organizados por conceptos alternativos, es decir, una foto no puede ser estereoscópica y monoscópica al mismo tiempo.

Estereoscopía/Monoscopía

La monoscopía implica que sólo existe una imagen para ambos ojos y por lo tanto no hay sensación de profundidad.

La estereoscopía es un factor muy importante en contenido multimedia para realidad virtual puesto que es el que más favorece la inmersión, como se explica en [1]. Consiste en tener dos imágenes que muestran la misma escena desde dos puntos cercanos a una distancia fija (normalmente de 6,5cm) y que representan la posición de los ojos (5.1). Cada una de estas imágenes se reproducen en una de las pantallas de las gafas, de tal manera que el cerebro del usuario se encarga de reconstruir la escena como si fuera realmente tridimensional. Aunque exista la ilusión de tridimensionalidad no es posible desplazarse por el entorno capturado.

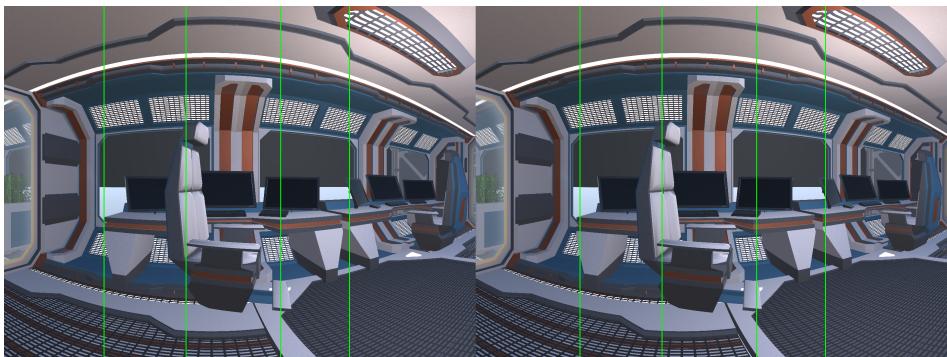


Figura 5.1: Ejemplo de imagen estéreo con lineas verticales para poder comparar el desplazamiento.

Es importante tener en cuenta que la separación entre las cámaras definirá la sensación de tamaño. Nuestro cerebro interpreta que si las cámaras están a menos distancia que nuestros ojos, todo es más grande y será mas pequeño si ponemos demasiada distancia.

Por último dentro de la estereoscopía se pueden encontrar las distribuciones arriba/abajo o *top/bottom*(5.2) y izquierda/derecha o *side by side*(5.3) que sitúa las imágenes de ojo derecho e izquierdo en vertical u horizontal respectivamente siendo la más extendida *top/bottom*.

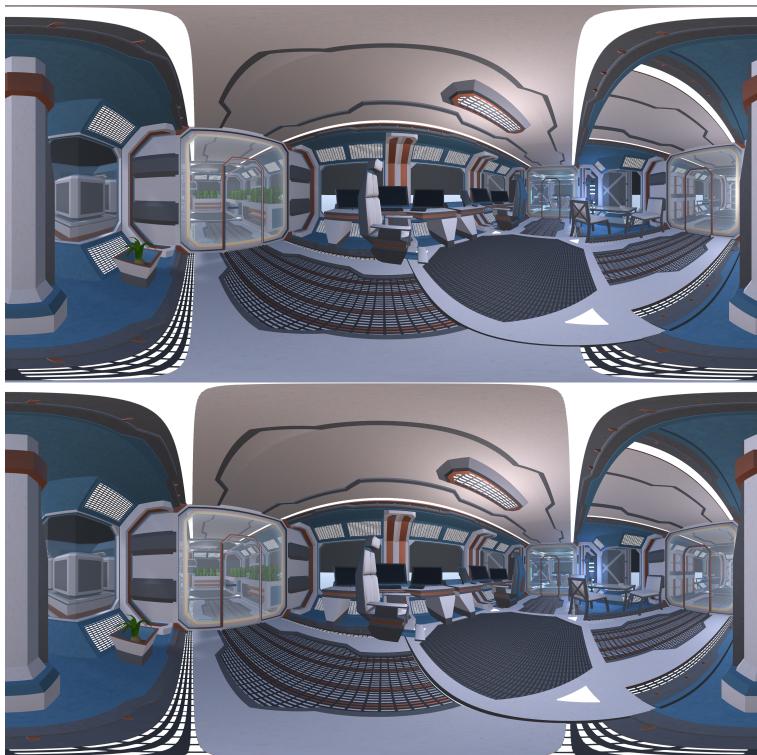


Figura 5.2: Ejemplo de *top/down*.

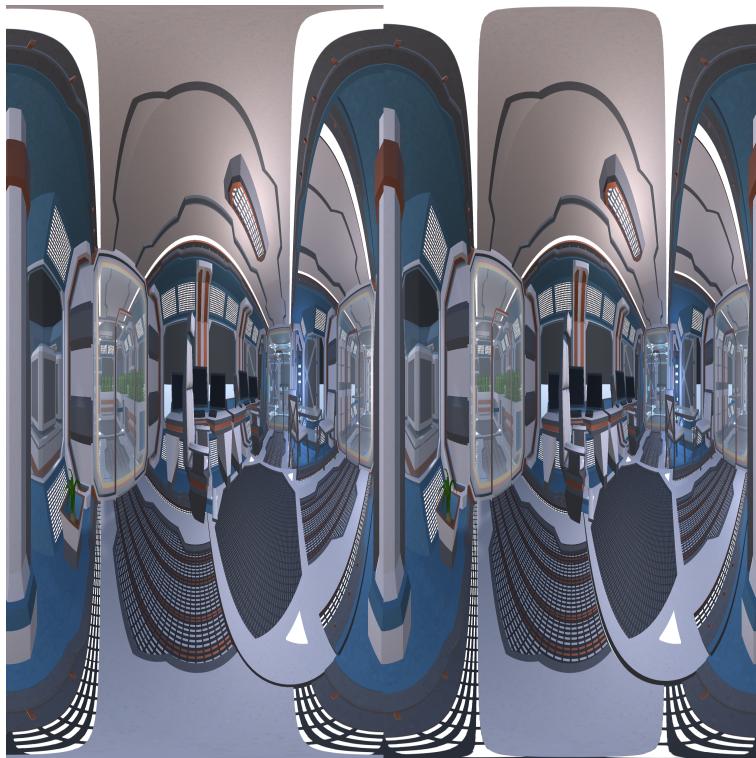


Figura 5.3: Ejemplo de *side by side*.

Campo de visión

Por lo general se utilizan dos ángulos para el campo de visión.

El más conocido y del que se suele hablar trata 360° de visión, es decir, una foto que genera una esfera alrededor de la vista del usuario. Es el que proporciona más inmersión.

El otro ángulo más utilizado es 180° que proporciona toda la visión frontal pero no hay imagen detrás. Las ventajas que más destacan son que puede

ser grabado con cámaras poco especializadas y que se puede concentrar mayor cantidad de píxeles en el frente, consiguiendo mayor calidad en la acción destacada.

Existen diferentes métodos para la captura de vídeo como se explicará más adelante.

Proyecciones

Una proyección es una forma de representación de un elemento tridimensional, en este caso una geometría esférica, en un espacio bidimensional. A la hora de guardar o reproducir un contenido multimedia es importante tener en cuenta la proyección a utilizar ya que pueden distorsionar la imagen y saturar regiones poco interesantes, como veremos a continuación, con una gran cantidad de píxeles.

Una de las proyecciones más extendidas y utilizadas en realidad virtual es la equirectangular (5.4 y 5.5), que coincide con la proyección más utilizada en la actualidad para representar el mundo en dos dimensiones. Esta proyección proporciona demasiada información en los polos y que típicamente es el lugar al que menos se suele mirar. Esto hace que gran parte de los píxeles se malgasten.

Facebook en [9] propone varias mejoras para su sistema de streaming de vídeo en realidad virtual 360°. Algunas de estas mejoras pueden ser utilizadas también en vídeo local como por ejemplo aplicar una distorsión intencionada a la imagen proporcionando más espacio a la parte central de la imagen y compensando esa distorsión en el reproductor y así obtener más definición en las zonas importantes.

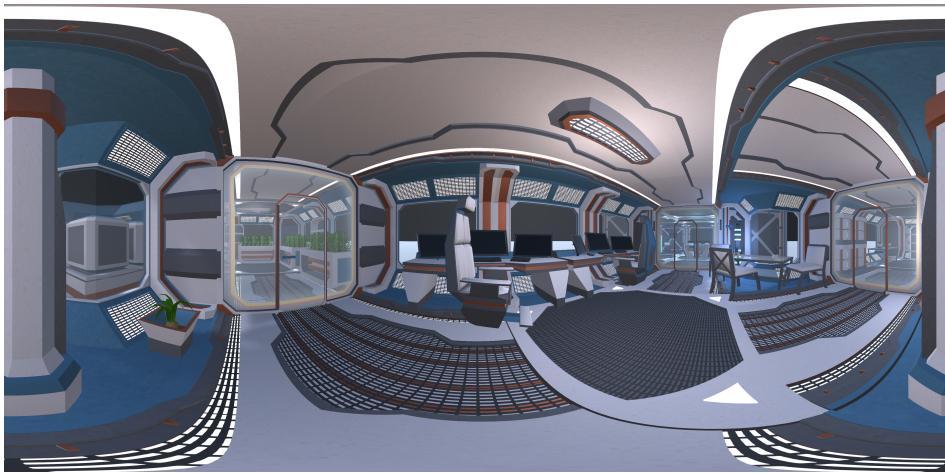


Figura 5.4: Ejemplo de una proyección equirectangular.

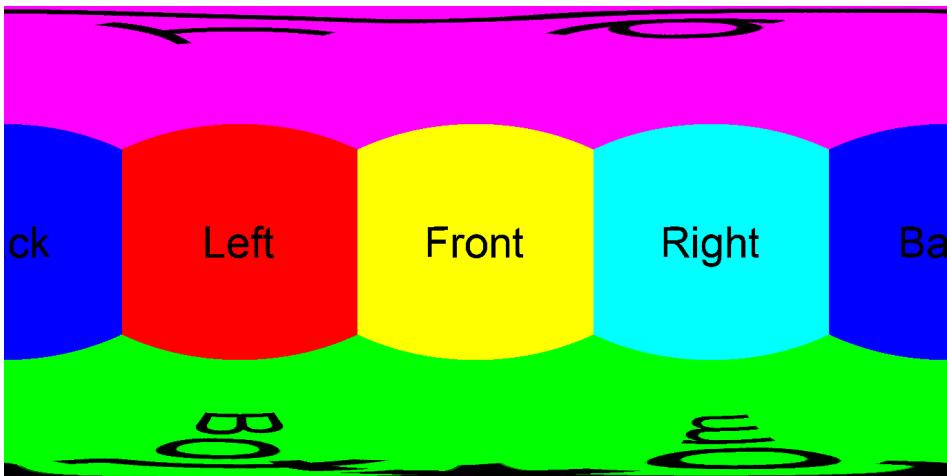


Figura 5.5: Esquema de una proyección equirectangular.

Otra proyección que se utiliza menos es la cúbica que divide la imagen en seis partes que forman las caras de un cubo. Tiene mayor densidad en las

aristas del cubo pero el porcentaje de píxeles útiles aumenta. Normalmente se le aplica una deformación al cubo dándole curvatura reduciendo la densidad de píxeles para que la distribución sea más uniforme.

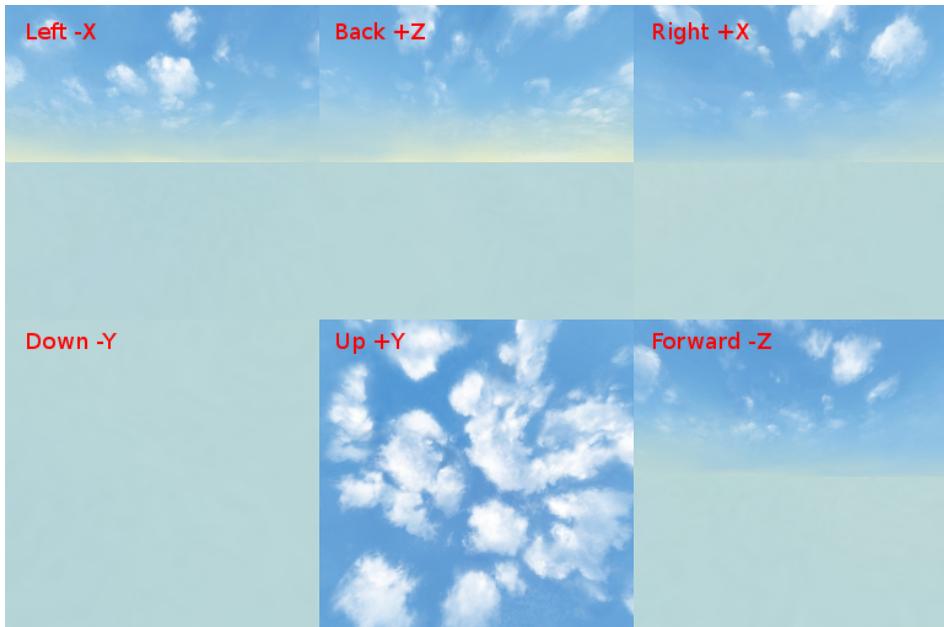


Figura 5.6: Ejemplo de proyección cúbica.

Por último mencionar la proyección de barril que se construye como un cilindro. La distribución de píxeles es uniforme en el campo de visión típico. Esta proyección desaprovecha píxeles que se pierden en los huecos que dejan las tapas del cilindro.

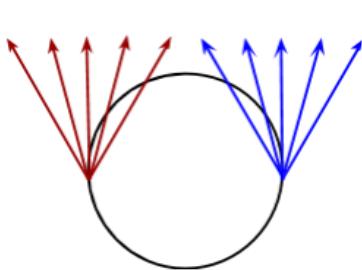
Existen una infinidad de proyecciones y cada una tiene una serie de ventajas e inconvenientes.

Omni-directional Stereo

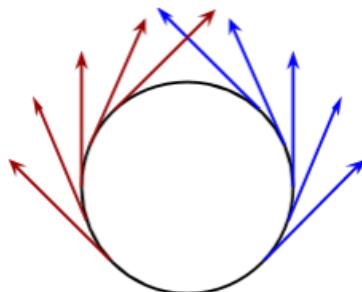
La captura con cámara de imágenes estereoscópicas 360° no es trivial.. Existen varios métodos para conseguir esto, y en vídeo el más común consiste en capturar un fotograma en diferentes puntos de una circunferencia teniendo en cuenta el radio de la cabeza, después esas fotos se unen y se reconstruye la imagen. Este proceso es complicado y si no se hace bien, se produce un efecto en el ensamblado llamado stitching y que se evidencia con unos saltos de color o cortes en objetos. Este problema no es trivial y cuesta eliminarlo como se comenta en [2].

Disney y Google proponen un sistema para captura de imágenes llamado ODS (Omni-directional stereo) que explican detalladamente en [11] y [14]. Este sistema consiste en dos cámaras, posicionadas a la distancia ocular y puestas en una circunferencia de un radio similar al de una cabeza. Durante la captura, las cámaras dan una vuelta por la circunferencia y guarda una linea de píxeles por cada posición de la cámara. Esto tiene como consecuencia que únicamente exista una linea de píxeles correcta en cada posición como se puede ver en la figura 5.7b. conlleva implicaciones como que los objetos en la visión periférica no se visualizan correctamente. Pese a esto, la sensación que se consigue es buena y proporciona una gran inmersión.

Por último comentar que el vídeo 180° tiene como beneficio que no necesita de ningún sistema como ODS ya que se hace directamente con cámara sin recurrir a métodos adicionales. Esto hace que todo el area de visualización sea correcta como se muestra en la figura 5.7a.



(a) Esquema de la visión completa.



(b) Esquema de la aproximación hecha con ODS.

Figura 5.7: Imágenes obtenidas de [11].

5.1.2. Mapas de profundidad

Debido a la cantidad de técnicas que utilizan mapas de profundidad o *depthmap* es interesante explicar en qué consisten.

Los mapas de profundidad son imágenes que en cada pixel se encuentra codificada la profundidad de la foto en ese punto. Generalmente se utiliza una escala de grises o de rojos aunque se pueden recurrir a métodos más complejos.[7]



Figura 5.8: Ejemplo de mapa de profundidad 360 en escala de rojos

En el caso de la escala de grises (5.9), los tonos más oscuros representan elementos en el fondo de la imagen, mientras que los tonos mas claro representan elementos más cercanos.

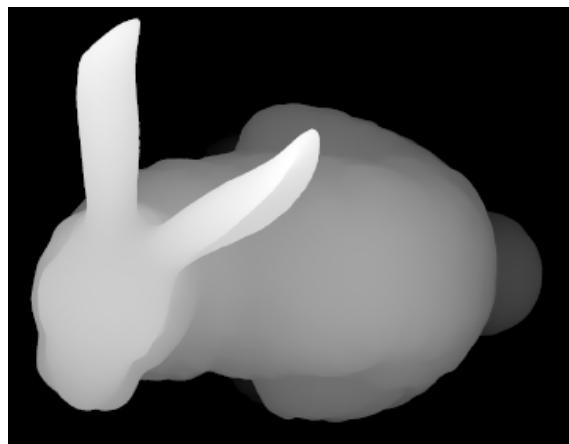


Figura 5.9: Ejemplo de mapa de profundidad cenital en escala de grises.

En el caso de una imagen generada por ordenador, es fácil obtener un buen mapa de profundidad. Sin embargo en el caso de las imágenes captadas

por cámaras reales, existe la posibilidad de que la cámara esté preparada o en caso contrario habría que aplicar algoritmos que calculen la profundidad en cada píxel.

Las cámaras que están preparadas para obtener el mapa de profundidad utilizan típicamente la emisión de infrarrojos. Existe una tecnología llamada LIDAR que obtiene mapas de profundidad de alta precisión con un haz láser, pero el tiempo que tarda en obtenerlo no lo hace compatible con la grabación de vídeo.

Dentro de los algoritmos que infieren el mapa de profundidad, los más conocidos utilizan imágenes estereoscópicas como el algoritmo BM intenta parear elementos que se encuentren a la misma altura y el algoritmo SGBM que es una variación del anterior añadiendo una ventana de búsqueda para encontrar las correspondencias.

5.1.3. Fotogrametría

La fotogrametría consiste en deducir la ubicación de múltiples puntos en el espacio a partir de una serie de fotografías. Después de eso se reconstruyen los triángulos para generar una malla texturizada que represente de la manera más fiel posible la escena fotografiada. Javier de Matías analiza en profundidad la fotogrametría en [4].

Al igual que en los mapas de profundidad, utilizar imágenes estereoscópicas ayudan a reconstruir la malla con mayor facilidad.

Algunos de los programas más conocidos para generación de mallas a partir de fotos son PIX4D y PhotoScan entre otros.

5.2. Project Sidewinder

Adobe presento en 2017 [3] una demo que utilizaba un *depthmap* de manera muy sencilla para permitir seis grados de libertad dentro de un video 360º. No proporcionan mucha información ya que es una prueba de concepto.

El desplazamiento punto a punto parece correcto pero se ve una distorsión en los bordes probablemente debido al estado temprano del proyecto.

5.3. Nube de puntos

Josh en [7] nos muestra una aplicación de esta técnica creando un punto en el espacio por cada píxel de la foto o el vídeo donde indique el mapa de profundidad.



Figura 5.10: Ejemplo de nube de puntos obtenida de [7].

Esta implementación por contra provoca que aparezcan muchos huecos como se puede ver en 5.9 y generalmente se suele acompañar la implementación con una selección del tamaño del punto como muestra [7] para ver una imagen más sólida.

Otro de los problemas que tiene esta técnica es la cantidad de puntos que deben ser tratados, ya que una resolución *QHD* (2560x1440) requiere cuatro millones de puntos siendo la resolución recomendada actualmente es *4K*. Probablemente el rendimiento sea bajo en equipos poco potentes y en dispositivos móviles.

5.4. Cámara y herramientas de realidad virtual de OTOY y Facebook

En el F8 de 2017 en Los Ángeles *OTOF* y *Facebook* presentaron una colaboración para producir vídeo 360º volumétrico e interactivo a un precio asequible [13]. No hay noticias de 2018 por lo que puede que esté abandonado.

La colaboración consistía en una cámara 360º especializada y una serie de herramientas para procesar el contenido y visualizarlo. El procedimiento implica subir el contenido a la nube de *OTOF* para procesarlo y así reconstruir la escena como una malla tridimensional.

La calidad presentada en las demos era buena con poca distorsión aunque el desplazamiento que presentaban era pequeño.

5.5. Facebook: Realidad virtual y Fotogrametría

Una de las formas de conseguir seis grados de libertad es reconstruir la escena mediante fotogrametría como muestra *Facebook* en [8] para procesar la imagen y obtener una malla que pueda ser mostrada usando técnicas convencionales.

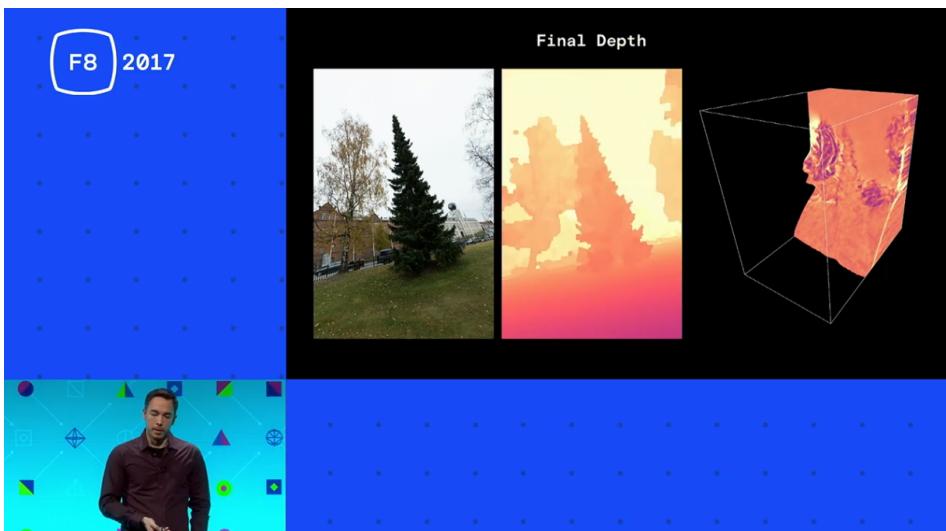
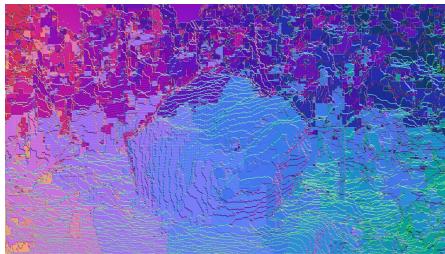


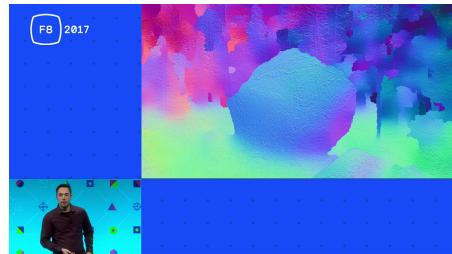
Figura 5.11: Conferencia mostrando un mapa de profundidad de límite inferior obtenida de [8].

Este procedimiento utiliza una técnica nueva que han llamado “mapa de profundidad de límite inferior” (5.9) que la profundidad de cada pixel debe ser estrictamente mayor que en el depthmap normal. Crear este mapa de profundidad le ayuda al ensamblado de las imágenes para crear la imagen final 360°. Además mezclando el mapa de profundidad con otros algoritmos son capaces de obtener un mapa de normales bastante preciso.

5.12b



(a) Conferencia mostrando los artefactos que obtienen.



(b) Conferencia mostrando los artefactos arreglados.

Figura 5.12: Imágenes obtenidas de [8].

Finalmente crean la malla a partir de una nube de puntos. Esta malla como veremos en otras técnicas, tiene agujeros detrás de los objetos que no pueden ser llenados por falta de información. En este caso optan por difuminar de manera sutil las zonas desconocidas dando un buen resultado.

Todo esto lo aprovechan para poder generar un entorno tridimensional con el que poder interactuar y lo ejemplifican jugando con la iluminación 5.13a o incluso inundando la escena 5.13b.



(a) Conferencia mostrando la malla iluminada.



(b) Conferencia mostrando un escenario inundándose.

Figura 5.13: Imágenes obtenidas de [8].

Una de las limitaciones que tiene este método es que está diseñado para fotografía en 360° y no se menciona en ningún momento al vídeo 360° por lo que se puede deducir que no está preparado. Por otro lado este tipo de procesado de imágenes requiere una cantidad grande de tiempo.

5.6. “Welcome to Lightfields”

Una compañía llamada Lytro creo un sistema que llamo campos de luz o Lightfields [12] por su similitud con el concepto físico. Más tarde Google se interesó por la compañía comprando algunas de sus patentes e incorporando empleados a su plantilla.

Google continuó el proyecto [5] y construyó un soporte que permite hacer fotografías de una escena desde puntos situados en una esfera.



Figura 5.14: Prototipo haciendo una captura del interior de una cabina. Obtenida de [5].

Este sistema no comprime las imágenes, sino que las utiliza directamente en el programa por lo que una “foto” en lightfields ocupa más de 250MB. A partir de todas las fotos se calcula la imagen correspondiente en tiempo real en función de la posición del usuario haciendo una interpolación entre diferentes imágenes, eso hace que se recupere una imagen muy fiel a lo que se vería en la realidad.

Como se puede apreciar en 5.15, el usuario debe estar dentro de la esfera de fotos, por lo que el rango de movimiento está limitado a menos de un metro.

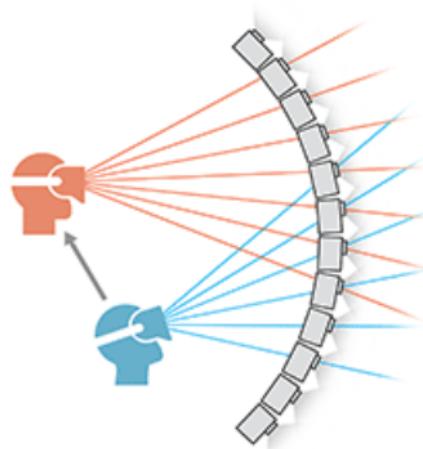


Figura 5.15: Esquema de funcionamiento de los lightfields. Obtenida de [5].

Además este sistema tiene una implicación que no había sido contemplada hasta ahora que son las superficies especulares. Estas superficies en el resto de técnicas se podían ver con reflejo o no, pero nunca respondería a la posición del usuario. La interpolación de imágenes hace que los espejos reflejen algo diferente en cada posición de la cabeza.

Esta técnica para fotografía es probablemente la mejor en cuanto a calidad de imagen pero sin embargo no se puede utilizar en vídeo debido al tiempo que se tarda en capturar un sólo fotograma. Tampoco permite alterar la luz como se mostraba en la demo de *Facebook*.

6. Desarrollo

El principal objetivo del trabajo es implementar un algoritmo que aporte seis grados de libertad a vídeo e imagen para realidad virtual y que se pueda ejecutar consumiendo la menor cantidad de recursos posibles. Para poder tratar este planteamiento en tiempo real es necesario utilizar herramientas de aceleración por *hardware*.

Las tecnologías que se van a explorar para los diferentes métodos son las siguientes:

- Unity como motor de desarrollo por ser de ámbito profesional y multiplataforma.
- *Shaders* gráficos como herramienta que permita la explotación de la GPU mediante técnicas clásicas de gráficos.
- *Shaders* de cómputo como herramienta alternativa a los *shaders* gráficos por la versatilidad que proporcionan.
- *Visual Studio* como entorno de edición y depuración de código por la cantidad de *plugins* disponibles.

6.1. El entorno de desarrollo

Los motores gráficos y de videojuegos son una herramienta para agilizar el desarrollo de demos y de aplicaciones multiplataforma. Unity se encarga de proporcionar la infraestructura necesaria para poder centrarse en el desarrollo específico del proyecto. Los principales elementos a utilizar serán:

- Las escenas se podrían definir como el contenedor global de objetos, cámaras y luces.
- Los GameObject son todos los elementos que pueden ser metidos en una escena y sus componentes que son la manera de añadirles lógica mediante el lenguaje de programación C # .
- Las cámaras que son GameObjects especializados en obtener imágenes del entorno tridimensional y que pueden convertirse en cámaras de realidad virtual.

6.1.1. Shaders Gráficos

Los *shaders* gráficos configuran la *pipeline* gráfica, que tiene la función de recibir una representación de la escena tridimensional como entrada y generar una imagen bidimensional como salida.

Son una de las partes de la infraestructura de Unity y se configuran mediante unos elementos llamados materiales.

Existen diferentes maneras de programar los *shaders* gráficos, entre las cuales se ha decidido utilizar los métodos de modificación de vértices y fragmentos, por tener experiencia previa utilizándolos y ofrecer lo necesario para el desarrollo.

6.1.2. Shaders de cómputo

Los *shaders* de cómputo surgieron como respuesta a los desarrolladores que utilizaban los *shaders* gráficos con un propósito general para aprovechar la capacidad de cálculo de la tarjeta gráfica.

Unity también proporciona maneras de utilizar estos *shaders* y al contrario que los anteriores, no están sujetos a una *pipeline*, sino que pueden ser utilizados en cualquier momento.

Estos *shaders*, al utilizar la GPU al igual que los gráficos pueden hacer uso, si así se les indica, de recursos de manera compartida. Esta funcionalidad es importante porque será utilizada durante el desarrollo.

6.2. Metodología

La metodología utilizada en el desarrollo se ha basado en *Scrum*, proponeiendo pequeños *sprint* una vez terminado el anterior. Se podrían clasificar estos sprint en tres grupos:

- Investigación, cuya motivación era el descubrimiento de tecnologías utilizadas por otros desarrolladores para conseguir mejorar los resultados ,tanto en calidad como en rendimiento.
- Implementación de nuevos algoritmos o mejoras de los ya existentes, habitualmente, en función de lo investigado.
- Evaluación de la validez de la implementación y poner de manifiesto los problemas que deben ser solucionados.

Como las implementaciones son diferentes entre ellas y se han realizado muchas pruebas, se van a ir comentando los resultados en cada iteración, en lugar de comentarlos todos al final.

Las pruebas se realizan en un ordenador de sobremesa con un procesador i7-7700K, una tarjeta gráfica Nvidia Geforce GTX1070 y 16 GB de RAM.

6.3. Generación de imágenes con las que trabajar

Con la finalidad de tener un entorno de pruebas controlado y fiable, se ha montado una escena con contenido gratuito del Asset Store. Las principales características buscadas eran un entorno con suficientes elementos con una buena distribución, que el entorno fuera complejo y que los tamaños de los objetos fueran coherentes. Finalmente se eligió el paquete “Sci-Fi Styled Modular Pack” (disponible en “<https://assetstore.unity.com/packages/3d/environments/sci-fi/sci-fiStyled-modular-pack-82913>”) y se estableció la cámara como indican las fotos 6.1a y 6.1b.



(a) Posición desde un punto de vista.



(b) Posición desde otro punto de vista.

Figura 6.1: Posición de la cámara para captar imágenes estéreo 360°.

Esta posición se ha elegido por varias razones. La primera es por la silla en los escritorios y la columna situada detrás de la cámara. Son objetos

bastante cercanos con otros elementos detrás de ellos, lo que permite comprobar si el desplazamiento con los 6 grados de libertad es correcto y se produce paralaje.

A la derecha de la cámara hay un pasillo con muchas líneas en el suelo que sirven para verificar que las líneas del suelo no se deforman. Además hay cristales para ver como se comporta el algoritmo en ese caso.

Una vez elegido el entorno para trabajar, se procede a capturar una imagen estéreo 360°. Para ello se utiliza una característica de las cámaras de Unity llamada `RenderTargetToCubemap` y que permite capturar *Cubemaps* basándose en el algoritmo de Google ODS ya mencionado en el “Estado del arte” y disponible en [11]. Este procedimiento hay que repetirlo una vez por cada ojo para obtener una imagen estéreo correcta. Esta característica proporciona dos *Cubemaps* y deben ser convertidos a una textura equirrectangular estereoscópica *top/down*. Una vez hecho, se codifica en un archivo de imagen PNG.

Por otro lado se necesita el mapa de profundidad estereoscópico que se consigue con un procedimiento parecido, pero con una serie de pasos previos que preparen la escena y restableciendo el estado anterior al final del *render*. Primero y más importante hay que construir un *shader* que pinte la profundidad, para ello hay diferentes maneras de hacerlo de las cuales se han probado dos:

- La primera que se implementó consiste en codificar la profundidad en las cuatro componentes de la imagen con la función que proporciona Unity llamada “`EncodeFloatRGBA`”. Este método se abandonó por aumentar la complejidad y el ruido ante posibles errores a la hora de desarrollar. Se puede ver un ejemplo en 6.2.

- Por otro lado la profundidad podía ser guardada como un solo número, y ante esto existía la posibilidad de utilizar una textura de con el formato “RFloat” que equivale a un único número en coma flotante de 32 bit por cada píxel, o una textura “ARGB32” habitual y guardarla en una de sus componentes. Se optó finalmente, por compatibilidad con diferentes plataformas, por la segunda forma guardando la profundidad en la componente roja. Se puede ver un ejemplo en 6.3.

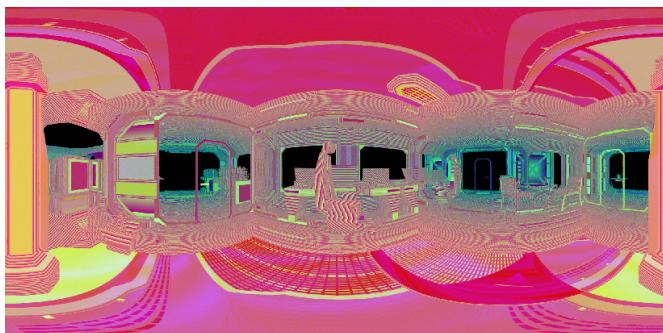


Figura 6.2: Mapa de profundidad codificado a float.



Figura 6.3: Mapa de profundidad guardado en la componente r de una textura “ARGB32”.

A continuación se guarda el estado de la escena y se configuran elementos como el “*Rendering Path*” o las “*Clear Flags*” para asegurar que las opciones por defecto no afectan al *render*. Este *shader* va a reemplazar todos los tipos de *shader* que se encuentren en la escena, por lo que hay que repetir parte del *shader* con diferentes opciones de “*RenderType*”. Por último hay que reemplazar el *shader* por defecto por este *shader* con la instrucción

```
camera.SetReplacementShader(depthShader, "RenderType");
```

y establecer los parámetros configurables del *shader*. Para mejorar la calidad del mapa se define un parámetro para establecer una profundidad máxima, con el fin de evitar elementos cercanos con poca precisión, ya que es dónde más se va a apreciar el paralaje.

Por último y como se había dicho anteriormente, se restablecen los parámetros de la escena para evitar que afecte a otras tareas posteriores.

6.4. Primera aproximación: Paralaje sencillo

La primera aproximación que se realiza consiste en un paralaje sencillo. Para ello se crea una escena nueva en la que hay una cámara de realidad virtual, y una esfera muy grande con un *shader*, para imágenes y vídeo 360, sencillo del paquete: Google VR.

6.4.1. Qué es el paralaje

Si un observador toma una foto y luego mueve la cámara hacia la derecha sin rotarla y toma otra foto, algunos objetos o parte de ellos que aparecían en la primera imagen, serán tapados por otros en la segunda imagen. Si se comparan ambas fotos tomando la primera como referencia, se puede apreciar que los objetos cercanos están aparentemente más desplazados en la segunda foto que los objetos lejanos. Esto es causado porque el ojo humano y las cámaras utilizan una proyección en perspectiva.

Este fenómeno es llamado paralaje y puedes ver el ejemplo con tres fotos en 6.4.

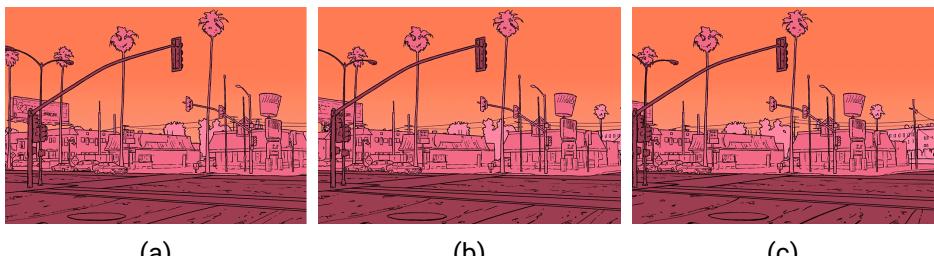


Figura 6.4: Ejemplo de paralaje en tres fotogramas.

6.4.2. Implementación

Es necesario hacer un paralaje en una imagen plana y para ello podemos mover píxeles por la superficie de la foto. Como se ha comentado en el apartado anterior, la cantidad que deben moverse los píxeles depende del dato obtenido del mapa de profundidades llevando a cabo una técnica similar a [6].

La primera idea es modificar el *shader* de visualización de vídeo y foto añadiéndole dos parámetros adicionales:

- *Parallax Amount*: cuanto un píxel se debe desplazar teniendo en cuenta su profundidad.
- *Relative Position*: simula el desplazamiento de la cámara en el eje x en espacio de vista.

Estos parámetros se exponen con la finalidad de hacer una evaluación rápida de los resultados. Más adelante, la posición relativa será establecida mediante los datos que proporciona el casco de realidad virtual.

El *shader* de vértices no es interesante para este caso, por lo tanto se va a comentar el *shader* de fragmentos:

```
float4 frag (v2f i) : SV_Target {
    // Obteniendo la profundidad
    float h = DecodeFloatRGBA(tex2D(_DepthTex, i.uv));
    // Calculo del desplazamiento usando la profundidad.
    float uDisplacement = h * _ParallaxAmount
        * _RelativePosition
        * _DepthTex_TexelSize.x * 40;
    // Obteniendo el pixel de la posicion desplazada.
    return gammaCorrect(tex2D(_MainTex,
        i.uv + float2(uDisplacement, 0)));
}
```

Cada píxel “A” en este caso, escoge el píxel “B” que se encuentra en la posición desplazada con respecto a la profundidad del píxel “A”, esto quiere decir que el píxel “B” se está desplazando tanto como indica la profundidad de “A”. Esto es una mala aproximación porque se tiene en cuenta la profundidad del píxel “A” y no el de “B”.

6.4.3. Resultado

Como se ve en la figura 6.5, se puede apreciar que los píxeles del fondo están siendo superpuestos sobre la silla y sin embargo, la silla no está siendo desplazada sobre el fondo.

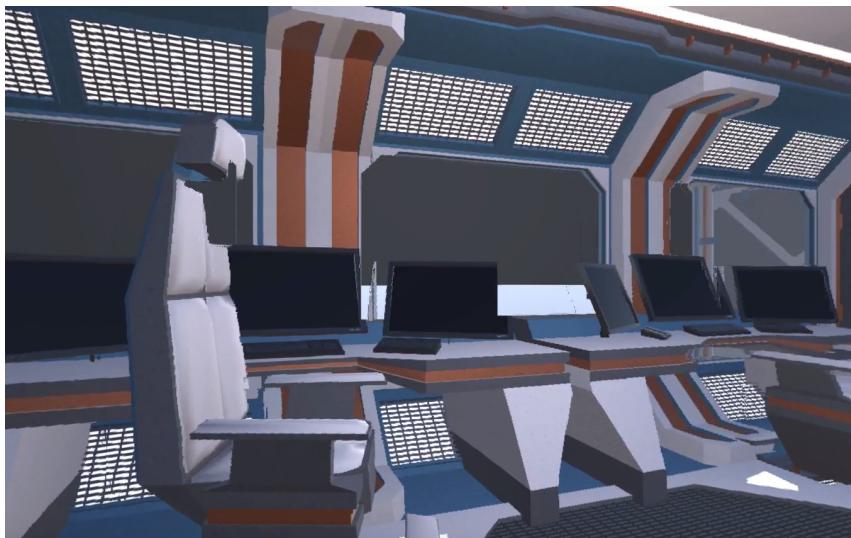


Figura 6.5: Paralaje erróneo usando el desplazamiento equivocado.

Se puede ver el vídeo en: <https://www.youtube.com/watch?v=F6zlchbR1Rg>

6.5. Segunda aproximación: Paralaje sencillo

El problema de la aproximación anterior era un mal planteamiento, debemos saber qué píxel B debe ocupar la posición del píxel A, pero la información para calcularlo, está en el píxel B. Por lo tanto para solucionar este

problema de paralaje, la siguiente modificación del *shader* realiza una búsqueda por la vecindad para encontrar cual es el mejor píxel para ocupar el actual.

6.5.1. Implementación

Esta idea realiza demasiados accesos a texturas y puede traer un problema muy grave de rendimiento. Sabiendo la dirección en la que se desplaza la cámara, sabemos la dirección en la que los píxeles se van a desplazar. De esta manera, simplificamos una búsqueda que implica un número de accesos a memoria cuadrático, en función de la distancia, a lineal.

Este es el nuevo *shader* de fragmentos comentando las líneas principales.

```
float4 frag (v2f i) : SV_Target {
    // obtener la altura
    float height = DecodeFloatRGBA(tex2D(_DepthTex, i.uv));
    float maxHeight = 0;
    float2 bestUV = 0;
    float u = 0;
    bool found = false;

    // Dimension de texel con signo dependiendo
    // de la direccion de desplazamiento
    float texelSignedSize = sign(-_RelativePosition)
        * _DepthTex_TexelSize.x;
    // factor de desplazamiento
    float displacementFactor = _ParallaxAmount
        * -_RelativePosition
        * _DepthTex_TexelSize.x * 40;
```

```

for (int itCount = 0; itCount < 40; itCount++) {
    // selecciona cada pixel en la linea elegida
    float2 currUV = i.uv + float2(u, 0);
    float currCellHeight = DecodeFloatRGBA(
        tex2D(_DepthTex, currUV));
    float currCelluDispl = currCellHeight
        * displacementFactor;
    // calcular donde se deberia mover el pixel seleccionado
    float2 newUV = currUV + float2(currCelluDispl, 0);

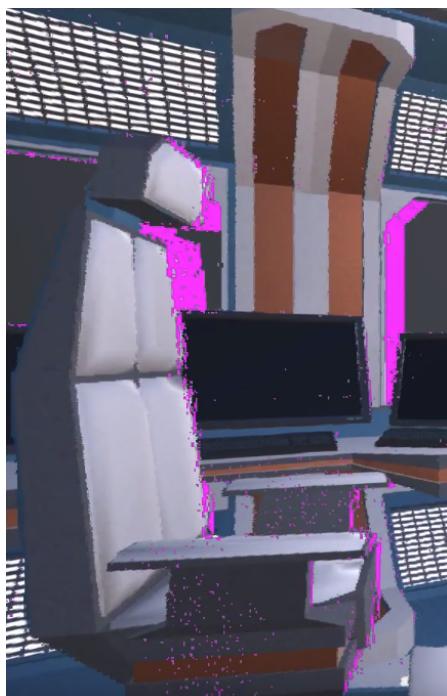
    // si el pixel seleccionado esta en los limites
    // del pixel actual, es un buen candidato
    if (abs(newUV.x - i.uv.x) <= _DepthTex_TexelSize.x) {
        // si es el pixel menos profundo,
        // entonces es el mejor candidato hasta ahora
        if (currCellHeight >= maxHeight) {
            maxHeight = currCellHeight;
            bestUV = currUV;
            found = true;
        }
    }
    // pase de iteracion
    u -= texelSignedSize;
}

// si hay un buen candidato, pintalo
// en otro caso, pinta en rosa
return found ? gammaCorrect(tex2D(_MainTex, bestUV))
    : float4(1.0, 0.0, 1.0, 1.0);
}

```

6.5.2. Resultado

Esta aproximación es notablemente mejor, puesto que como se puede ver en 6.6, los píxeles se desplazan en la dirección correcta, sin embargo se ve mucho ruido. Este ruido es causado las imágenes que son utilizadas, ya que en las opciones de Unity tiene se han activado tanto la compresión como el filtro bilineal.



(a) Relative position a 0.



(b) Relative position a 1.

Figura 6.6: Paralaje correcto con ruido.

Por simplicidad, se están utilizando accesos contiguos a textura y están ofreciendo un número alto de fotogramas por segundo. Si cambiamos el

acceso añadiéndole otra dimensión de desplazamiento al *shader*, los tiempos de acceso aumentarían mucho. Además el algoritmo no es escalable, pues cuanta más distancia haya que desplazar, más accesos son necesarios.

Se puede ver el vídeo en: https://www.youtube.com/watch?v=wDxo_LH5Wjs

6.6. Tercera aproximación: Paralaje sencillo con *shaders* de cómputo

El último método da unos resultados razonablemente buenos pero el problema reside en el número de accesos a memoria hechos.

Los *shaders* gráficos tienen una limitación muy grande que consiste en que cada fragmento/píxel debe seleccionar su color, no puede pintar otro píxel del color indicado. Por este motivo se decide optar por los *shaders* de cómputo.

6.6.1. Implementación

Esta idea realiza demasiados accesos a texturas y puede traer un problema muy grave de rendimiento. Sabiendo la dirección en la que se desplaza la cámara, sabemos la dirección en la que los píxeles se van a desplazar. De esta manera, simplificamos una búsqueda que implica un número de accesos a memoria cuadrático en función de la distancia a lineal.

Este es el nuevo *shader* de fragmentos comentando las líneas principales. La ejecución consiste en una llamada al *kernel* “*Clear*” para limpiar los resultados y otra llamada al *kernel* “*DisplaceAlbedo*”. El resultado se guarda en una *RenderTexture* asociada al *shader* gráfico básico de la esfera.

```
// Public kernels -----
#pragma kernel Clear
#pragma kernel DisplaceAlbedo
#include "UnityCG.cginc"

// Config -----
// 360 stereo depth texture (IN)
Texture2D<fixed4> DepthTexture;
// 360 stereo albedo texture (IN)
Texture2D<fixed4> AlbedoTexture;
// 360 stereo computed result (OUT)
RwTexture2D<float4> Result;
// Los mismos parametros de configuracion
// de los otros shader
float RelativePosition;
float ParallaxAmount;

// inicializar con rosa para ver los huecos
[numthreads(32, 32, 1)]
void Clear(uint3 id : SV_DispatchThreadID) {
    Result[id.xy] = float4(1.0f, 0.0f, 1.0f, -1.0f);
}

[numthreads(32, 32, 1)]
void DisplaceAlbedo (uint3 id : SV_DispatchThreadID) {
    // descodificar la altura
    float height = DecodeFloatRGBA(DepthTexture[id.xy]);
```

```

// calculo de desplazamiento
float displacementFactor = height
    * ParallaxAmount
    * RelativePosition;

// aplicar el desplazamiento
uint2 newUV = uint2(
    (id.x + displacementFactor) % 4096,
    id.y);
// devolver el albedo en la posicion calculada
Result[newUV.xy] = float4(AlbedoTexture[id.xy].rgb, 1.0);
}

```

6.6.2. Resultado

Esta aproximación tiene un problema nuevo, que es el de concurrencia. Cuando un hilo calcula la nueva posición para escribir, muchos hilos pueden intentar escribir en la misma posición y el que termina escribiendo es uno aleatorio entre todos. Si eso ocurre, se ve algo parecido al “z-fighting” que provoca el parpadeo de píxeles entre los diferentes colores que intentan ser escritos.

Como se puede ver en la figura 6.7, las líneas negras señaladas con los círculos rojos, parpadean durante la ejecución.

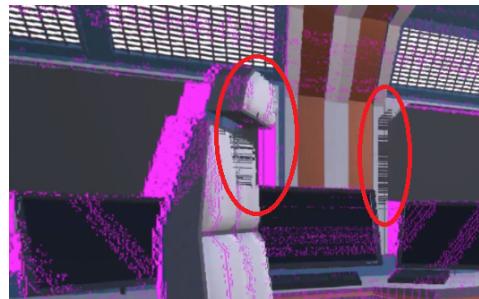


Figura 6.7: Paralaje con condición de carrera.

Se puede ver el vídeo en: <https://www.youtube.com/watch?v=R2rfZYyaCcE>

6.7. Cuarta aproximación: Resolviendo la condición de carrera

Para solucionar la condición de carrera la primera idea que surge es utilizar un *z-buffer*.

6.7.1. Implementación

Para esta implementación, se pretende utilizar la instrucción `InterlockedMax` que asegura una escritura atómica del máximo entre la casilla seleccionada y el dato proporcionado a la función. Primeramente se utiliza una textura, pero esta instrucción no es compatible con texturas, por lo que se lleva a cabo con un *buffer* lineal de `int`.

Por simplificación, se eliminan las líneas ya explicadas en anteriores apartados. Esta vez se incorpora un nuevo *kernel* llamado “`WriteDepth`” que deberá ser llamado antes del “`DisplaceAlbedo`”

```

...
#pragma kernel WriteDepth
...
RWStructuredBuffer<int> Depth;
...
[numthreads(32, 32, 1)]
void Clear(uint3 id : SV_DispatchThreadID) {
    Result[id.xy] = float4(1.0f, 0.0f, 1.0f, -1.0f);
    Depth[bufferPos(id)] = -1;
}

uint bufferPos(uint3 id) {
    return id.x + id.y * 4096;
}

[numthreads(32, 32, 1)]
void WriteDepth(uint3 id : SV_DispatchThreadID) {
    float height = DecodeFloatRGBA(DepthTexture[id.xy]);
    float displacementFactor = height
        * ParallaxAmount
        * RelativePosition;
    uint2 newUV = uint2((id.x + displacementFactor) % 4096,
                        id.y);
    int heightInt = height * 1024;

    // Se guarda la altura correcta en el buffer
    InterlockedMax(Depth[bufferPos(newUV)],
                    heightInt);
    AllMemoryBarrier();
}

```

```

[numthreads(32, 32, 1)]
void DisplaceAlbedo(uint3 id : SV_DispatchThreadID) {
    int heightInt = Depth[bufferPos(id)];
    float height = (heightInt / 4096.0f);
    float displacementFactor = height
        * ParallaxAmount
        * -RelativePosition;
    uint2 newUV = uint2((id.x + displacementFactor) % 4096,
                           id.y);

    // Se comprueba si la altura calculada
    // es la que debe usarse
    float3 albedoColor = heightInt == -1 ?
        float3(1.0f, 0.0f, 1.0f) :
        AlbedoTexture[newUV.xy].rgb;
    Result[id.xy] = float4(albedoColor, height);
}

```

6.7.2. Resultado

Este paraje es el que mejor funciona hasta ahora como se puede ver en la figura 6.8. El problema reside en que la función *InterlockedMax* no esta estandarizada en todos los dispositivos. Además hay que llamar a tres *kernels*, por lo que la cantidad de cálculos aumentan.



Figura 6.8: Paralaje InterlockedMax.

Se puede ver el vídeo en: https://www.youtube.com/watch?v=MSG_IU-Xcjc

6.8. Quinta aproximación: Resolviendo la condición de carrera 2

Llegados a este punto se opta por utilizar métodos más convencionales y se dejan sólo el kernel “*Clear*” y “*DisplaceAlbedo*”.

```
...
[numthreads(32, 32, 1)]
void DisplaceAlbedo(uint3 id : SV_DispatchThreadID) {
    float height = DepthTexture[id.xy].r;
```

```
float displacementFactor = (float)height
    * (float)ParallaxAmount
    * -RelativePosition;
uint3 newUV = uint3(
    (id.x + displacementFactor + 4096) % 4096,
    id.y,
    id.z);

// Comprobar si se debe escribir muchas
// veces para evitar sobreescrituras
if ( Result[newUV.xy].a < DepthTexture[id.xy].r ) {
    Result[newUV.xy] =
        fixed4(AlbedoTexture[id.xy].rgb,
               DepthTexture[id.xy].r);
}
if ( Result[newUV.xy].a < DepthTexture[id.xy].r ) {
    Result[newUV.xy] =
        fixed4(AlbedoTexture[id.xy].rgb,
               DepthTexture[id.xy].r);
}
if ( Result[newUV.xy].a < DepthTexture[id.xy].r ) {
    Result[newUV.xy] =
        fixed4(AlbedoTexture[id.xy].rgb,
               DepthTexture[id.xy].r);
}
if ( Result[newUV.xy].a < DepthTexture[id.xy].r ) {
    Result[newUV.xy] =
        fixed4(AlbedoTexture[id.xy].rgb,
               DepthTexture[id.xy].r);
}
if ( Result[newUV.xy].a < DepthTexture[id.xy].r ) {
    Result[newUV.xy] =
        fixed4(AlbedoTexture[id.xy].rgb,
               DepthTexture[id.xy].r);
}
```

```
        }  
    }
```

6.8.1. Resultado

Este simple algoritmo elimina el parpadeo en un porcentaje muy alto, dando los mismos resultados que en el caso anterior. Además el rendimiento mejora con respecto a la versión de *InterlockedMax*.

En estos momentos el *shader* proporciona entre 160 y 180 fotogramas por segundo.

6.9. Aproximación final: Pasando a espacio de mundo

Hasta ahora todo lo que se había desarrollado era haciendo un desplazamiento de los elementos de la textura en la dimensión X. Esto es un desplazamiento falso, pues en un caso extremo podríamos tener delante de los ojos un elemento que se encuentre en nuestra espalda.

Para solucionar esto, se va a recalcular el desplazamiento en coordenadas de mundo.

6.9.1. Implementación

Para traducir a coordenadas de mundo hay que tener en cuenta que las dimensiones de la imagen están relacionadas con las coordenadas polares. Por lo tanto el orden de operaciones debería ser el siguiente:

1. Transformar de polares a cartesianas.
2. Desplazar las coordenadas cartesianas según el vector de desplazamiento indicado.
3. Transformar de cartesianas a polares.

A continuación se puede ver la función que permite aplicar el desplazamiento. Hay muchas constantes nuevas que se utilizan para hacer los cálculos, algunas de ellas se podrían transformar en variables para ser modificadas externamente como la dimensión de la textura, pero para favorecer unas pruebas ágiles se considera innecesario.

```
static const float _PI = 3.14159265359;
static const float _2PI = 6.28318530718;
static const float _PI_2 = 1.57079632679;

static const uint2 TexDim = uint2(4096, 4096);
static const uint2 TexGrid = uint2(1, 2);
static const uint2 GridElemDim = uint2(4096, 2048);
static const uint FloatToUintDepthMultiplier = 4096;
static const float UintToFloatDepthMultiplier = 1.0/4096.0;

...
```

```
uint3 Displacement(uint2 id,
                      float height,
                      float3 displacementVector) {

    // cambiar de altura a profundidad
    float depth = 1 - height;

    // id = ([0, 4095], [0, 2047])
    float2 onePixelInRadians = float2(_2PI, _PI)
        / (GridElemDim - 1);

    // calcular radianes de coordenadas polares
    float2 theta_phi = (id.xy * onePixelInRadians)
        - float2(_PI, _PI_2);

    // calcular senos y cosenos de theta phi
    float2 sin_theta_phi = sin(theta_phi);
    float2 cos_theta_phi = cos(theta_phi);

    // pasar a coordenadas de mundo
    float3 worldPos =
        float3(cos_theta_phi.y*sin_theta_phi.x,
                sin_theta_phi.y,
                cos_theta_phi.y*cos_theta_phi.x) * depth;

    // desplazar las coordenadas en
    // la direccion indicada
    worldPos += displacementVector;

    // calcular la nueva profundidad
    float newDepth = length(worldPos);
```

```

// y normalizar la nueva posicion
float3 newWorldNorm = worldPos / newDepth;

// calcular atan2 para pasar a polares
float atan2ThetaPhi = atan2(newWorldNorm.x,
                             newWorldNorm.z);

// calcular coordenadas polares
theta_phi = float2(atan2ThetaPhi,
                  asin(newWorldNorm.y));
theta_phi = (theta_phi / float2(_2PI, _PI) + 0.5)
            * (GridElemDim - 1);

// devolver las coordenadas polares
// y la nueva profundidad
return uint3(theta_phi,
              (1-newDepth) * FloatToUintDepthMultiplier);
}

...
[ numthreads(32, 32, 1) ]
void WriteDepth(uint3 id : SV_DispatchThreadID) {
    float height = DepthTexture[id.xy].r;
    // calculamos el cuadrante
    uint2 quadrant = id.xy / GridElemDim;

    // Se aplica el desplazamiento
    uint3 newUV = Displacement(
        uint2(id.x, id.y) % GridElemDim,
        height,
        float3(ParallaxAmount *
                -RelativePosition,

```

```

        0, 0));

// devolvemos las uv a su cuadrante
newUV.xy += quadrant * GridElemDim;

// MULTIWRITE MODE
if ( Result[newUV.xy].a < DepthTexture[id.xy].r ) {
    Result[newUV.xy] = fixed4(AlbedoTexture[id.xy].rgb,
                           DepthTexture[id.xy].r);
}
...
}
}

```

6.9.2. Resultado

Como se puede observar en 6.9 y 6.10, el desplazamiento producido es bastante realista, pero deja demasiados huecos rosas.

Este *shader* reduce el número de fotogramas por segundo un poco, dejándolo en aproximadamente 140-160 fotogramas por segundo.

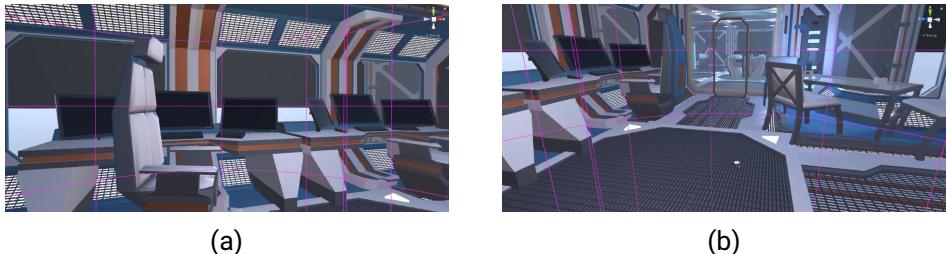


Figura 6.9: Sin desplazamiento.

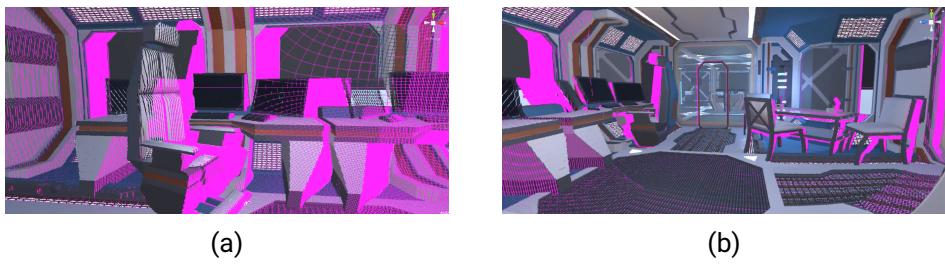


Figura 6.10: Con el máximo desplazamiento.

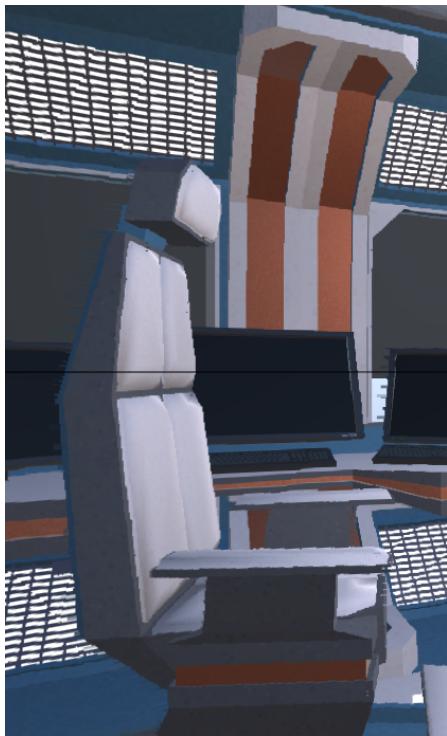
Se puede ver el vídeo en: <https://www.youtube.com/watch?v=vdR3ZICLFCQ>

6.10. Otras pruebas

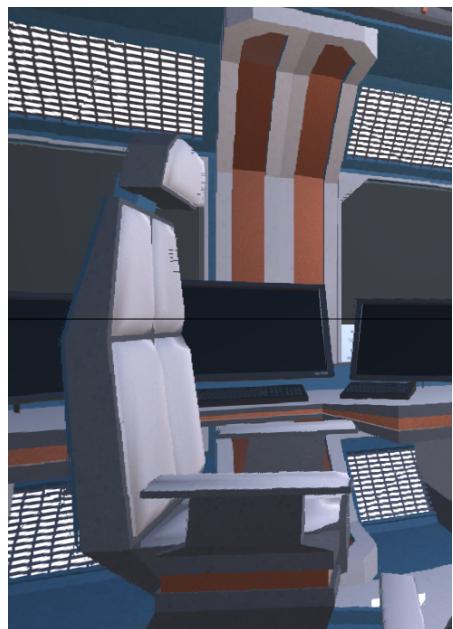
6.10.1. Evitar borrar el *buffer* de color

Además de lo visto en el *shader*, se han realizado pruebas limpiando únicamente el *buffer* de profundidad que actualmente es la componente *alpha* de la textura de resultados. Estas pruebas evitan la aparición de elementos rosas y rellena con el último color que tuvo esa textura.

Esto da resultados visualmente muy buenos, pues los píxeles vacíos provocados por un estiramiento de la textura, son tapados automáticamente. No da tan buen resultado con los grandes huecos provocados por saltos de profundidad.



(a)



(b)

Figura 6.11: Desplazamiento sin borrar el color.

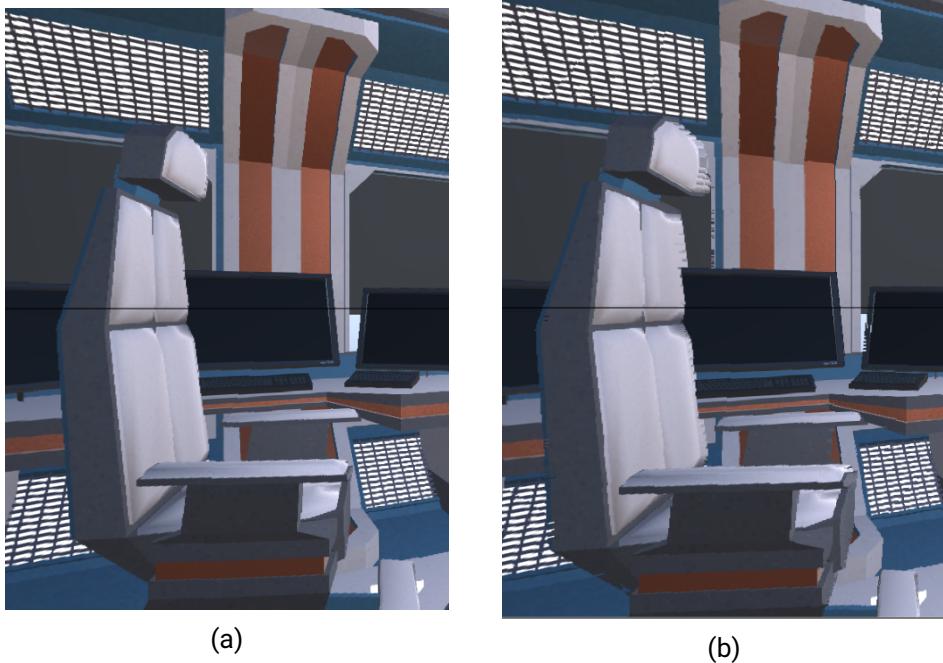


Figura 6.12: Desplazamiento sin borrar el color.

6.10.2. Desplazamiento en espacio de mundo con shaders gráficos

Se han realizado pruebas aplicando la lógica de desplazamiento en espacio de mundo a un *shader* gráfico. Al no poder aprovechar la ventaja de saber en qué dirección buscar el píxel apropiado, la búsqueda realizada debía ser cuadrática y el rendimiento bajaba en un ordenador de sobremesa con tarjeta gráfica de última generación por debajo de los 30 fotogramas por segundo con un *kernel* de búsqueda de 40x40, dando además un resultado con grandes cantidades de píxeles perdidos.

6.10.3. Completar con el ojo opuesto

Una de las propuestas para mejorar este algoritmo, es utilizar la textura del ojo opuesto como herramienta para llenar huecos, que de otra manera quedarían vacíos, utilizando las mismas operaciones que se venían usando, pero aplicando pequeñas modificaciones para el nuevo ojo. Se ha realizado un esbozo de esta tecnología pero no se ha podido llevar a cabo por falta de tiempo.

7. Conclusiones

El vídeo y la fotografía es hoy en día una parte imprescindible del entretenimiento en realidad virtual.

Sin embargo, tanto el vídeo como la fotografía son modelos de entretenimiento muy pobres en comparación con las experiencias interactivas que se ofrecen utilizando modelos tridimensionales. Por todo esto es importante preocuparse por mejorar estos medios y trabajar por ofrecer mejores soluciones.

Durante este proyecto se han estado trabajando con soluciones que hacen uso de algoritmos con una gran carga de trabajo, con su consecuente pérdida de rendimiento en los dispositivos de menores especificaciones. Para poder encontrar el algoritmo que mejor funciona hay que probar muchas implementaciones e ir descartando las soluciones menos prácticas.

Durante el desarrollo de las diferentes aproximaciones, encontré una serie de problemas a los que tuve que dar solución. Por ejemplo, utilizar un shader gráfico no era una opción viable para el programa.

Por otro lado, el código tiene partes donde se encontraron problemas que habitualmente se resuelven depurándolos, como es el caso de la función de desplazamiento en espacio de mundo. La depuración de shaders que se ejecutan sobre hardware gráfico es muy complicada de llevar a cabo y generalmente debe ser resuelta sacando colores por pantalla, cosa que no siempre es todo lo intuitivo que cabría esperar.

El objetivo propuesto tiene mucho fondo, pues el último propósito no estaba planteado para conseguirlo en el plazo de un trabajo de fin de máster,

sino que todavía queda mucho trabajo por hacer en este campo. Aún así creo que se ha conseguido realizar una aproximación que permita continuar el desarrollo con nuevas características como el rellenado de huecos mediante imágenes que se encuentren paralelas a la actual y la sustitución de la vista actual por estas imágenes si el desplazamiento es superior a cierta cantidad.

Este trabajo a su vez ha requerido un gran componente de investigación, pues hay muchas y grandes empresas como Disney, Facebook y Google trabajando para hacer posible esta idea. Al ser empresas pioneras en tecnología, lo hace un campo de trabajo muy complicado.

Como resultado de este trabajo puedo decir que las tecnologías de video e imagen en realidad virtual tienen mucho camino por recorrer pero que están encaminadas para que los creadores de contenido tengan una vía más completa de realizar sus obras.

Todavía queda mucho por hacer pero creo que no está muy lejos el día que, mientras tenemos unas gafas de realidad virtual puestas, podamos movernos alrededor de los personajes de nuestra película favorita y podemos sentirnos parte de ella.

Bibliografía

- [1] Diego Bezares. Charla gamelab 2016 . estereoscopía en realidad virtual. r.v v.r vr,. URL <https://www.youtube.com/watch?v=70gqUSPTriE>.
- [2] Diego Bezares. Video vr en daydream y plataformas de r.v móviles, . URL <https://www.youtube.com/watch?v=y2mkVQ57-90&t=634s>.
- [3] Adobe Creative Cloud. Projectsidewinder: Adobe max 2017 (sneak peeks) | adobe creative cloud. URL <https://www.youtube.com/watch?v=HSXMs2wnNc4>.
- [4] Javier de Matías Bejarano. *TÍTULO: TÉCNICAS DE FOTOGRAMETRÍA Y VISIÓN POR COMPUTADOR PARA EL MODELADO 3D DE ESTRUCTURAS GEOMORFOLÓGICAS DINÁMICAS*. PhD thesis, Universidad de Extremadura, 2013.
- [5] Paul Debevec. Experimenting with light fields. URL <https://www.blog.google/products/google-ar-vr/experimenting-light-fields/>.
- [6] Pedro Fernando Gomez Fernandez. Parallax test. URL <https://www.youtube.com/watch?v=LPfYrwvZHsQ>.
- [7] Josh Gladstone. Use depth maps to create 6 dof in unity. URL <https://www.immersiveshooter.com/2018/01/23/use-depth-maps-create-6dof-unity/>.
- [8] Facebook Inc. Casual 3d capture, . URL <https://developers.facebook.com/videos/f8-2017/casual-3d-capture/>.
- [9] Facebook Inc. The evolution of dynamic streaming, . URL <https://developers.facebook.com/videos/f8-2017/the-evolution-of-dynamic-streaming/>.

- [10] Facebook Inc. Oculus go: Designing for media and entertainment, . URL <https://developers.facebook.com/videos/f8-2018/oculus-go-designing-for-media-and-entertainment/>.
- [11] Google Inc. Rendering omni-directional stereo content, . URL <https://developers.google.com/vr/jump/rendering-ods-content.pdf>.
- [12] Ben Lang. Lytro is positioning its light field tech as vr's master capture format. URL <https://www.roadtovr.com/lytro-is-positioning-its-light-field-tech-as-vrs-master-capture-format/>.
- [13] OTOY. Otoy and facebook release revolutionary 6dof video vr camera pipeline and tools. URL <https://home.otoy.com/otoy-facebook-release-revolutionary-6dof-video-vr-camera-pipeline-tools/>.
- [14] Christopher Schroers, Jean-Charles Bazin, and Alexander Sorkine-Hornung. An omnistereoscopic video pipeline for capture and display of real-world vr. *ACM Trans. Graph*, page 13, August 2018. doi: 10.1109/TVCG.2018.2794071. URL <http://richardt.name/publications/parallax360/>.