Universidad Zaragoza

Trabajo Fin de Máster

Simulador de imágenes omnidireccionales fotorealistas para visión por computador

Photo-realistic and Omnidirectional image simulator for computer vision

Autor

Samuel Bruno Berenguel Baeta

Director/es

Jose Jesús Guerrero Campo
Jesús Bermúdez Cameo

Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza
2019

Resumen

La motivación de este proyecto es la necesidad de bases de imágenes omnidireccionales y panorámicas para visión por computador. Su elevado campo de visión permite obtener una gran cantidad de información del entorno a partir de una única imagen. Sin embargo, la distorsión propia de estas imágenes requiere desarrollar algoritmos específicos para su tratamiento e interpretación. Además, un elevado número de imágenes es imprescindible para el correcto entrenamiento de algoritmos de visión por computador basados en aprendizaje profundo. La adquisición, etiquetado y preparación de estas imágenes de forma manual con sistemas reales requiere una cantidad de tiempo y volumen de trabajo que en la práctica limita el tamaño de estas bases de datos. En este trabajo se propone la implementación de una herramienta que permita generar imágenes omnidireccionales sintéticas fotorrealistas que automatice la generación y el etiquetado como estrategia para aumentar el tamaño de estas bases de datos. Este trabajo se apoya en los entornos virtuales que se pueden crear con el motor de videojuegos Unreal Engine 4, el cual se utiliza junto a uno de sus plugin, UnrealCV. A partir de estos entornos virtuales se construyen imágenes de una variedad de cámaras omnidireccionales y 360º con calidad fotorrealista. Las características del entorno permiten además generar imágenes de profundidad y semánticas. Al hacerse todo de forma virtual, se pueden controlar los parámetros de adquisición de la cámara y las características del entorno, permitiendo construir una base de datos con un etiquetado automático sin supervisión. Conocidos los parámetros de calibración, posición y orientación de la cámara y la distribución del entorno y sus objetos, se puede conseguir el “ground truth” para diversos algoritmos de visión. Con las imágenes e información que se dispone, se pueden evaluar algoritmos de extracción de rectas en imágenes dióptricas y catadióptricas, obtención de *layouts* en panoramas o métodos de reconstrucción 3D como la localización y mapeado simultáneos (SLAM).

Abstract

The aim of this project is the need of large data sets of omnidirectional and 360° images for computer vision. Its wide field of view allows to obtain a great amount of information of an environment from only one image. However, the distortion of these images requires developing specific algorithms for the treatment and interpretation. Besides, a high number of images are essential for the correct training of computer vision algorithms based on deep learning. When dealing with real systems the acquisition, the manual labelling and the set up of these images takes a big amount of time and work. In practice that reduces the volume in these data sets. In this work, a tool allowing the generation of omnidirectional, synthetic, photorealistic images is proposed. This approach allows us enlarging omnidirectional data sets automatizing the acquisition and labelling process. This work is based on virtual environments made with the game engine Unreal Engine 4, which is used in combination with UnrealCV plugin. In these virtual environments, images are built from a diversity of omnidirectional and 360° cameras. The environment features allow to generate depth and semantic images. Since everything is virtually made, the acquisition parameters of the camera and the features of the environment can be controlled, been easy to build a labelled data set without supervision. Known the calibration parameters, pose and orientation of the cameras and the position of objects and layout of the rooms, ground truth for computer vision algorithms can be achieved. With this images and information available, several algorithms can be tested as line extraction from dioptric and catadioptric images, layout reconstruction from panoramas or 3D-reconstruction methods just as Simultaneous Location and Mapping (SLAM).

Índice general

1 Introducción **1**
1.1 Motivación 1
1.2 Estado del arte 1
1.3 Objetivo 3

2 Unreal Engine 4 **5**
2.1 Plugin: UnrealCV 5
2.2 Sistema de coordenadas 7

3 Modelos de Cámara **9**
3.1 Cámaras centrales 10
3.1.1 Modelo Equirectangular 10
3.1.2 Modelo cilíndrico 12
3.1.3 Cámaras ojo de pez 13
3.1.4 Cámaras catadióptricas 15
3.1.5 Modelo de Scaramuzza 17
3.2 Cámaras no centrales 19
3.2.1 Panorama 19
3.2.2 Cámaras catadióptricas 22

4 Simulador de cámaras centrales **26**
4.1 Adquisición de imágenes 27
4.1.1 Límites de imagen 32
4.2 Composición de imágenes 33
4.2.1 Equirectangular 33
4.2.2 Cilíndrico 35
4.2.3 Ojo de pez 36
4.2.4 Catadióptricas 38
4.2.5 Scaramuzza 40
4.2.6 Intersección rayo-*cubemap* 41

5 Simulador de cámaras no centrales **44**
5.1 Panorama no central 44
5.2 Catadióptricos no centrales 46

6 Evaluaciones 49
6.1 Evaluación . . . 49
6.1.1 Corners for Layout . . . 49
6.1.2 Toolbox de calibración . . . 52

7 Conclusiones 55
7.1 Conclusión . . . 55
7.2 Trabajo Futuro . . . 56

A Comandos UnrealCV 60

B Implementación Panorama no central en Python 65

C Implementación Catadióptricos no centrales en Python 69

D Máscaras binarias 75

E Simulador 78

Capítulo 1

Introducción

1.1 Motivación

La necesidad de imágenes omnidireccionales y de 360º para visión por computador ha sido la principal motivación para el desarrollo de este proyecto. La gran cantidad de información que se puede obtener de una sola imagen de 360º le permite ser muy útil para su utilización en algoritmos de visión por computador. Este tipo de imágenes presentan distorsiones que dificultan su tratamiento, haciendo difícil su creación y su utilización posterior, puesto que no siguen los modelos de proyección convencionales. Sin embargo, obtener toda la información del entorno con la obtención de una única imagen compensa la dificultad de su procesado.

El rápido desarrollo de la visión por computador y los algoritmos basados en deep-learning han puesto en evidencia la falta de recursos visuales para el funcionamiento con imágenes omnidireccionales. Un buen entrenamiento de estos algoritmos requiere tener grandes bases de imágenes. Las bases de imágenes existentes para visión omnidireccional son muy limitadas en tamaño debido al trabajo que requiere la adquisición, etiquetado y post-procesado de las imágenes. Además, solo cubren parte de los distintos tipos de sistemas de adquisición omnidireccionales. El trabajo manual necesario para el etiquetado de cada una de las imágenes adquiridas y las posibilidades de error que ello conlleva, impide tener bases de imágenes ricas en variedad de escenas, objetos y tipos de cámara. Este problema motiva la aparición de herramientas que permitan generar etiquetados de forma automática y da paso a la generación de bases de imágenes sintéticas, reduciendo el tiempo y esfuerzo que requiere la adquisición de imágenes reales.

Además, la aparición de entornos virtuales fotorrealistas abre una posibilidad a esta generación sintética de bases de imágenes. Unreal Engine 4 es un ejemplo de entorno virtual en el que se pueden crear escenarios de una amplia diversidad y con una calidad difícilmente distinguible de la realidad. La figura 1.1 muestra un ejemplo de la capacidad de renderizado de este motor gráfico, donde se compara una imagen real, extraída de Tripadvisor, y una imagen sintética obtenida de Unreal Engine 4.

1.2 Estado del arte

El rápido desarrollo de la visión por computador ha impulsado la generación de bases de imágenes sintéticas para proyectos de investigación. Dada la gran cantidad de información que se

(a) (b)

Figura 1.1: Imagen real y sintética. El nivel de realismo de las imágenes sintéticas ya permite su uso para visión por computador

necesita en algoritmos basados en deep-learning, han aparecido una gran cantidad de bases de imágenes sintéticas. El desarrollo de videojuegos y el aumento de la calidad de sus gráficos han permitido que se puedan adquirir imágenes fotorrealistas de entornos virtuales. En trabajos como [1], [2], [3], [4] y [5] se parte de videojuegos actuales, con gráficos fotorrealistas para obtener imágenes y su etiquetado semántico con precisión de píxel. Estos trabajos se centran en imágenes y vídeos de entornos exteriores para ser utilizados en algoritmos de conducción autónoma y SLAM. En lugar de utilizar videojuegos, los trabajos [6], [7] y [8] utilizan directamente motores gráficos como Unreal Engine 4 para la obtención de imágenes y vídeos desde el que extraen la información semántica. Sin embargo, todas estas bases de imágenes y simuladores únicamente obtienen imágenes en entornos exteriores, muy enfocado a conducción autónoma y con cámaras perspectivas convencionales. Para entornos de interior, [9] y [10] proveen una variedad de imágenes fotorrealistas, etiquetado semántico e imágenes de profundidad de forma automática. Sin embargo, todas las bases de imágenes anteriores, tanto de exterior como de interior, solo permiten obtener imágenes perspectivas.

Las imágenes omnidireccionales son muy útiles en visión por computador y robótica. Sin embargo, la distorsión que se produce en la proyección complica el proceso de calibración [11] [12]. Además, la diversidad de sistemas de adquisición (catadióptricos, dióptricos, panorámicos) requiere el desarrollo de algoritmos específicos [13] [14].

Bases de imágenes de cámaras omnidireccionales sintéticas existen pocas en la actualidad. Bases de imágenes como Std-2D-3D [15] y SUN360 [16] presentan imágenes panorámicas de entornos cerrados, con etiquetados mayormente manuales. En [17] se encuentran vídeos con segmentación semántica de cámaras de ojo de pez, como ejemplo de otro tipo de imagen omnidireccional.

En este proyecto se propone introducir una herramienta que permita obtener imágenes fotorrealistas, con etiquetado semántico de objetos e imágenes de profundidad de una variedad de cámaras omnidireccionales. Se van a considerar, no solo panoramas equirectangulares, sino panoramas no centrales, distintos modelos de proyección de cámaras de ojo de pez e imáge-

nes catadióptricas con distintos espejos, que generan proyecciones centrales y no centrales. Este proyecto engloba las distintas opciones que se encuentran en el estado del arte, añadiendo una mayor variedad de modelos de cámaras de 360º.

1.3 Objetivo

El objetivo de este proyecto es crear una herramienta para poder generar imágenes de cámaras omnidireccionales dióptricas, catadióptricas y panorámicas centrales y no centrales, con las que crear bases de imágenes con información semántica y de profundidad. La creación de escenarios virtuales permite utilizar entornos que vayan a facilitar el ajuste, aprendizaje o evaluación de los algoritmos de visión por computador, facilitando el trabajo a investigadores y permitiendo la creación de bases de imágenes especializadas para tareas concretas.

Las imágenes sintéticas se adquieren en un entorno virtual creado con Unreal Engine 4. La interacción con el entorno se realiza a través de un plugin, UnrealCV, con el cual se puede crear una comunicación entre el entorno virtual y el usuario a través de un terminal o script en Python. La decisión de utilizar este motor gráfico se debe a la alta calidad de renderizado, llegando a obtener imágenes fotorrealistas de entornos virtuales. También se ha escogido este motor gráfico puesto que es muy conocido y permite crear entornos propios, haciendo muy versátil su utilización en otros trabajos. Sin embargo, el conjunto UE4 + UnrealCV está limitado a la adquisición de imágenes perspectivas convencionales. En este trabajo se va a extender el marco a modelos proyectivos que engloban todo tipo de cámaras omnidireccionales.

En este proyecto se ha trabajado con dos grupos de sistemas de adquisición de imágenes omnidireccionales, *Cámaras centrales* y *Cámaras no centrales*. Las cámaras centrales se caracterizan por tener un centro óptico único al que convergen todos los “rayos” de luz. En las cámaras no centrales el centro óptico puede ser distinto para cada pixel de la imagen, quedando distribuido a lo largo de una trayectoria o superficie.

En particular, los sistemas centrales considerados son dos modelos catadióptricos, compuestos por una cámara perspectiva y un espejo: *paracatadióptrico* e *hipercatadióptrico*; cuatro modelos dióptricos, basados en distorsión por lentes de ojo de pez: *equi-angular*, *estereográfica*, *ortogonal* y *equi-ángulo sólido*; y panoramas de 360º *equirectangular* y *cilíndrico*. Por otro lado, se ha considerado dos modelos catadióptricos no centrales, con espejos cónico y esférico; y un panorama equirectangular no central.

La estructura de este trabajo de fin de máster comienza con la descripción de este entorno virtual y como se interactúa con él desde Python. Posteriormente se expondrán los modelos matemáticos utilizados para modelar las distintas proyecciones de las cámaras, como se han implementado en Python y su implementación con el entorno virtual. Por último se realizará una evaluación de los resultados obtenidos utilizando algoritmos de visión por computador. En particular, se ha realizado una evaluación de un sistema de detección de *layouts* de habitaciones basado en deep-learning utilizando panoramas equirectangulares [13]. Por otro lado, se ha evaluado una toolbox de calibración y extracción de rectas para las imágenes catadióptricas y ojo de pez [18]. Finalmente se expondrán las conclusiones de las evaluaciones realizadas y se propondrá un trabajo futuro para la continuación en la misma línea de este.

El desarrollo de este proyecto se ha realizado en el marco de un grupo de investigación, para dar soporte a una serie de herramientas para imágenes omnidireccionales y de 360º. El alcance

del proyecto consiste en la creación de un simulador que compone imágenes fotorrealistas a partir de sus modelos de proyección, donde se han considerado modelos dióptricos de ojo de pez, catadióptricos y panoramas centrales y no centrales. En dicho simulador se han desarrollado funciones con las que conseguir un etiquetado automático y un *ground truth* con los que poder utilizar las herramientas de evaluación existentes.

Capítulo 2

Unreal Engine 4

Unreal Engine es un motor gráfico desarrollado por la compañía EpicGames, diseñado inicialmente para videojuegos en primera persona. La capacidad inicial del motor era escasa, aunque acorde a su época. Año a año ha ido mejorando y han ido apareciendo nuevas generaciones de este motor gráfico, hasta llegar al Unreal Engine 4 (UE4), última generación de este motor el cual dispone de una amplia variedad de herramientas y está disponible para un gran rango de plataformas digitales.

El atractivo de UE4 es que está programado en C++, lenguaje muy conocido en el mundo de la informática. Además, tiene un código abierto para todo aquel que quiera utilizarlo. Esto ha permitido que se hayan desarrollado una gran variedad de Plugins y programas externos de soporte al motor principal, como el plugin UnrealCV.

Otro de sus principales atractivos para su utilización es la potencia gráfica que desarrolla, permitiendo obtener entornos virtuales de una calidad realista. La posibilidad de generar y trabajar con gráficos realistas en tiempo real permite hacer simulaciones y generar bases de imágenes virtuales que puedan ser usadas en algoritmos de visión por computador, por ejemplo, en entrenamiento de redes neuronales. En este proyecto se ha diseñado un simulador que trabaja en UE4 para poder generar bases de imágenes realistas de distintas cámaras fotográficas. La posibilidad de obtener esta información a gran escala y sin un trabajo de campo hace que se agilice y facilite el trabajo de investigadores en el campo de la visión por computador y learning.

EpicGames ofrece una interfaz gráfica de UE4, así como la posibilidad de trabajar desde el código. Desde la perspectiva de este proyecto, se ha trabajado en la interfaz gráfica, a través de un editor de escenarios. En este proyecto se ha utilizado el editor para conocer los sistemas de referencia de los entornos, así como sistemas de movimiento y giros de los jugadores, o cámaras, dentro del entorno. En la documentación del editor, la cual se puede encontrar en [19], se define cada una de sus funciones.

2.1 Plugin: UnrealCV

Como se ha comentado, UE4 tiene una gran variedad de plugins que le permite mucha versatilidad. UnrealCV [10], es un plugin diseñado para ser utilizado en visión por computador. Este plugin permite una comunicación directa con el editor y los proyectos de UE4 para poder definir cámaras y capturar imágenes de los escenarios diseñados. El control de la cámara se puede hacer

de forma manual desde el editor de UE4, introduciendo los comandos del plugin directamente en la línea de comando que se habilita. Este proceso es útil para un primer aprendizaje en la utilización de esta herramienta, pero no es muy eficiente si se quiere crear una base de datos o realizar capturas en distintos puntos de la escena. Este problema se puede solventar utilizando *scripts* externos que inicien una comunicación con UE4 a través de este plugin.

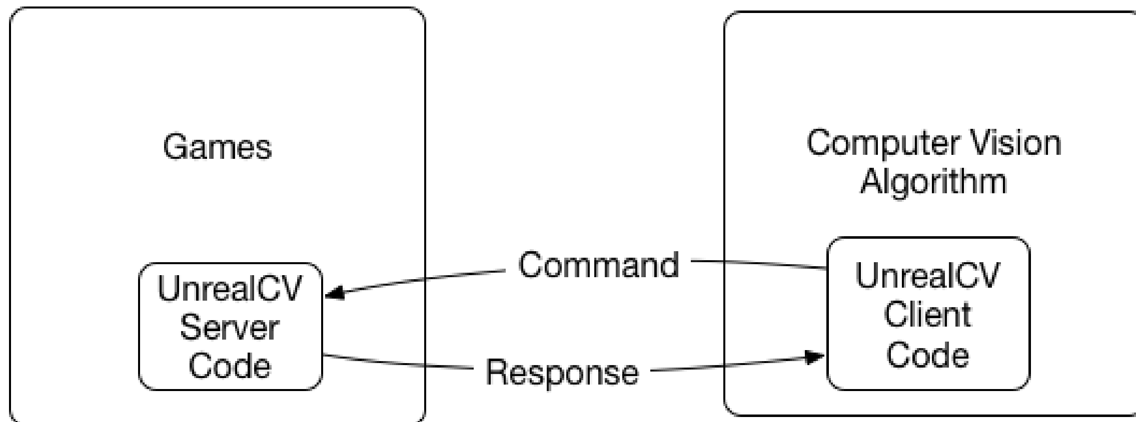


Figura 2.1: Comunicación entre UE4 y un programa externo a través de UnrealCV, obtenido de [10]

Utilizando *Python* como lenguaje de programación, y la librería propia del plugin UnrealCV para dicho lenguaje, se puede hacer un control sobre la cámara dentro del entorno desde un programa externo.

La comunicación entre el programa y el entorno de UE4 se realiza como se ve en la figura 2.1, donde se tiene una comunicación cliente-servidor. El cliente, programa de control externo, manda una serie de comandos al servidor, entorno de UE4, que procesa la orden y la ejecuta, dando una respuesta de vuelta al cliente cuando dicha ejecución finaliza. De esta forma, se pueden crear programas de control complejos para obtener resultados de forma más rápida que si se hicieran de forma manual.

Una vez establecida la comunicación entre el programa y el entorno de UE4, este plugin contiene distintas funciones. Lo principal que se ha usado son las distintas formas de captura de imagen: *iluminación, semántica* y *profundidad*. Desde un mismo programa se puede obtener los tres tipos de captura del escenario, obteniéndose mucha información del entorno a través de las imágenes en poco tiempo. En la sección 4.1 se muestra qué imágenes se obtienen de estos modos y cómo se han implementado dentro del simulador.

Otra función muy útil es la posibilidad de ocultar objetos. Conociendo qué objetos ocultar y cuáles no, se puede obtener un *layout* de los distintos entornos en los que se trabaja. El inconveniente de esta función es el tratamiento de la iluminación por parte de UE4, puesto que la hace en un pre-procesado. Al no compilar la iluminación de forma continua, cuando un objeto desaparece, deja la sombra que proyectaba cuando si que está, impidiendo tener imágenes iluminadas realistas. Sin embargo, es muy útil para imágenes semánticas o de profundidad, puesto en estas no se tienen en cuenta los focos de luz.

En el anexo A se ha incluido el capítulo de la documentación de UnrealCV, [10], donde se exponen los comandos que se pueden utilizar con el plugin y que función desempeñan.

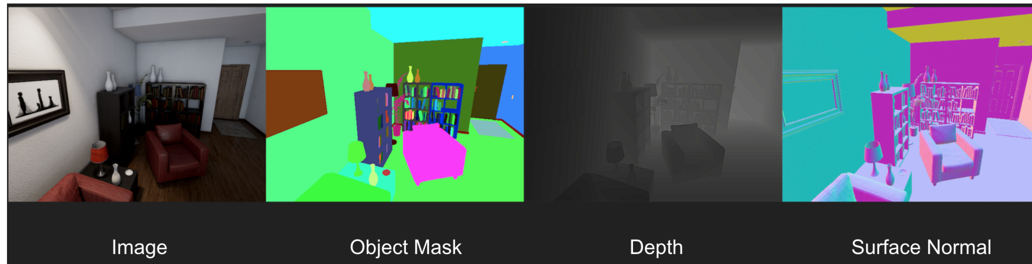
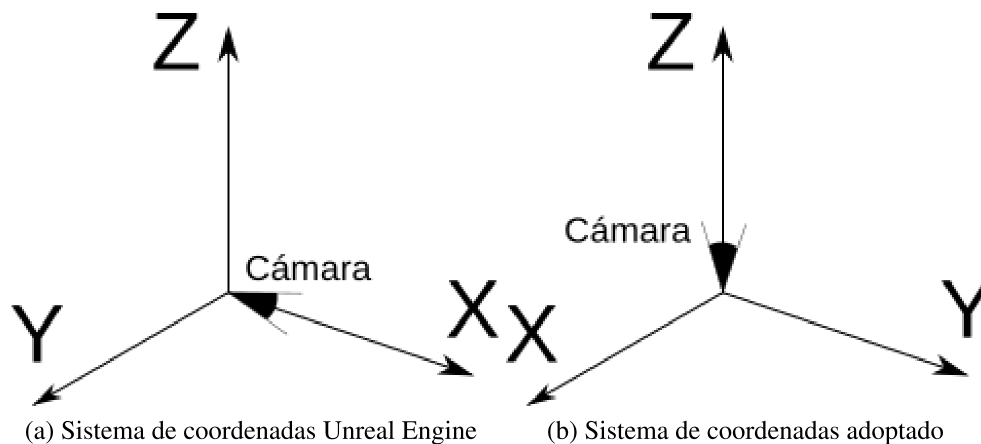


Figura 2.2: Ejemplo de imágenes que se pueden obtener con el plugin UnrealCV. Imagen obtenida en [10]

2.2 Sistema de coordenadas

Una de las partes más importantes que se necesitaban conocer sobre UE4 es el sistema de coordenadas. Puesto que se va a trabajar con modelos matemáticos basados en posiciones en el espacio y vectores, tener presente cual es el sistema de coordenadas del entorno es fundamental.



(a) Sistema de coordenadas Unreal Engine (b) Sistema de coordenadas adoptado

Figura 2.3: a) Sistema de coordenadas de Unreal Engine 4; b) Sistema de coordenadas que se ha utilizado en este proyecto

Como en la mayoría de simuladores gráficos, los ejes del sistema de coordenadas están definidos de forma levógira, como se puede ver en la figura 2.3. Este tipo de sistema de coordenadas resulta muy útil para videojuegos, puesto resulta más intuitivo para el movimiento de los jugadores. Sin embargo, los modelos matemáticos presentados se basan en sistemas dextrógiros, por lo que se realiza un cambio de sistema de coordenadas, transparente al usuario, para poder hacer una aplicación directa de las ecuaciones. De esta forma, un usuario que utilice el simulador propuesto en el proyecto, puede definir sus parámetros en un sistema de coordenadas dextrógiro, el más habitual cuando se trata de matemáticas, y el propio simulador hará el cambio a levógiro para compatibilizar los cálculos.

Además, la definición de los giros sobre los ejes también son distintos a los usuales. Al igual que con el sistema de coordenadas, su definición está hecha para que sea más intuitiva para el jugador de videojuegos, pero presenta alguna dificultad a nivel matemático. Su distinta definición, uno levógiro y dos dextrógiros, hace que se tenga que tener cuidado con su utilización en la aplicación de matrices de rotación. En la figura 2.4 se ve como están definidos estos giros sobre los ejes. A efectos de este proyecto, la cámara virtual de UnrealCV está orientada en el eje $+X$, mientras que la cámara del simulador está orientada en el eje $+Z$. Por otro lado, en UE4 los giros *roll, pitch, yaw* están definidos como como (θ, φ, ψ), según se puede ver en la figura 2.4. Dentro del editor, estos giros tienen unos límites y origen definidos, distintos para cada ángulo

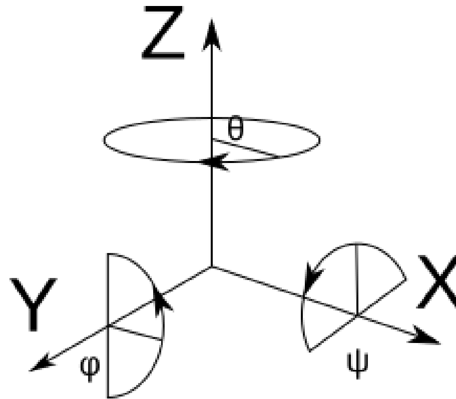


Figura 2.4: Giros en torno a los ejes de Unreal Engine 4

de giro. En la figura 2.4 se ha definido el campo que abarca cada ángulo de giro así como su origen, definido por una línea en el arco. Estos límites se han tenido que tener en cuenta para la obtención de las imágenes, puesto marcan restricciones en la orientación de la cámara respecto a los modelos que se proponen.

El cambio de un sistema coordenado a otro se realiza en dos pasos. Por un lado, el cambio de sistema dextrógiro a levógiro se consigue invirtiendo el eje Y, es decir, sustituyendo $+Y$ por $-Y$. Por otro lado, el cambio de orientación de la cámara se consigue aplicando la matriz de rotación (2.1).

$$R_{to-cam} = \begin{pmatrix} 0 & 0 & -1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \tag{2.1}$$

Como conclusión, UE4 es una herramienta gráfica muy potente, con una gran variedad de extensiones y plugins que le permite realizar entornos y dinámicas complejas y realistas. La potencia unida a la posibilidad de utilizarlo libremente en educación e investigación es lo que ha motivado su utilización en este proyecto, aunque aún queda mucho trabajo que se puede realizar con esta herramienta. Al final del proyecto se expondrá alguna línea de trabajo futuro.

Capítulo 3

Modelos de Cámara

En este capítulo se van a definir los distintos modelos de proyección que se han implementado en el simulador de imágenes fotorrealistas. Se van a separar en dos tipos de cámaras: *Cámaras centrales* y *Cámaras no centrales*. Dentro de cada uno de los tipos de cámaras encontramos distintos modelos matemáticos para diferentes configuraciones de cámaras. Dentro de las cámaras centrales encontramos:

- Imágenes panorámicas
  - Imágenes equirectangulares
  - Imágenes cilíndricas
- Imágenes dióptricas: modelos de cámaras de ojo de pez
  - Equi-angulares
  - Estereográficas
  - Equi-ángulo sólido
  - Ortogonal
- Imágenes catadióptricas
  - Espejos parabólicos: *Para-catadióptricas*
  - Espejos hiperbólicos: *Hiper-catadióptricas*

A su vez, para cámaras no centrales también se han realizado distintos modelos e implementaciones:

- Imagen panorámica no central
- Imágenes catadióptricas
  - Espejos esféricos
  - Espejos cónicos

Estos modelos y conceptos se explicarán en las secciones siguientes. En la segunda parte de este proyecto se explicará cómo se han implementado estos modelos dentro del simulador.

3.1 Cámaras centrales

Las cámaras centrales son aquellas donde la proyección de rayos converge en un punto. Este tipo de cámara es el más utilizado, y por esto se han buscado un mayor número de modelos e implementaciones de este tipo.

Antes de introducir los modelos de las cámaras, se va a discutir un detalle de la implementación. Desde el entorno virtual UE4 + UnrealCV solo se pueden obtener imágenes perspectivas, por lo que es necesario buscar una solución con la que se pueda obtener la información necesaria del entorno para componer las imágenes omnidireccionales. La solución adoptada es la que se puede ver en la figura 3.1, el *cubemap*. Esta solución consiste en obtener 6 imágenes perspectivas desde una posición en el espacio, de forma que se abarque la totalidad del entorno alrededor de este punto. Dada la característica de las cámaras centrales, la localización desde donde se toman estas imágenes, centro óptico de la cámara virtual perspectiva de UnrealCV, será a su vez el centro óptico de las cámaras omnidireccionales, pudiendo ser aprovechado el mismo *cubemap* para todos los modelos de cámara central.

Figura 3.1: Ejemplo de *cubemap* obtenido con UE4+UnrealCV

De esta forma, con 6 imágenes que sean capaces de cubrir la totalidad del entorno en una posición dada se optimiza el proceso, reduciendo enormemente el tiempo de computación y la memoria necesaria para la construcción de las imágenes omnidireccionales de dicha posición.

3.1.1 Modelo Equirectangular

Las imágenes equirectangulares son imágenes omnidireccionales de 360 grados donde se puede ver todo el entorno alrededor de un punto, donde se situaría el observador o cámara. Este tipo de

imágenes es muy útil para poder obtener de en un simple vistazo una idea de cómo es el entorno de la localización actual, sin tener que cambiar la orientación o posición de la cámara.

Como todas las imágenes que se van a exponer, estas imágenes tienen una distorsión que puede hacer difícil su comprensión para el ojo humano al principio. Para este modelo esta deformación se da principalmente en los extremos superior e inferior. La imagen equirectangular modela el entorno en una esfera que luego es desplegada en la imagen rectangular final, como en el ejemplo de la figura 3.2.

Figura 3.2: Imagen equirectangular

Este modelo tiene una implementación directa desde el entorno virtual hasta la imagen final, puesto que no se tiene que considerar la física de ninguna lente o espejo.

En el caso de este simulador, se ha seguido un proceso de construcción inverso al modelo, es decir, se han tomado las coordenadas de la imagen final, vacía, para buscar la información del píxel correspondiente en el entorno. Se considera *inverso* porque para la obtención de la imagen se realizan el camino contrario al lógico en la definición de dicha imagen. En lugar de partir de una esfera donde está mapeado el entorno, se parte de la imagen final, mapeada en unas *coordenadas píxel*, desde la cual se pasa a un sistema de coordenadas esféricas, desde donde se buscará la información de píxel en el entorno. En las ecuaciones (4.2) y (4.3) se puede ver el paso de un sistema de coordenadas a otro, mientras que en la imagen posterior estos sistemas de coordenadas están definidos en la imagen equirectangular.

$$\theta = \left(\frac{2x}{x_{max}} - 1\right)\pi \tag{3.1}$$

$$\phi = \left(1/2 - \frac{y}{y_{max}}\right)\pi \tag{3.2}$$

La obtención de la información del píxel no se hace tomando información directamente del entorno virtual, puesto que este proceso requeriría un gran esfuerzo computacional. En este caso, se ha hecho una modelización de la esfera a través del *cubemap*.

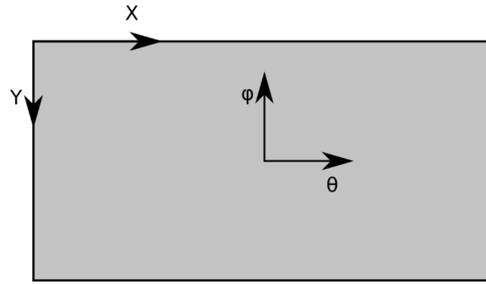


Figura 3.3: Coordenadas esfericas en imagen equirectangular

3.1.2 Modelo cilíndrico

En el modelo cilíndrico, en lugar de proyectar el entorno en una esfera, como se hace en el modelo equirectangular, el entorno queda proyectado en la pared lateral de un cilindro, dejando las bases del cilindro sin proyectar.

En este tipo de imágenes se deben definir los parámetros de la imagen final, puesto que ya no abarca la totalidad de la escena. Como norma general, y como se ha hecho en este proyecto, se definen dos ángulos: el campo de vista horizontal y el campo de vista vertical. Destacar que si lleváramos estos parámetros hasta su extremo, 360º para el campo de vista horizontal y 180º para el vertical, se obtendría una imagen similar a la equirectangular.

Al igual que en el modelo equirectangular, la imagen se obtiene partiendo de la imagen vacía y buscando la información del píxel correspondiente en el *cubemap* del escenario. La diferencia es el cambio de coordenadas para obtener esta información del píxel. Las *coordenadas píxel* cambian ligeramente respecto al modelo anterior, debido a la diferente forma de proyectar la escena. Buscando una homogeneización de las estructuras de los programas y matemática utilizada, el cambio a las coordenadas del entorno busca una similitud con el anterior modelo, definiendo dos ángulos en coordenadas esféricas para buscar la información del píxel en el *cubemap*. La principal diferencia que se encuentra es que partimos de unas coordenadas polares de las que se obtiene posteriormente las coordenadas esféricas con los ángulos que se necesita. En las ecuaciones (3.3), (3.4) y (3.5) se define este cambio desde *coordenadas píxel* hasta los ángulos de las coordenadas esféricas que se utilizan.

$$\theta = \left(\frac{2x}{x_{max}} - 1\right)\frac{FOV_h}{2} \tag{3.3}$$

$$z = 1 - \frac{2y}{y_{max}} \tag{3.4}$$

$$\phi = \arcsin z \tag{3.5}$$

Como en el modelo equirectangular, se obtienen dos ángulos con los que se obtendrá la información necesaria del entorno. Puesto que se ha modelizado el entorno a partir de las mismas imágenes, es decir, solo se realiza una adquisición de imágenes por cada posición en la escena, se utiliza el mismo *cubemap* para la misma posición dentro de la escena. Como ya se ha comentado, esto reduce la memoria necesaria y acelera la obtención de las imágenes finales.

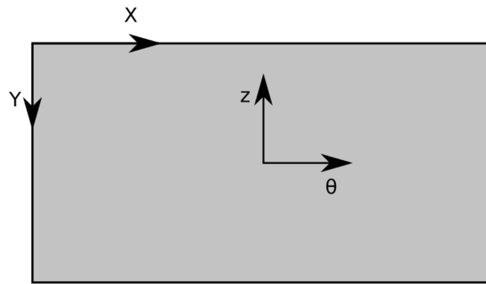


Figura 3.4: Coordenadas cilindricas en imagen

Figura 3.5: Imagen cilíndrica: $FOV_{horizontal} = 180^{o}$, $FOV_{vertical} = 120^{o}$

3.1.3 Cámaras ojo de pez

En este apartado se va a explicar el modelo para cámaras ojo de pez. La característica de este tipo de cámaras es el gran campo de vista que se tiene a partir de una única adquisición. Hasta ahora, las cámaras que se habían expuesto son resultado de la composición de un número de imágenes para conseguir la imagen omnidireccional final. En este tipo de cámara se pueden conseguir ángulos de visión de hasta 220° con una única captura. Este tipo de cámara puede resultar muy útil para visualizar una escena con un equipamiento mínimo, puesto que solo sería necesario utilizar dos cámaras, trabajando a la vez, para tener una visión completa del entorno.

Para el modelado de este tipo de cámara se ha tomado un modelo unificado para cámaras con simetría de revolución, [20] y [18].

El modelo de simetría de revolución que se explica consiste en la proyección de los rayos 3D del entorno en una esfera de radio unidad, desde donde se proyectan a través de una función $\hat{r} = h(\phi)$ que relaciona el radio en la imagen proyectada con el ángulo de incidencia del rayo 3D. Esta función depende de la lente y modelo proyectivo de la cámara que se esté usando.

La distorsión que provoca la función $\hat{r} = h(\phi)$ hace que la imagen final quede circular, tal y como la lente de la cámara. Esta función depende de la lente y modelo de proyección que se utilice. En este trabajo se han modelado 4 modelos de proyección distintos: *equiangular*,

Figura 3.6: Ejemplo imagen de ojo de pez equi-angular

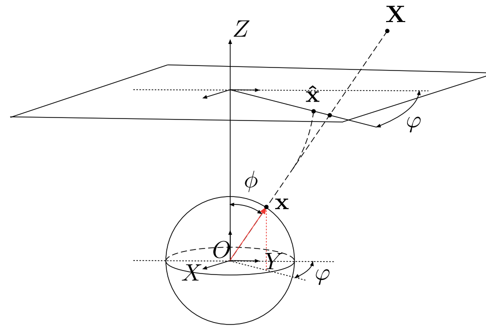


Figura 3.7: Esquema modelo cámaras ojo de pez

estereográfico, *ortogonal* y *equi-ángulo sólido*.

Este modelo se define en coordenadas polares, definidas por $(\hat{r}, \hat{\theta})$. Posteriormente se define la función $h(\phi)$ para cada uno de los tipos de modelos de proyección, donde se relaciona el ángulo de incidencia en la esfera ϕ con la componente radial $\hat{r}$. Puesto que se ha definido el modelo como un sistema de simetría de revolución, se define la componente angular $\hat{\theta} = \varphi$, dejando las distintas proyecciones depender de la componente radial.

Equi-angular	Estereográfica	Ortogonal	Equi-angulo solido
$\hat{r} = f\phi$	$\hat{r} = 2f\tan(\frac{\phi}{2})$	$\hat{r} = f\sin\phi$	$\hat{r} = f\sin(\frac{\phi}{2})$

Cuadro 3.1: Definición de $\hat{r}$ para cada modelo de cámara de ojo de pez

En el cuadro 3.1 se encuentra la relación entre la componente radial y el ángulo de inci-

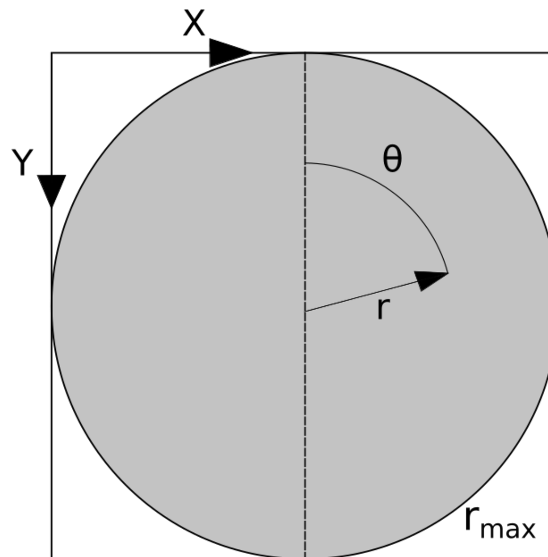


Figura 3.8: Coordenadas polares en imagen de ojo de pez

dencia dependiendo del modelo de proyección. De estas ecuaciones se obtiene la imagen final en coordenadas polares, de la cual se partirá en el apartado de composición de imágenes para construir esta imagen en el simulador.

En la sección del simulador donde se explica la composición de las imágenes, se verán las ecuaciones para pasar desde las coordenadas de la imagen hasta la obtención del punto 3D del entorno utilizando el modelo aquí descrito.

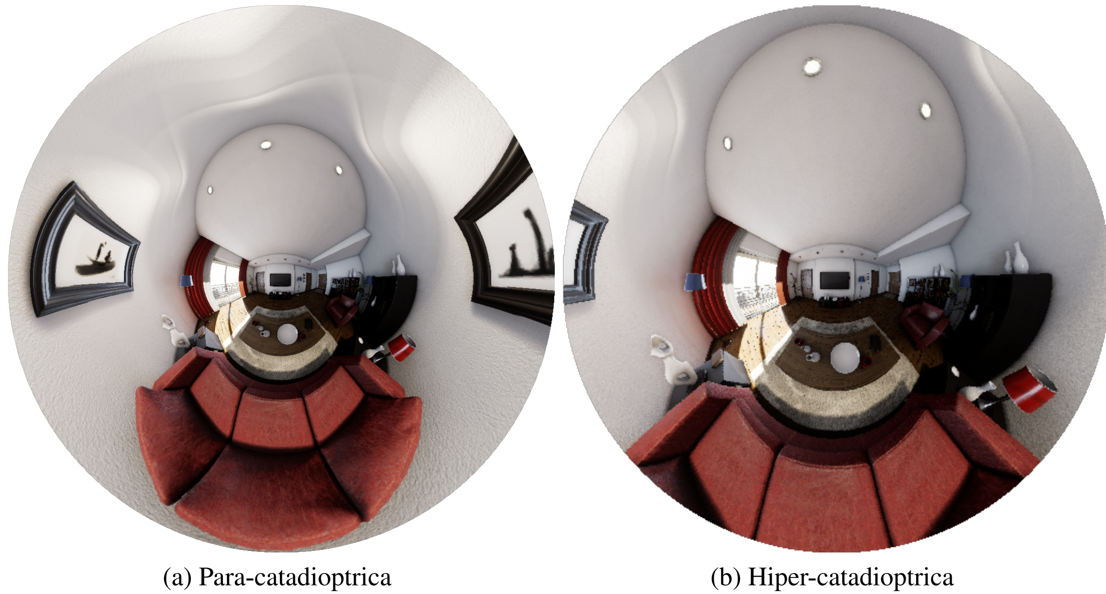
3.1.4 Cámaras catadióptricas

El modelo de la esfera es un modelo matemático que define cómo se proyecta un punto, o imagen, en cámaras catadióptricas. En este apartado se va a describir el modelo y las ecuaciones para la obtención de dos de los modelos implementados en el simulador: cámaras para-catadióptricas y cámaras hiper-catadióptricas.

Como primera definición, se define la cámara catadióptrica como una cámara perspectiva enfocada hacia un espejo. Dependiendo el tipo de espejo que se tiene, la parametrización en el modelo de la esfera cambia, siempre que estos espejos definan una cámara central, es decir con un único centro óptico. Los espejos parabólicos e hiperbólicos entran dentro de esta definición para cámaras catadióptricas centrales. Para otros tipos de espejos, como los esféricos y cónicos, será necesario definir otro modelo porque este tipo de espejos no definen cámaras centrales, puesto que se puede demostrar que no tienen un único centro óptico [21].

La proyección según el modelo de la esfera, detallado en [22] y [23], se puede definir en dos pasos:

- Intersección del punto en el espacio con el centro de la esfera. La línea que une el punto en el espacio que se quiere proyectar con el centro de una esfera, de radio unidad, donde se sitúa el origen de coordenadas del sistema de referencia, intersecta con dicha esfera en dos puntos,$r_{\pm}$, de los cuales solo uno tendrá significado físico real.



(a) Para-catadioptrica (b) Hiper-catadioptrica

Figura 3.9: Ejemplos imágenes catadioptricas

- Proyección al plano normalizado y posteriormente al plano de proyección. Los puntos de intersección de la esfera son proyectados a un plano normalizado con una función no lineal definida pasando por un punto C_p definido como $C_p = (0,0,-\xi)^T$. Esta función no lineal depende del parámetro ξ, el cual se define a partir del tipo de cámara. Una vez se tienen los puntos proyectados en este plano normalizado, se vuelven a proyectar en un plano de proyección el cual es el plano de imagen. Esta proyección viene dada por la matriz H_c donde está definida la calibración de la cámara, entre otras matrices.

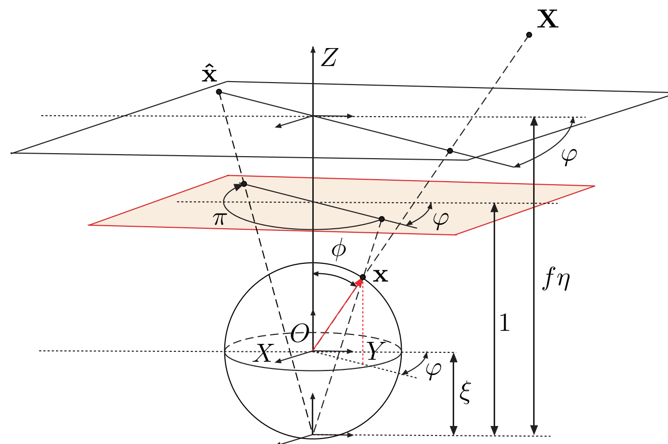


Figura 3.10: Esquema modelo de la esfera, con el plano normalizado y el plano de proyección sobre estos

Como se ha explicado, existen varios parámetros, matrices y funciones que caracterizan este modelo. Primero se va a caracterizar cada cámara con sus parámetros identificatorios.

Perspectiva	$\xi = 0$	$\Psi = -$
Hiper-catadioptrica	$0 < \xi < 1$	$\Psi = \frac{d+2p}{\sqrt{d+4p^2}}$
Para-catadioptrica	$\xi = 1$	$\Psi = 1 + 2p$

Cuadro 3.2: Parámetros del modelo de la esfera dependiendo el tipo del tipo de cámara

En el cuadro 3.2 se definen los parámetros de la esfera para la cámara perspectiva y los espejos centrales, parabólico e hiperbólico. El parámetro $2p$ define el semi *latus rectum* del espejo, mientras que d es la distancia entre el espejo y la cámara perspectiva. También se define el parámetro Ψ que se utilizará en las matrices de proyección al plano proyectivo.

Siguiendo el modelo, se define la función no lineal que proyecta las intersecciones del rayo en 3D y la esfera en el plano normalizado.

$$x = h(\chi) = \begin{pmatrix} X \\ Y \\ Z \pm \xi\sqrt{X^2+Y^2+Z^2} \end{pmatrix} \tag{3.6}$$

Con la ecuación 3.6, se puede pasar de las coordenadas del rayo en $3D$ a los puntos en el plano normalizado conociendo el parámetro ξ que define el tipo de cámara.

El siguiente paso son las matrices, donde se incluye la matriz de calibración de la cámara. Además de esta, se tiene una matriz de rotación de la cámara y una matriz que refleja el comportamiento del espejo a partir de sus parámetros.

$$H_c = K_c R_c M_c \tag{3.7}$$

$$K_c = \begin{pmatrix} f_x & s & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.8}$$

$$M_c = \begin{pmatrix} \Psi - \xi & 0 & 0 \\ 0 & \xi - \Psi & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -\eta & 0 & 0 \\ 0 & \eta & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.9}$$

A partir de estas matrices, se puede pasar de los puntos del plano normalizado al plano de proyección de la imagen.

Pasando por la función $h(\chi)$ y posteriormente por la matriz H_c, se puede pasar de un punto en el espacio, a través de la esfera hasta el plano de proyección de la imagen. En el apartado del simulador se especificarán las ecuaciones utilizadas, puesto que el camino seguido es el contrario, se pasa del plano proyectivo hasta el punto $3D$, utilizando las ecuaciones inversas de las expuestas.

3.1.5 Modelo de Scaramuzza

El modelo de Scaramuzza es un modelo matemático para definir de forma general sistemas omnidireccionales de imágenes con simetría de revolución. Como se comenta en [24], en la mayoría de las cámaras omnidireccionales se pueden definir dos sistemas de referencia, uno en

el plano del sensor, (u'', v''), y otro en en el plano de la imagen de la cámara, (u', v'). En la imagen 3.11, obtenida en [24], se puede ver cómo están definidos estos sistemas de coordenadas.

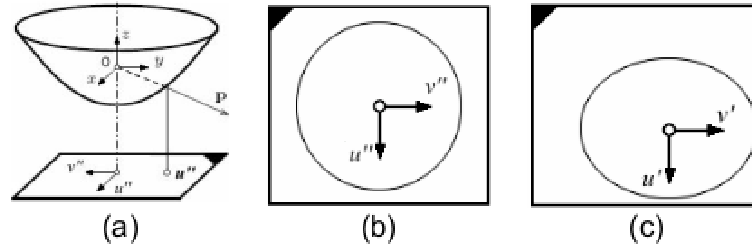


Figura 3.11: Modelo de Scaramuzza: (a) Sistemas de referencia en el modelo de Scaramuzza; (b) Plano del sensor; (c) Plano de imagen de la cámara

Suponiendo que $\boldsymbol{X}$ es un punto en la escena, se define $u'' = \left[u'', v''\right]^T$ como la proyección de dicho punto en el plano del sensor y $u' = \left[u', v'\right]^T$ la proyección en el plano de la cámara. Tal como se explica en [24] y en [25], se puede definir una transformación afín para pasar de u'' a u'. Además, se define una función no lineal $\mathbf{g}$, que permite relacionar la proyección en el sensor con el vector desde el origen de coordenadas hasta punto proyectado de la escena. De esta forma se puede definir el modelo de Scaramuzza como:

$$\lambda p = \lambda g(u'') = \lambda g((Au' + t)) = P\boldsymbol{X} \tag{3.10}$$

donde $\boldsymbol{X}$ está expresada en coordenadas homogéneas y P es la matriz de proyección. Se suponen conocidos los parámetros de calibración de la cámara que se quiere modelar y que cumplen con lo especificado en la ecuación (3.10), se define la función $\mathbf{g}$ como:

$$\mathbf{g}(u'', v'') = (u'', v'', f(u'', v''))^T$$

donde la función f se define como una función con simetría de revolución respecto del eje del sensor.

Como generalización, f se parametriza como una función polinómica de grado N, haciendo que sea válida para diversos tipos de cámara omnidireccional. Por otro lado, el parámetro ρ se define como la distancia métrica de cada punto de la imagen con el centro de coordenadas, el cual coincide con el centro del sensor. La función f queda definida como:

$$f(u'', v'') = a_0 + a_1\rho + a_2\rho^2 + ... + a_N\rho^N \tag{3.11}$$

donde los parámetros $[a_0, a_1, ..., a_N]$ vienen dados por la calibración de la cámara que se quiere modelar y ρ está definido como $\rho = \sqrt{u''^2 + v''^2}$.

Esta definición es muy útil para modelizar cámaras con simetría de revolución respecto de su eje principal, evitando desalineamientos o imperfecciones en lentes y espejos. Sin embargo, debido a la alta precisión en la fabricación de este tipo de cámaras omnidireccionales, el modelo es una muy buena aproximación matemática a este.

Figura 3.12: Ejemplo de imagen creada usando el modelo de Scaramuzza

3.2 Cámaras no centrales

Como se ha visto, las cámaras centrales se caracterizan porque todos los rayos de luz se cruzan en un único punto, ya sea en la cámara o en el espejo. Por el contrario, las no centrales se caracterizan porque no tienen un único punto de intersección de los rayos de luz.

Para la definición de los modelos de cámaras no centrales se introducen las coordenadas de Plücker, las cuales parametrizar una recta en el espacio 3D definiendo un vector director y su perpendicular en dirección al origen de coordenadas. Esto permite parametrizar de fácilmente cualquier rayo 3D y trabajar con él. En uno de los anexos del [22] se definen este tipo de coordenadas y cómo se trabaja con ellas.

$$\Xi = \begin{pmatrix} \xi \\ \bar{\xi} \end{pmatrix} = \begin{pmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \\ \bar{\xi}_1 \\ \bar{\xi}_2 \\ \bar{\xi}_3 \end{pmatrix} \tag{3.12}$$

A continuación, se definirán los modelos de proyección de las cámaras no centrales implementadas. Se empieza por el panorama no central y se continuará con cámaras catadióptricas.

3.2.1 Panorama

Los panoramas no centrales son muy similares a los panoramas equirectangulares propuestos en la sección 3.1.1. Son imágenes que cubren los 360º alrededor de un punto y los 180º arriba y abajo. La principal diferencia es que las imágenes no están tomadas desde este punto, sino desde una distancia fija alrededor de este. Este puede parecer un concepto extraño, pero son

un tipo de imágenes que todos hemos realizado alguna vez, puesto es la base de las imágenes panorámicas que se pueden crear con el teléfono móvil. En este gesto, la persona rota sobre sí misma con la cámara en alto, enfocando una escena y girando el móvil consigo, lo que hace que el foco de la cámara también se mueva, creando una imagen sin un foco puntual, sino con un recorrido de focos de cámara. La principal diferencia que se va a apreciar es el radio de este recorrido de focos. Desde un teléfono móvil, este radio suele ser muy pequeño (10-20 cm) en un entorno muy grande, como paisajes. El modelo propuesto e implementado permite un radio mayor, creando imágenes donde no se ve la totalidad de la escena, puesto puede haber zonas que se salten o queden detrás del trayecto de centros ópticos.

De forma similar a la proyección equirectangular en la esfera, el panorama no central es el desarrollo de un semi-toroide, como el de la figura 3.13, donde se proyecta el entorno. La principal diferencia es que no se tiene un centro óptico único, sino que está distribuido en una trayectoria circular dentro de este semi-toroide.

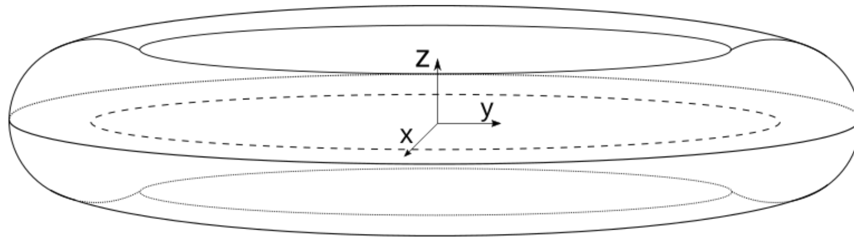


Figura 3.13: Panorama no central: toroide de revolución

El modelo de proyección consiste en una simetría de revolución donde los “rayos” de luz cortan con el eje de revolución y una trayectoria circular simultáneamente, como se puede ver en la figura 3.14.

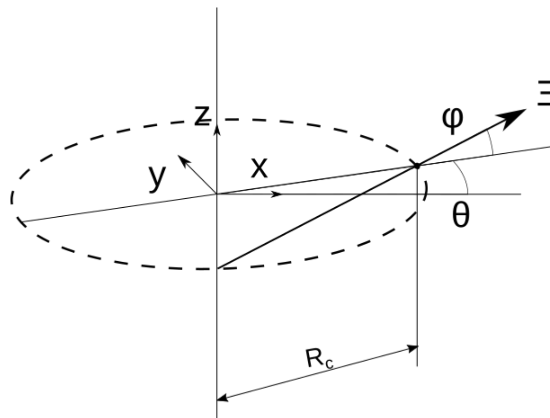


Figura 3.14: Esquema panorama no central

Utilizando las coordenadas de Plücker, se pueden definir los rayos de luz que salen hacia el entorno como en la ecuación (3.13), obtenida de [26]. Se puede ver que estos rayos dependen de dos coordenadas angulares, φ y θ, y de un parámetro geométrico, R_c.

$$\Xi = \begin{pmatrix} \xi \\ \bar{\xi} \end{pmatrix} = \begin{pmatrix} \sin\varphi\cos\theta \\ \sin\varphi\sin\theta \\ \cos\varphi \\ R_c\sin\varphi\sin\theta \\ -R_c\sin\varphi\cos\theta \\ 0 \end{pmatrix} \tag{3.13}$$

donde R_c es la distancia entre el eje de revolución y la trayectoria de centros ópticos definida como un círculo y φ y θ son ángulos en coordenadas esféricas del rayo de luz desde su centro óptico correspondiente.

Definido el rayo, interesa saber cuál es la relación entre sus parámetros y los puntos del entorno. Para ello, suponiendo conocido el parámetro geométrico R_c, se definen lsa coordenadas angulares según las ecuaciones (3.14) y (3.15).

$$\theta = \arctan\left(\frac{x_2}{x_1}\right) \tag{3.14}$$

$$\varphi = \arctan\left(\frac{x_3}{\sqrt{{x_1}^2 + {x_2}^2} - x_4 R_c}\right) \tag{3.15}$$

donde el punto en el entorno está definido en coordenadas homogeneas como:

$$P = (x_1, x_2, x_3, x_4)^T$$

En la figura 3.15 se puede ver un ejemplo de este tipo de imagen. Aunque similar al panorama equirectangular, la distorsión es distinta y el campo de vista cambia, puesto las zonas interiores del semi-toroide no se pueden llegar a apreciar.

Figura 3.15: Panorama no central con radio 100cm

3.2.2 Cámaras catadióptricas

El modelo para cámaras catadióptricas no centrales difiere respecto al de cámaras centrales. Mientras que el principio es el mismo, enfocar un espejo con una cámara perspectiva para obtener la imagen reflejada del espejo, el modelo matemático de los espejos no centrales difiere enormemente respecto de los espejos centrales. En este proyecto se han implementado dos tipos de espejos no centrales: espejos cónicos y espejos esféricos. En [22] se hace una extensa explicación de cómo se obtienen los modelos matemáticos de este tipo de espejos. En este proyecto se va a hacer un resumen de dónde vienen las ecuaciones y qué parámetros se van a utilizar para su implementación en el simulador.

Siguiendo el esquema de la figura 3.16, se puede ver cómo está definido el rayo que llega al espejo no central con las coordenadas de Plücker.

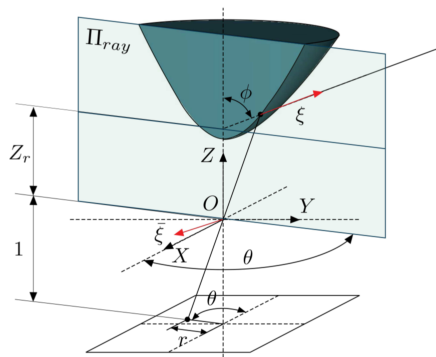


Figura 3.16: Esquema espejo no central

A partir de dicho esquema, se puede hacer una definición de las coordenadas de Plücker utilizando los ángulos de azimut y elevación, θ y ϕ, y parámetros de calibración, como Z_r, para obtener una definición genérica de los rayos que se reflejan en el espejo.

$$\Xi = \begin{pmatrix} \xi \\ \bar{\xi} \end{pmatrix} = \begin{pmatrix} \sin\phi\cos\theta \\ \sin\phi\sin\theta \\ \cos\phi \\ -Z_r\sin\phi\sin\theta \\ Z_r\sin\phi\cos\theta \\ 0 \end{pmatrix} \tag{3.16}$$

A partir de esta definición genérica de parámetros, se puede particularizar a los distintos tipos de sistemas catadióptricos.

En este proyecto no se va a entrar en detalle a el desarrollo matemático, sino que se va a presentar el resultado utilizado. Este desarrollo y una definición más extensa de las coordenadas de Plücker se puede encontrar en [22], [21] y [27].

En los **espejos cónicos**, se define una cámara perspectiva colocada en el eje de revolución del cono, para garantizar la simetría de revolución. Esta disposición permite definir varias distancias entre cámara y espejo que entrarán en el modelo como parámetros para la definición del rayo de luz que entra en la cámara.

Suponiendo que la cámara perspectiva se encuentra en el centro de coordenadas de la figura 3.17, se puede definir la distancia Z_m como la distancia entre cámara y espejo. Conocida esta

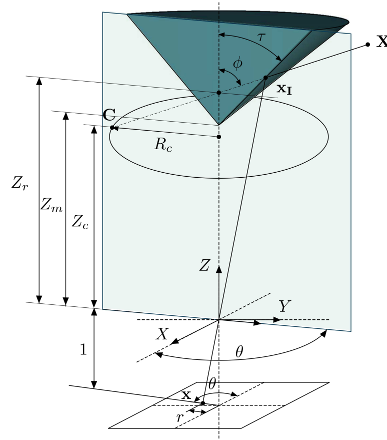


Figura 3.17: Esquema espejo cónico

distancia y el ángulo de apertura del cono, τ, se pueden definir los parámetros R_c y Z_c, los cuales determinan la circunferencia donde se encuentran los focos de la imagen catadióptrica. Como ya se explicó en la sección 3.2.1, en cámaras no centrales no se tiene un único centro óptico, sino que están distribuidos a lo largo de una curva o superficie, o como en este caso, una circunferencia.

$$R_c = Z_m \sin 2\tau \tag{3.17}$$

$$Z_c = Z_m(1 - \cos 2\tau) \tag{3.18}$$

Conocido dónde se encuentra esta circunferencia, haciendo la intersección de esta esfera con el plano que contiene el eje de revolución del cono y el punto 3D que se quiere proyectar, X, se obtiene el punto C en la circunferencia con el que definir el rayo de luz que entra en la cámara. De este modo se define el modelo de proyección, conocido el punto 3D, como:

$$\bar{x} = \begin{pmatrix} \left(\sin 2\tau \frac{X_3 - Z_m X_4}{\sqrt{{X_1}^2 + {X_2}^2}} - \cos 2\tau\right) X_1 \\ \left(\sin 2\tau \frac{X_3 - Z_m X_4}{\sqrt{{X_1}^2 + {X_2}^2}} - \cos 2\tau\right) X_2 \\ Z_m X_4 + (X_3 - Z_m X_4)\cos 2\tau + \sqrt{{X_1}^2 + {X_2}^2}\sin 2\tau \end{pmatrix} \tag{3.19}$$

donde el punto 3D está definido como $X = (X_1, X_2, X_3, X_4)^T$ en coordenadas homogéneas.

De esta forma queda definido cómo se proyecta un punto 3D en el plano de proyección de una cámara perspectiva. En el apartado del simulador se expondrá el proceso contrario, es decir, pasar del plano de proyección de la cámara hasta el punto 3D en el entorno.

Por otro lado, partiendo del mismo modelo, se tienen los **espejos esféricos**. En este caso no importa la localización de la cámara, puesto un espejo esférico tiene infinitos ejes de revolución. Siempre que la cámara esté colocada perpendicularmente al espejo, este modelo será válido. Para este tipo de espejo se definen otros dos parámetros, R_s y Z_s, los cuales definen el radio

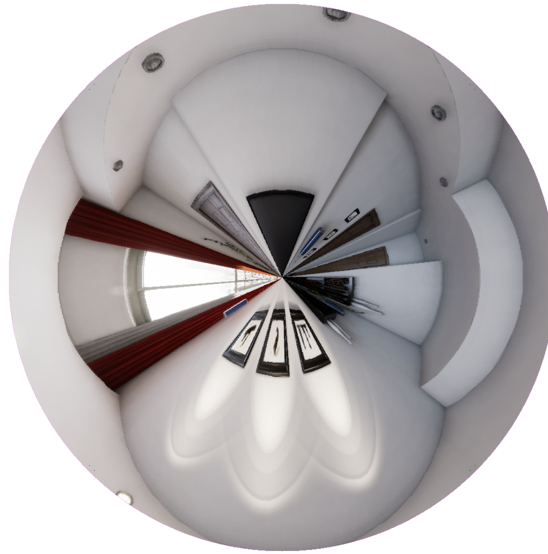


Figura 3.18: Ejemplo imágen catadióptrica sobre un espejo cónico de $\tau = 55^{o}$

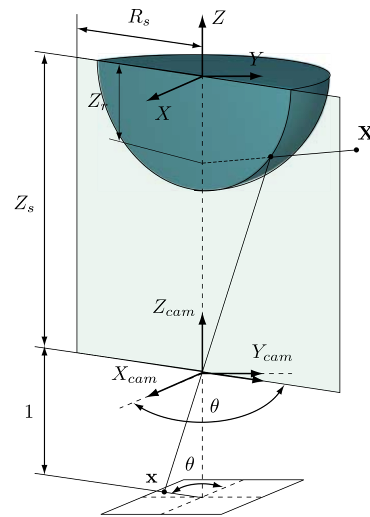


Figura 3.19: Esquema espejo esférico

del espejo y la distancia del centro de este a la cámara. Para poder reutilizar los parámetros anteriores, se puede definir la distancia como:

$$Z_s = Z_m + R_s$$

donde Z_m es la distancia entre el vértice del espejo, más fácil de medir, y la cámara.

En [22] define que, si la hay, no existe una solución cerrada única para definir la proyección de un punto 3D en el plano de proyección de la cámara perspectiva. Sin embargo, sí se tiene una solución para el proceso inverso, definir un punto 3D del entorno a partir de su proyección en el plano de proyección. Puesto que es este segundo procedimiento el que es interesante para este proyecto, es el que se va a definir a continuación, y el que se resumirá en el apartado de simulación.

Como en la exposición del modelo, se va a definir los resultados obtenidos y no el proceso de obtención de estos resultados. Dicho esto, se define el rayo que entra en la cámara en sus coordenadas de Plücker como (3.20)

$$\Xi = \begin{pmatrix} x\delta \\ y\delta \\ -\zeta \\ -(\delta+\varepsilon)yZ_s \\ (\delta+\varepsilon)xZ_s \\ 0 \end{pmatrix} \tag{3.20}$$

donde

$$\delta = 2r^2Z_{rel}^4 - 2z\sqrt{\gamma}Z_{rel}^2 - 3\rho^2Z_{rel}^2 + \rho^2 \tag{3.21}$$

$$\varepsilon = (-r^2+z^2)Z_{rel}^2 + 2\sqrt{\gamma}z + \rho^2 \tag{3.22}$$

$$\zeta = 2r^2zZ_{rel}^4 - z\rho^2Z_{rel}^2 - 2\sqrt{\gamma}(-r^2Z_{rel}^2+\rho^2) - z\rho^2 \tag{3.23}$$

$$\gamma = (-r^2Z_{rel}^2+\rho^2)Z_{rel}^2 \tag{3.24}$$

$$r = \sqrt{x^2+y^2} \tag{3.25}$$

$$\rho = \sqrt{x^2+y^2+z^2} \tag{3.26}$$

$$Z_{rel} = \frac{Z_s}{R_s} \tag{3.27}$$

En la definición de estas ecuaciones se parte de las coordenadas del plano de proyección de la cámara para conseguir el rayo que refleja en el espejo en coordenadas de Plücker. En el apartado del simulador se definirá como se ha implementado para poder obtener las imágenes catadióptricas de espejos esféricos.

Figura 3.20: Ejemplo de imagen catadióptrica con espejo esférico de radio 50cm

Capítulo 4

Simulador de cámaras centrales

En esta parte del proyecto se va a describir cómo está compuesto el simulador, cómo se interactúa con el entorno virtual con UnrealCV y cómo se han implementado los distintos modelos de cámara mencionados. La obtención de una imagen en este simulador tiene dos pasos:

- **Adquisición de imágenes**: este primer paso consiste en crear un *cubemap* en la localización del entorno donde se quiere obtener la imagen final, independiente de la orientación de la cámara. En este primer paso es donde se interactúa con el entorno virtual de UnrealCV y el paso más costoso, puesto que se tiene que conocer el entorno para poder introducir las coordenadas de la cámara.

- **Composición de imágenes**: este paso es independiente del entorno virtual, pudiendo incluso cerrarlo para aligerar la carga de trabajo del ordenador. En este paso se aplican los modelos de cámara vistos para componer una imagen final a partir de un *cubemap* determinado. Seleccionada una posición y orientación de la cámara, se obtiene la imagen final.

Para cámaras centrales, los dos pasos para la creación de las imágenes se hace de forma independiente, es decir, se procede a crear primero los *cubemaps*, pudiendo verificar que las posiciones seleccionadas son las correctas y posteriormente iniciar la composición de las imágenes omnidireccionales.

Para las cámaras no centrales se podría hacer este mismo proceso. Sin embargo, debido a que no comparten un único centro óptico y la gran cantidad de imágenes necesarias para la composición de las imágenes, estos pasos se hacen de forma consecutiva sin conservar las imágenes de la adquisición reduciendo la memoria necesaria.

4.1 Adquisición de imágenes

Como ya se ha comentado, la adquisición de imágenes es el primer paso a realizar para construir las imágenes omnidireccionales. En este primer paso se interactúa indirectamente con el entorno de UnrealCV a través de *scripts* escritos en Python. Desde estos *scripts* se puede acceder a determinados comandos para definir lo necesario en la construcción de los *cubemaps* y de los distintos tipos de imagen que se pueden extraer del entorno:

- Orientación y posición de la cámara
- Modo de captura: Iluminación, semántica o profundidad
- Campo de vista de la cámara

En el anexo A están definidos todos los comandos programables en Python definidos en [10], resaltando aquellos que se han utilizado para este proyecto.

En el desarrollo de videojuegos, se denomina *cubemap* a imágenes que se utilizan de fondo en los escenarios, de forma que se tenga una imagen cuando un jugador mira al horizonte. Normalmente se utilizan 5 o 6 imágenes colocadas como un cubo alrededor del jugador, moviéndose con este, de forma que lo acompaña durante todo el juego. En el caso de este simulador, el *cubemap* se genera a través de 6 imágenes perspectivas tomadas desde dentro de la escena, perdiendo en este caso la definición de estar en el horizonte, y dependiendo de la posición donde se coloque la cámara, perdiendo la condición de que el jugador nunca puede llegar a él. Llegados a este punto, ¿por qué se sigue denominando *cubemap*?. La respuesta viene en el paso de la composición de las imágenes. Una vez construido el *cubemap*, la cámara que se estará modelando lo verá en el infinito, proyectando todos los elementos que tenga a su alrededor desde el punto de vista de la cámara. De esta manera se consigue tener libertad de 360 grados en el giro de la cámara.

Sin embargo, 360º de libertad entorno a una posición sería la definición de una esfera cuyo centro sea esa posición en lugar de un cubo. Este objetivo se podría conseguir, realizando capturas para cada grado, obteniendo una resolución mucho mayor para cada giro posible dentro de esta esfera, pero este planteamiento conllevaría una gran necesidad de memoria y tiempo de computación para la adquisición de las imágenes. Simplificando la esfera hasta la figura del cubo, se obtiene una aproximación lo suficiente buena como para no ver discrepancias en el resultado final. Por otro lado, construir el *cubemap*, incluso antes de aplicar un modelo de cámara, facilita la composición de una imagen de cualquier campo de visión y en cualquier dirección principal de cámara, obteniéndose rápidamente adquisiciones en distintas posiciones dentro de la escena, con una baja utilización de almacenamiento.

Siguiendo en el paso de la adquisición de las imágenes, el entorno de UnrealCV proporciona distintos tipos de imágenes, los cuales proporcionan información muy interesante, como la creación de bases de imágenes fotorrealistas con un etiquetado automático. Como ya se ha comentado, existen tres modos de captura o tipo de imagen: iluminación, son las imágenes que se han estado mostrando hasta ahora; semántica, son imágenes que distinguen por colores cada uno de los objetos de la escena; y profundidad, devuelve la distancia de cada píxel a la cámara.

Como se ha ido mostrando, las imágenes del tipo **iluminación**, *lit*, muestran la escena con gráficos foto-realistas, teniendo en cuenta los focos de luz y reflejos de estos en los distintos materiales. Estos efectos se consiguen desde el propio motor gráfico, Unreal Engine 4, dependiendo

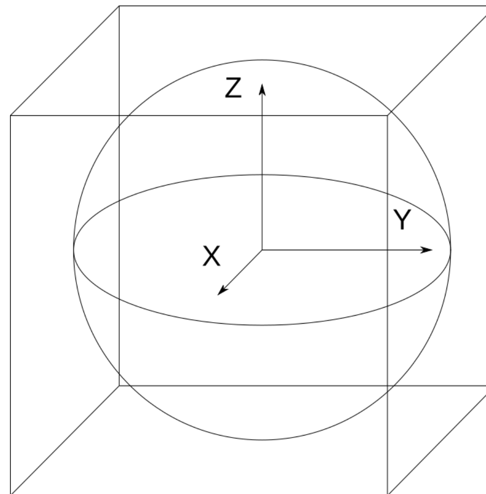


Figura 4.1: Simplificación de la esfera al *Cubemap*

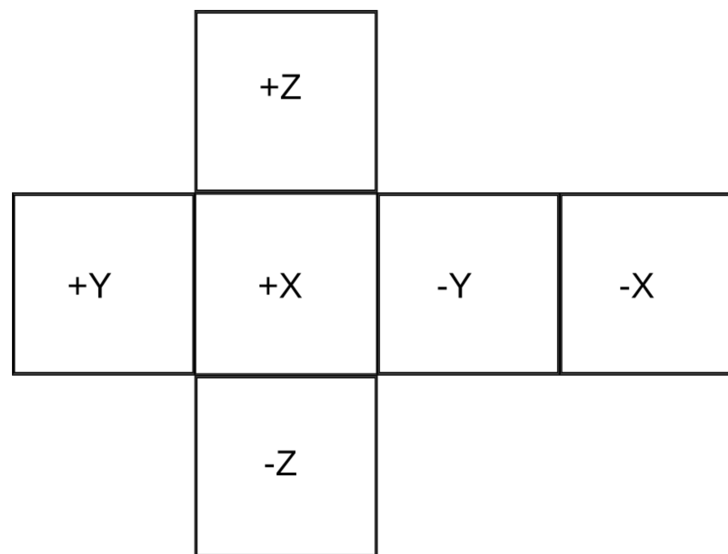


Figura 4.2: Desarrollo del *cubemap* definiendo la dirección de cada lado

del diseñador del escenario. Puesto que el objetivo del simulador es la obtención de imágenes que puedan ser utilizadas en visión por computador, los escenarios deben ser diseñados en vistas a la obtención de imágenes foto-realistas.

Otro tipo de captura es la **semántica**, *object mask*, donde cada objeto de la escena está definido con un único color. Además, este modo de captura nos da un *ground truth*, o realidad, con la que se pueden asegurar las dimensiones y geometría de los objetos. Este modo de captura semántico permite un etiquetado automático de las imágenes que se obtienen, puesto se puede obtener la información de que objeto corresponde con qué color. La gran ventaja que se obtiene de este tipo de imagen es el etiquetado automático, es decir, tener una certeza del 100 %, con precisión de píxel, de dónde se encuentra un objeto dentro de una imagen. Este tipo de captura permite obtener, al final, una base de datos totalmente etiquetada, sabiendo que no se van a tener errores humanos y sin tener a una persona etiquetando imagen por imagen, lo cual es un trabajo

Figura 4.3: Cubemap en modo de captura *lit*

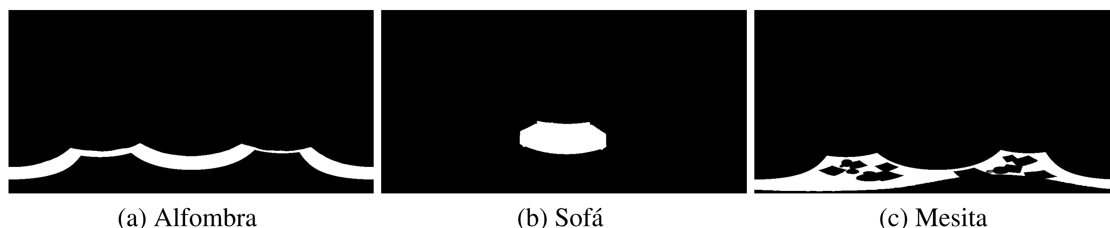
tedioso y propenso a errores. Esta cualidad hace al simulador una herramienta muy potente para el entrenamiento de redes neuronales, puesto que no depende de etiquetados humanos que conllevan horas de trabajo. Permite etiquetar cualquier escena que se haya diseñado, independiente de lo compleja o número de objetos que tenga. Al trabajar en colores RGB, se pueden obtener hasta 16,5 millones de combinaciones diferentes para definir los objetos de una imagen.

Figura 4.4: Cubemap en modo de captura *object mask*

A raíz de este tipo de captura, se desarrollaron dos funciones para aprovechar el *ground truth* y discriminación semántica de las imágenes. Por un lado, se ha desarrollado una función que genera máscaras binarias de cada objeto de la escena, permitiendo ayudar a entrenar redes

de reconocimiento de objetos. Como se puede ver en la figura 4.5, estas máscaras binarias representan únicamente la parte visible de cada objeto en la escena, pudiéndose usar como etiquetado en el reconocimiento de objetos.

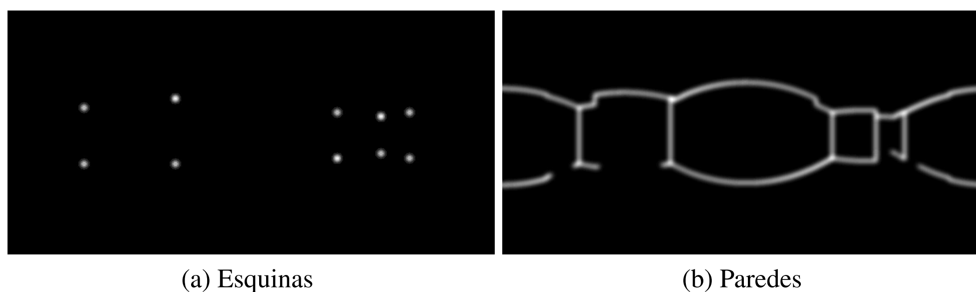
Aunque no se ha podido evaluar su resultado, se ha implementado este tipo de máscara buscando ser compatible con el trabajo [28]. En el anexo D se presentan varios escenarios con esta implementación de extracción de objetos para distintos modelos de cámaras omnidireccionales.



(a) Alfombra (b) Sofá (c) Mesita

Figura 4.5: Máscaras binarias en imagen equirectangular

Otra de las funciones desarrolladas ha sido la extracción de esquinas e intersección de paredes para reconocimiento de *layouts* de habitaciones. Al igual que la obtención de máscaras binarias, se implementa en la imagen compuesta, pero se obtiene a partir de la captura semántica de la escena. En la figura 4.6 se puede ver un ejemplo de la implementación de la extracción de paredes y esquinas en el escenario principal que se ha usado en el proyecto.



(a) Esquinas (b) Paredes

Figura 4.6: Esquinas y paredes

El etiquetado, la creación de las máscaras binarias y la extracción de esquinas e intersección de paredes es un proceso totalmente automático. Poder obtener estos elementos necesarios en algoritmos de visión de forma automática, sin la intervención de una persona, permite agilizar el proceso de obtención de una base de imágenes amplia y totalmente funcional desde el primer minuto, sin tener que hacer un trabajo extra para acondicionar los datos a las necesidades del aprendizaje.

Por último, se tiene la captura de **profundidad**, *depth*, la cual se trata de forma diferente, puesto se quiere conocer el valor exacto de profundidad de cada punto a la imagen. Puesto UnrealCV tiene una estrecha relación con Python, utiliza una de sus librerías matemáticas más utilizadas, ***Numpy***, para exportar la información de profundidad en formato *.npy*. Este formato guarda la información de forma matricial, dando el valor de profundidad de cada píxel en su

posición matricial correspondiente. De esta forma se puede conocer con exactitud la distancia de cada píxel de la imagen con la cámara. Este tipo de información es muy útil y necesaria para percepción y navegación en robots y vehículos autónomos, puesto ayuda a controlar las distancias de seguridad que deben tomar para evitar colisiones y/o obstáculos en un camino determinado.

En la implementación de este simulador, se han adoptado dos soluciones, una más gráfica y otra puramente numérica. En la solución numérica, se almacena la información de profundidad en archivos *.npy* como se ha descrito anteriormente. De esta forma se puede reconstruir el escenario desde el punto de vista de la cámara sin perder mucha información. En la aproximación gráfica, véase figura 4.7, se define un límite máximo de distancia que se puede percibir para representar en escala de grises una imagen con las distancias a la cámara, representando en oscuro lo más cercano y en claro lo más alejado. Se selecciona un máximo de distancia para poder tener una resolución suficiente en los objetos cercanos, los cuales suelen ser los más importantes cuando el objetivo es la conducción o navegación autónoma. Determinar la distancia máxima que se puede representar es necesario en casos donde tenemos distancias infinitas desde el objetivo de la cámara, por ejemplo, imágenes con una visión parcial del cielo o el horizonte, donde no hay ningún obstáculo que dé un valor determinado de profundidad.

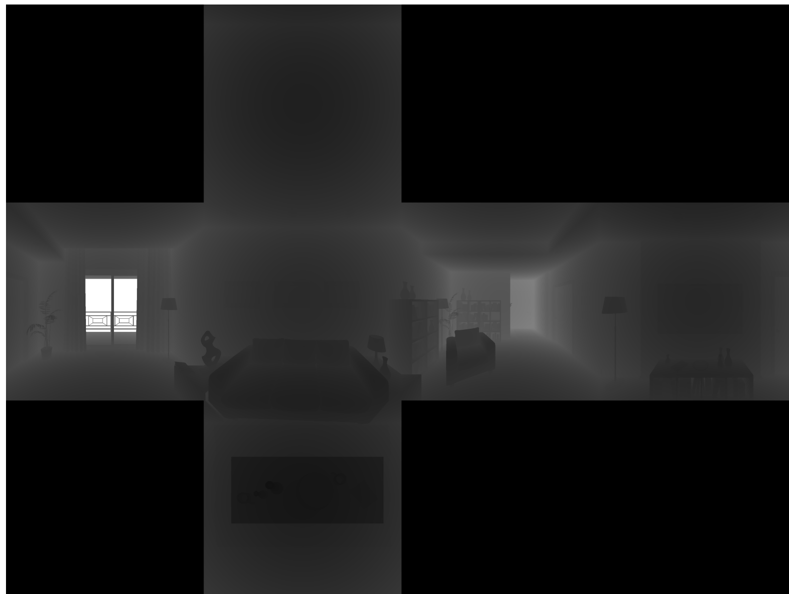


Figura 4.7: Cubemap en modo de captura de profundidad, representación gráfica

En este apartado se ha comentado como se crean los *cubemaps* y qué tipos de imágenes se han creado en este simulador. También se ha comentado un uso práctico, que se desarrollará con más atención posteriormente, para el uso de estas imágenes en redes neuronales como CFL [13], pudiendo crear imágenes target con las que entrenar dicha red. A continuación, se va a definir cómo se crean las imágenes a partir de los *cubemaps*.

4.1.1 Límites de imagen

En la creación de los *cubemaps*, y posterior composición de imágenes, se descubrió que aparecían unas 'costuras' alrededor de las imágenes. Esto es debido a que UnrealCV diseña una cámara perspectiva que tiene una pérdida de luminosidad en el borde de la imagen, simulando el efecto de viñeteo de objetivos ópticos de cámaras reales. Por ello, aunque la lógica dictaba que utilizando imágenes con un campo de vista de $90^o x 90^o$ (horizontal x vertical) se cubría la totalidad del entorno alrededor de la cámara, con este tipo de adquisición aparecían estas 'costuras' que estropeaba una composición limpia de imágenes, como se puede ver en la imagen 4.8.

Figura 4.8: Imagen equirectangular con 'costuras' entre las imagenes adquiridas

Para evitar estas 'costuras' que aparecen en las imágenes, la solución aplicada fue tomar imágenes con un campo de vista mayor, recortando posteriormente lo que estuviera fuera del campo de vista propio de una imagen de $90^o x 90^o$. Puesto que interesa mantener una resolución adecuada en la adquisición de las imágenes, se ha aumentado la resolución de la adquisición para obtener, tras el recorte, imágenes de un mega-píxel (1024x1024 píxeles), tal como se ve en la figura 4.9.

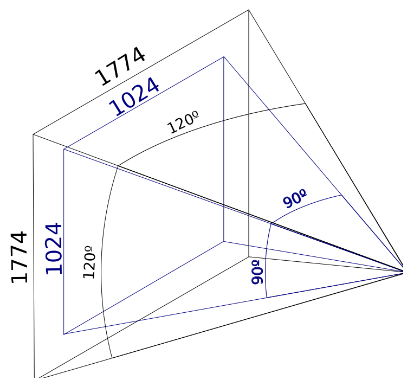


Figura 4.9: Esquema del recorte en la adquisición de las imágenes

4.2 Composición de imágenes

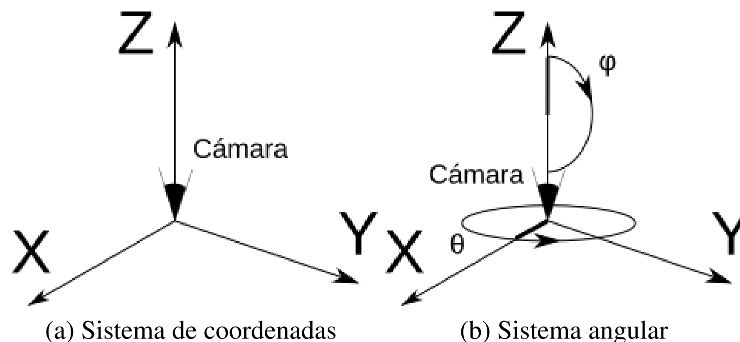
La adquisición y composición de cámaras centrales y no centrales difiere sensiblemente. Mientras que en los sistemas centrales la composición se obtiene a partir de un *cubemap* adquirido en la escena, en las cámaras no centrales la adquisición y composición se hace de forma simultánea.

Las imágenes centrales se componen a partir de un *cubemap* adquirido en la escena. La composición de cada tipo de imagen depende del modelo matemático de ésta, aunque siguen un patrón común para facilitar y unificar el proceso de composición de las imágenes.

$$\begin{pmatrix} u \\ v \end{pmatrix} \stackrel{f(p)}{\rightarrow} \begin{pmatrix} \theta \\ \phi \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{4.1}$$

La ecuación (4.1) muestra el camino a seguir en la composición de las imágenes. Inicialmente se parte de las coordenadas píxel de la imagen destino, $(u,v)^T$, que se quiere crear, desde donde se calculan las coordenadas esféricas que indican la posición de ese píxel en la imagen. Este paso se hace a través del modelo de la cámara que se está simulando, siendo $f(p)$ la función que incluye dicho modelo, así como otros pasos. Una vez se conocen las coordenadas esféricas, $(\theta,\phi)^T$, del píxel cuya información se está buscando, se construye el vector dirección, $(x,y,z)^T$, del rayo de proyección que pasa por el centro óptico de la cámara y por el píxel de la imagen destino. Finalmente, se calcula la intersección de este rayo con el *cubemap* obteniendo de este la información que queremos proyectar (color, profundidad o etiquetado semántico).

A continuación, se va a repasar los distintos modelos matemáticos de las cámaras y se va a detallar cómo se han conseguido estas coordenadas esféricas. En todos los apartados se va a considerar el sistema de coordenadas dextrógiro y coordenadas angulares según la figura 4.10.



(a) Sistema de coordenadas (b) Sistema angular

Figura 4.10: Sistemas utilizados en los modelos

4.2.1 Equirectangular

El caso de las imágenes equirectangulares es el más sencillo. Como se ha explicado en el modelo, la imagen equirectangular hace un mapeado del entorno alrededor del punto desde donde se toma la imagen destino. Por lo tanto, la imagen destino se puede definir en coordenadas esféricas directamente, obteniendo una correlación directa entre el píxel de la imagen destino y las coordenadas esféricas.

$$\theta = \left(\frac{2x}{x_{max}} - 1\right)\pi \tag{4.2}$$

$$\phi = \left(1/2 - \frac{y}{y_{max}}\right)\pi \tag{4.3}$$

En las ecuaciones (4.2) y (4.3) se puede ver la correlación entre las coordenadas píxel de la imagen destino con las coordenadas esféricas que se utilizan para buscar la información en el entorno. En la figura 4.11 se puede ver gráficamente esta correlación.
Con la definición utilizada, estas coordenadas esféricas se mueven en el rango:

$$-\pi < \theta < \pi$$

$$-\frac{\pi}{2} < \phi < \frac{\pi}{2}$$

De esta forma, el centro de las coordenadas esféricas queda en el centro de la imagen destino, definiendo la dirección principal de la cámara con las coordenadas

$$(\theta, \phi) = (0, 0)$$

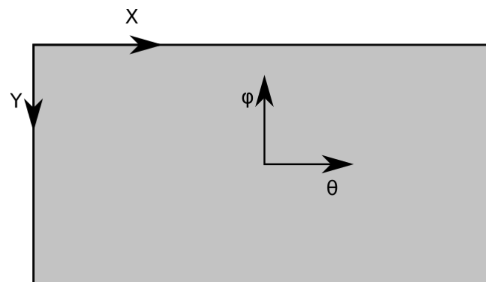


Figura 4.11: Coordenadas esfericas en imagen equirectangular

La implementación del modelo equirectangular se puede resumir con el siguiente código. Se puede observar que la salida de esta función es el vector 3D que sale de la cámara, el cual se rotará posteriormente para obtener las coordenadas esféricas en el sistema de coordenadas del *cubemap*.

```
#import numpy as np
def equirectangular(x, y, x_max, y_max):
    theta = (2*x/float(x_max) - 1.0)*math.pi
    phi = (1/2.0 - y/float(y_max))*math.pi
    x_og = math.cos(phi)*math.cos(theta)
    y_og = math.cos(phi)*math.sin(theta)
    z_og = math.sin(phi)
    vec = np.matrix('{};{};{}'.format(x_og,y_og,z_og))
    return vec
```

4.2.2 Cilíndrico

En el modelo de imagen cilíndrica, las coordenadas utilizadas son las coordenadas polares. Puesto que tenemos la imagen destino definida en la pared lateral de un cilindro, este tipo de coordenadas se ajusta perfectamente. Sin embargo, dada la parametrización del entorno en un *cubemap*, se debe hacer un paso a coordenadas esféricas.
En esta implementación, inicialmente se calculan las coordenadas polares respecto de las coordenadas del píxel. Esto hace que el mapeado de la imagen destino quede en coordenadas polares, como se puede apreciar en la figura 4.12.

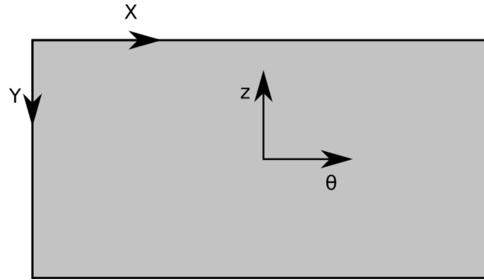


Figura 4.12: Coordenadas cilindricas en la imagen

Las ecuaciones (4.4) y (4.5) definen el paso de las coordenadas del píxel a las coordenadas polares. En esta ecuación entra un parámetro que no depende del sistema coordenado, FOV_h. Este parámetro es el campo de visión horizontal de la imagen cilíndrica. En el caso de este simulador es un parámetro que el usuario puede cambiar a su conveniencia. Estas ecuaciones definen el rango de las coordenadas polares como:

$$-\frac{FOV_h}{2} < \theta < \frac{FOV_h}{2}$$

$$-1 < z < 1$$

Posteriormente hay que hacer el cambio a coordenadas esféricas. En este caso, la coordenada angular θ sería la misma en en ambos sistemas de coordenadas. Para la obtención de la coordenada ϕ se ha supuesto que la esfera del sistema coordenado es de radio unidad, permitiendo definir esta coordenada como en la ecuación (4.6).

$$\theta = \left(\frac{2x}{x_{max}} - 1\right)\frac{FOV_h}{2} \tag{4.4}$$

$$z = 1 - \frac{2y}{y_{max}} \tag{4.5}$$

$$\phi = \arcsin z \tag{4.6}$$

Una vez definidas las coordenadas esféricas, en este modelo encontramos algunas restricciones. Puesto que el modelo proyecta el entorno en la pared lateral de un cilindro, se debe tener en cuenta otro parámetro externo definido por el usuario, el campo de vista vertical. En la implementación de esta imagen, se omite la búsqueda en el entorno de todos los píxeles cuya componente ϕ supere el valor dado por el usuario como campo de vista vertical. De esta forma,

todo rayo que salga de la imagen y quede fuera del cilindro donde se proyecta el entorno no encontrará un valor determinado, dejando el espacio en negro, como se aprecia en la imagen 3.5. Por otro lado, en la figura 4.13 se puede ver cómo están definidos los campos de vista de una imagen cilíndrica.

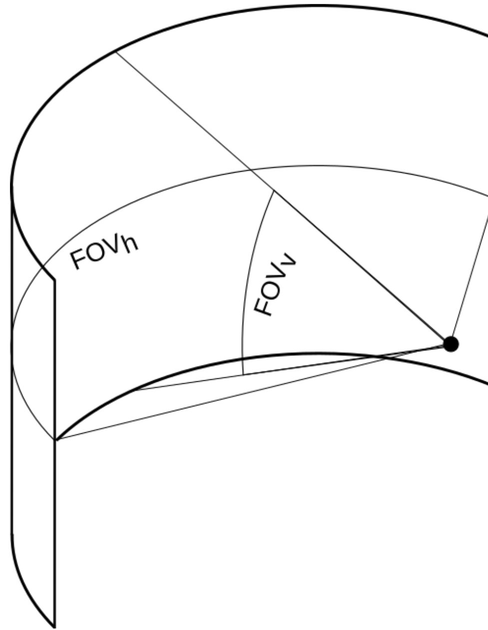


Figura 4.13: Campos de visión en imagen cilíndrica

De forma similar al modelo equirectangular, las siguientes líneas de código definen el modelo cilíndrico. El vector de salida definirá las coordenadas esféricas finales, las cuales se utilizarán para delimitar el campo de visión de este modelo.

```
#import numpy as np
def cilindrico(x, y, x_max, y_max, horizontal, vertical):
    theta = (2*x/float(x_max) - 1.0) * (horizontal/2.0)
    z_og = (1 - y/(y_max/2.0))
    phi = np.arcsin(z_og)
    x_og = math.cos(phi)*math.cos(theta)
    y_og = math.cos(phi)*math.sin(theta)
    vec = np.matrix('{};{};{}'.format(x_og,y_og,z_og))
    return vec
```

4.2.3 Ojo de pez

En el modelo de ojo de pez se debe tener en cuenta el tipo de lente que se va a simular. Puesto los 4 modelos de cámara de ojo de pez comparten modelo matemático, se implementan desde un mismo programa, definido con los mismos parámetros. La única diferencia entre ellos es la definición de una función geométrica que mapea el radio $\hat{r}$ al ángulo ϕ, la cual corresponde con el cuadro 4.1, obtenida de [23]. De esta función se obtendrá posteriormente el vector que marca la dirección del rayo que sale de la cámara.

Equi-angular	Estereográfica	Ortogonal	Equi-angulo solido
$\phi = \frac{\hat{r}}{f}$	$\phi = 2\arctan\left(\frac{\hat{r}}{2f}\right)$	$\phi = \arcsin\left(\frac{\hat{r}}{f}\right)$	$\phi = 2\arcsin\left(\frac{\hat{r}}{f}\right)$

Cuadro 4.1: Definición de ϕ para cada modelo de cámara de ojo de pez a partir del parámetro $\hat{r}$

El modelo de cámaras de ojo de pez está definido por dos parámetros, θ y ϕ. La coordenada esférica θ corresponde con la coordenada polar angular del píxel en la imagen destino respecto del punto principal (4.7). En este modelo, la coordenada ϕ se obtiene a partir del parámetro $\hat{r}$, la cual se puede obtener a partir de las coordenadas de píxel, como se muestra en la ecuación (4.8).

$$\theta = \arctan\left(\frac{x_0 - x}{y - y_0}\right) \tag{4.7}$$

$$\hat{r} = \sqrt{(x - x_0)^2 + (y - y_0)^2} \tag{4.8}$$

Posteriormente, calculadas las coordenadas esféricas, (θ, ϕ), se define el vector 3D con el que se buscará la intersección con el *cubemap*, que nos dará el píxel correspondiente de las imágenes que lo componen.

Hay que definir en este modelo un parámetro que se debe tener en cuenta en la composición final. Las imágenes de ojo de pez tienen simetría de revolución, por lo que el resultado que se obtiene es una imagen circular, como se puede ver en la figura 3.6. Por ello se define un parámetro r_{max}, que delimita para qué valores de $\hat{r}$ se tiene definida la imagen final.

Para la implementación del modelo de cámara de ojo de pez se definen dos funciones. La primera función es similar a la de modelos anteriores, donde se obtiene el vector que sale de la cámara a partir de las coordenadas de la imagen. En la segunda función se define cómo se obtiene el parámetro ϕ, el cual se ha visto que depende del tipo de cámara.

```
#import numpy as np
def ojo_de_pez(x, y, x_max, y_max, f, system):
  x_0 = x_max/2.0
  y_0 = y_max/2.0
  theta_hat = np.arctan2(x_0-x,y-y_0)
  r_hat = ((x-x_0)**2 + (y-y_0)**2)**0.5
  r_hat,PHI = cam_system(system,r_hat,f)
  x_cam = np.sin(PHI)*np.sin(theta_hat)
  y_cam = np.sin(PHI)*np.cos(theta_hat)
  z_cam = np.cos(PHI)
  vec_cam = np.matrix('{};{};{}'.format(x_cam,y_cam,z_cam))
  return r_hat, vec_cam

def cam_system(system,r,f):
  if system == 'equiang':
    return r,r/f
  elif system == 'stereo':
    return r,2*np.arctan2(r,2*f)
```

```
elif system == 'orth':
  aux = r/f
  if aux>1:
    r *= 10
    aux = 1
  return r,np.arcsin(aux)
elif system == 'equisol':
  aux = r/f
  if aux>1:
    r *= 10
    aux = 1
  return r,2*np.arcsin(aux)
else:
  print('Camera system ERROR')
```

4.2.4 Catadióptricas

En el modelo de cámaras catadióptricas se tiene una situación similar al modelo de ojo de pez. Se tiene un mismo modelo matemático con distintas definiciones para los parámetros, dependiendo del tipo de espejo. Como en el modelo anterior, se han implementado todas las imágenes catadióptricas centrales en un mismo programa, cambiando únicamente la definición de los parámetros para cada uno de los espejos.

En este modelo, los parámetros que cambian son ξ y η. En el simulador se han dejado para poder ser definidos por el usuario a través de la distancia con el espejo y el *latus-rectum*:

Para-catadióptrica	$\xi = 1$	$\eta = -2p$
Hiper-catadioptrica	$\xi = \frac{d}{\sqrt{d^2+4p^2}}$	$\eta = -\frac{2p}{\sqrt{d^2+4p^2}}$
Plano	$\xi = 0$	$\eta = 1$

donde el parámetro d es la distancia entre el espejo y la cámara perspectiva y 2p es el semi *latus-rectum* del espejo.

Definido el parámetro η, se puede calcular la matriz M_c y la matriz H_c, puesto se considera conocida la calibración de la cámara, representada por la matriz K_c. Conocidas estas matrices, las cuales definen parte del modelo matemático, se puede empezar a calcular el vector 3D con el que hacer la búsqueda en el *cubemap*.

Como se explicó en la sección 3.1.4, el proceso de composición sigue la matemática inversa a la del modelo, es decir, se parte de la imagen final y se calcula el vector de salida de la cámara. Partiendo de lo que se formula en [12] y en [29], primero se define el punto de la imagen final como en la ecuación (4.9). Las ecuaciones están definidas en el espacio proyectivo, por lo que se usan coordenadas homogéneas para pasar a otros espacios, como es el de la imagen final y el del punto 3D de la escena. En el inicio, no se tiene información de homogeneización en la imagen final, por lo que la coordenada $\hat{z}$,que definiría la homogeneización, se considera con valor unidad, obteniéndose el vector $\hat{v} = (\hat{x}, \hat{y}, \hat{z})^T = (u, v, 1)^T$, ya definido en el espacio normalizado proyectivo P^2.

$$p = \begin{pmatrix} u \\ v \end{pmatrix} \rightarrow \hat{v} = \begin{pmatrix} \hat{x}/\hat{z} \\ \hat{y}/\hat{z} \\ 1 \end{pmatrix} \tag{4.9}$$

A continuación, se calcula la matriz ${H_c}^{-1}$, cuya inversa define los parámetros intrínsecos del sistema catadióptrico. Como se puede ver en la ecuación (4.10), se pasa el punto obtenido de la imagen final en un plano normalizado proyectivo al plano proyectivo del modelo de la esfera.

$$\bar{v} = H_c^{-1}\big|_{3x3} \cdot \hat{v} \tag{4.10}$$

Una vez se ha obtenido el punto en el plano de la esfera, se aplica la transformación no lineal inversa a la definida en 3.1.4, definida como en la ecuación (4.11).

$$h^{-1}(\bar{v}) = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \bar{x}\frac{\bar{z}\xi + \sqrt{\bar{z}^2 + (1-\xi^2)(\bar{x}^2+\bar{y}^2)}}{\bar{x}^2+\bar{y}^2+\bar{z}^2} \\ \bar{y}\frac{\bar{z}\xi + \sqrt{\bar{z}^2 + (1-\xi^2)(\bar{x}^2+\bar{y}^2)}}{\bar{x}^2+\bar{y}^2+\bar{z}^2} \\ \bar{z}\frac{\bar{z}\xi + \sqrt{\bar{z}^2 + (1-\xi^2)(\bar{x}^2+\bar{y}^2)}}{\bar{x}^2+\bar{y}^2+\bar{z}^2} - \xi \end{pmatrix} \tag{4.11}$$

De esta ecuación se obtiene un vector 3D en la dirección de la cámara catadióptrica definida por los parámetros proporcionados. Como en casos anteriores, este vector se debe poner en contexto, es decir, rotarlo en la dirección en la que apunta la cámara dentro del escenario en el que se esté trabajando.

Obtenido el vector final, después de la rotación en la dirección de la cámara, se obtienen las coordenadas esféricas para empezar a buscar el píxel dentro del *cubemap*. En la función `catadioptrico` se puede ver cómo se programa la obtención de las coordenadas esféricas. Para la componente θ se deben hacer varias opciones, puesto existen puntos singulares, los cuales se pueden resolver conociendo el signo de las componentes del vector de la cámara.

Para la implementación del modelo catadióptrico, la definición del espejo se hace en la introducción de los parámetros del programa. En los códigos que se muestran, estos parámetros están incluidos en la matriz ${H_c}^{-1}$ y el parámetro `eps`. Como en el modelo anterior, se ha dividido en dos funciones el modelo. La principal obtiene el vector de salida de la cámara y sus coordenadas esféricas, mientras que la segunda función calcula valor de la función inversa $h^{-1}(\bar{v})$.

```
#import numpy as np
def catadioptrico(x, y, x_max, y_max, f, eps, Hc_inv, R):
  x_0 = x_max/2.0
  y_0 = y_max/2.0
  r_hat = ((x-x_0)**2 + (y-y_0)**2)**0.5
  vec_hat = np.matrix('{};{};{}'.format(x,y,1.0))
  vec_norm = Hc_inv * vec_hat
  vec_cam = h_inverse(vec_norm,eps)
  PHI = np.arccos(vec_cam[2,0])
  if vec_cam[1,0] == 0:
    if vec_cam[0,0] >= 0:
```

```
            theta = 0.0
        else:
            theta = -math.pi
    elif vec_cam[1,0] > 0:
        theta = np.arccos(vec_cam[0,0]/np.sin(PHI))
    elif vec_cam[1,0] < 0:
        theta = -np.arccos(vec_cam[0,0]/np.sin(PHI))
    return r_hat, theta, PHI, vec_cam

def h_inverse(vec,e):
    x_v = vec[0,0]
    y_v = vec[1,0]
    z_v = vec[2,0]
    norm = x_v**2 + y_v**2 + z_v**2
    sqrt = np.sqrt(z_v**2 + (1-e**2)*(x_v**2+y_v**2))
    x = ((z_v*e + sqrt)*x_v)/norm
    y = ((z_v*e + sqrt)*y_v)/norm
    z = ((z_v*e + sqrt)*z_v)/norm - e
    vector = np.matrix('{};{};{}'.format(x,y,z))
    return vector
```

4.2.5 Scaramuzza

El modelo de Scaramuzza sigue una estructura similar a los anteriores. Se parte de las coordenadas píxel de la imagen final, (x,y), y se calculan sus correspondientes al sistema de referencia que se tiene en el modelo. Puesto se está considerando que el centro de las imágenes es el origen coordenado, obteniéndose como en (4.12), el parámetro ρ definido en el modelo se consigue como en (4.14).

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \begin{pmatrix} \frac{x_{max}}{2} \\ \frac{y_{max}}{2} \end{pmatrix} \tag{4.12}$$

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} x - x_0 \\ y_0 - y \end{pmatrix} \tag{4.13}$$

$$\rho = \sqrt{u^2 + v^2} \tag{4.14}$$

Obtenido el parámetro ρ desde la imagen final, se calcula el valor de la función paramétrica $f(\rho)$ introduciendo los valores de calibración del modelo, los cuales deben ser conocidos. El valor de esta función dará el factor de homogeneización del vector del rayo de luz que sale, o entra, de la cámara, pudiendo obtener a posteriori la imagen del *cubemap* a la que apunta y obtener el valor de píxel que se debe definir en la imagen final. La obtención de este vector se puede ver en la ecuación (4.15), donde se construye y normalizan las coordenadas, para posteriormente rotarlas en la dirección de la cámara dentro de la escena y obtener las coordenadas esféricas.

$$v = \begin{pmatrix} \frac{u}{u^2+v^2+f(\rho)^2} \\ \frac{v}{u^2+v^2+f(\rho)^2} \\ \frac{f(\rho)}{u^2+v^2+f(\rho)^2} \end{pmatrix} \tag{4.15}$$

En la implementación del simulador, la cual se puede ver a continuación, se han dejado unos parámetros de calibración por defecto, sustituibles por el usuario, en el vector PARAM. También se ha implementado una función de grado 4, puesto según trabajos como [24], este grado de polinomio ajusta bastante bien a modelos de cámaras reales sin una gran necesidad computacional.

```
#import numpy as np
def scaramuzza(x, y, x_max, y_max, param, R):
    x_0 = x_max/2.0
    y_0 = y_max/2.0
    u = (x - x_0)
    v = (y_0 - y)
    rho = (u**2 + v**2)**0.5
    f_rho = param[0,0] + param[0,1]*rho + param[0,2]*rho**2 +
      + param[0,3]*rho**3 + param[0,4]*rho**4
    norm = (u**2 + v**2 + f_rho**2)**0.5
    x_cam, y_cam, z_cam = u/norm, v/norm, f_rho/norm
    vec_abs = np.matrix('{};{};{}'.format(x_cam,y_cam,z_cam))
    vec_cam = R * vec_abs
    theta = np.arctan2(vec_cam[1,0],vec_cam[0,0])
    PHI = np.arccos(vec_cam[2,0])
    return rho, theta, PHI, vec_cam
```

4.2.6 Intersección rayo-*cubemap*

Hasta la obtención de las coordenadas esféricas, cada uno de los modelos de cámara sigue un procedimiento distinto, puesto cada uno de los modelos utiliza una información ligeramente distinta. Sin embargo, una vez se adquieren los valores de las coordenadas esféricas y, por tanto, la dirección del rayo que se busca, todas las cámaras centrales siguen los mismos pasos en la composición.

En este punto, el vector que se obtiene está referenciado en el sistema de coordenadas de la cámara. Para poder hacer la intersección con el *cubemap* se debe hacer un cambio de sistema de coordenadas. Esto se consigue con la matriz de rotación definida por la orientación de la cámara dentro del entorno, es decir, la rotación de la cámara respecto del *cubemap*. Definido este giro, el vector resultante estará orientado en la misma dirección que se quiere para la cámara dentro del entorno.

Como implementación por defecto, se ha definido que la dirección principal de la cámara sea en el eje $+z$, pudiendo definir a partir de este una matriz de rotación para direccionar la cámara. En el simulador se ha implementado una función con matrices de rotación predefinidas para

direccionar la cámara en los ejes principales, $\pm x, \pm y, \pm z$, facilitando al usuario rotar la cámara fácilmente.

Una vez obtenido el vector definitivo, se recalculan las coordenadas esféricas, puesto ayudarán en el último paso de la composición, la selección de la imagen del *cubemap*. Aunque se ha definido el *cubemap* como un todo, en la realidad se guarda como 6 imágenes independientes, una en cada dirección de los ejes principales. Con el valor de las coordenadas esféricas, se puede delimitar que imágenes del *cubemap* pueden ser las que se encuentren en la 'zona de búsqueda' del rayo de luz para obtener el valor del píxel final. Esta 'zona de búsqueda' delimita una única dirección, positiva o negativa, de cada uno de los ejes principales, permitiendo utilizar el vector director para la selección de la imagen de la que se obtendrá el color del píxel.

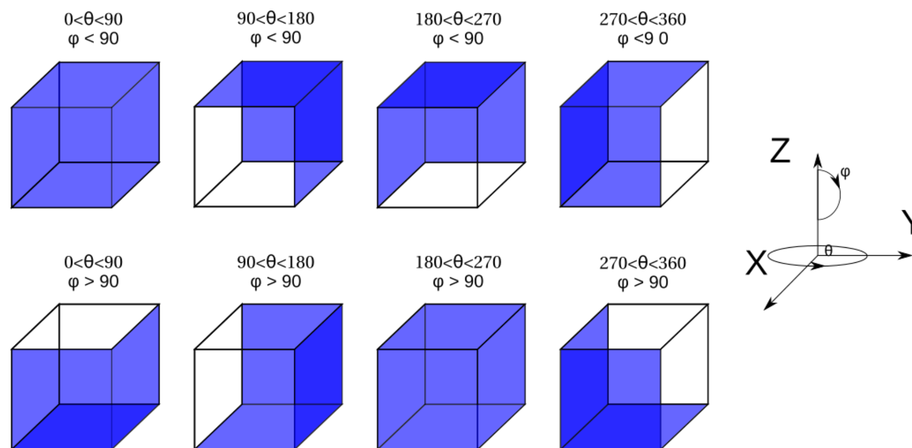


Figura 4.14: 'Zonas de búsqueda' dependiendo de los valores de las coordenadas esféricas

La selección de la imagen se hace proyectando el vector del rayo de luz en las 3 direcciones que han dejado la 'zona de búsqueda'. Puesto que se conoce en qué direcciones se toman las imágenes del *cubemap*, se conocen los vectores que definen los planos que estas imágenes definen. Tomando el mayor módulo de este vector proyectado se escoge la imagen a la que apunta el rayo de luz. El siguiente fragmento de código muestra cómo se ha implementado esta selección de imagen dentro de la zona de búsqueda.

```
def seleccion_imagen(vec, imagen1, R1, n1,
    imagen2, R2, n2, imagen3, R3, n3):
  u1 = vec -
    np.transpose(np.inner(np.dot(np.transpose(n1),vec),n1))
  d1 = np.sqrt(u1[0,0]**2 + u1[1,0]**2 + u1[2,0]**2)
  u2 = vec -
    np.transpose(np.inner(np.dot(np.transpose(n2),vec),n2))
  d2 = np.sqrt(u2[0,0]**2 + u2[1,0]**2 + u2[2,0]**2)
  u3 = vec -
    np.transpose(np.inner(np.dot(np.transpose(n3),vec),n3))
  d3 = np.sqrt(u3[0,0]**2 + u3[1,0]**2 + u3[2,0]**2)
  if d1 <= d2 and d1 <= d3:
    return imagen1, R1, n1
  elif d2 < d1 and d2 <= d3:
```

```
        return imagen2, R2, n2
    elif d3 < d1 and d3 < d2:
        return imagen3, R3, n3
```

Definida de qué imagen del *cubemap* se va a obtener el píxel para la imagen destino, queda cómo seleccionar este píxel. El primer paso de esta selección es rotar el vector para situarlo en el mismo plano que el definido por la imagen, definido en la ecuación (4.16). Como se conoce la orientación de todos los planos, se define una matriz de rotación para cada uno de ellos y se pre-multiplica el vector por esta matriz. Sin embargo, esta rotación no escala el vector al tamaño real de la imagen, por ello es necesario utilizar la matriz de proyección de la cámara perspectiva de UnrealCV, como se puede ver en la (4.17), donde queda un vector dirección con una última componente de escalado, con lo que se obtendrá el píxel que se quiere.

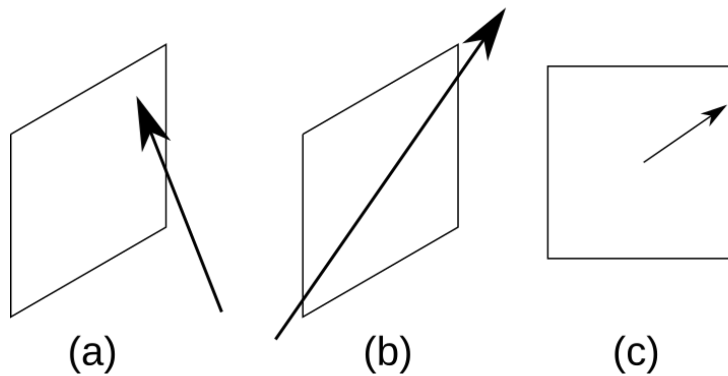


Figura 4.15: Selección del píxel: (a) vector 3D sobre plano de imagen; (b) vector en el mismo plano de imagen; (c) vector escalado en la imagen apuntando al píxel a seleccionar

$$R \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \tag{4.16}$$

$$K \cdot v_{ec} = \begin{pmatrix} f_x & s & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} u \\ v \\ k \end{pmatrix} \tag{4.17}$$

siendo f_x, f_y, s parámetros de calibración de la cámara virtual perspectiva simulada en UnrealCV y (u_0, v_0) las coordenadas del centro de coordenadas de la imagen respecto del origen de esta, normalmente en la esquina superior izquierda.

De esta forma, el píxel que se utiliza para rellenar la imagen final viene definido en coordenadas homogéneas por: $(u, v, k)^T$. Donde los valores de u y v definen la dirección desde el centro de coordenadas de la imagen y el valor k es un parámetro de escalado por el que se tendrá que dividir para obtener las coordenadas del píxel. Obtenido el valor de este píxel, se lleva a la imagen destino y se continúa el proceso para todos los píxeles de la imagen. Es un proceso muy repetitivo pero fácilmente paralelizable, puesto unos píxeles no dependen de otros.

Capítulo 5

Simulador de cámaras no centrales

La implementación del simulador en las cámaras no centrales es diferente a la de cámaras centrales. En este caso, no se puede obtener un único *cubemap* con el que obtener las imágenes a posteriori. Tampoco es eficiente guardar las capturas que se precisan para generar cada una de las imágenes. Estas diferencias hacen que se deba enfocar la adquisición de imágenes no centrales de forma distinta.

La composición de este tipo de imágenes se inicia igual que en las cámaras centrales, partiendo de la imagen final que se quiere componer. A partir de las coordenadas píxel de la imagen final, dependiendo el modelo de imagen no central que se esté modelando, se calculan las coordenadas del centro óptico que tendría ese píxel, realizando en ese momento la adquisición de imágenes para esa zona de la imagen no central. Obtenida la imagen del entorno, se seleccionan los píxeles de interés y se continúa barriendo la imagen final hasta que queda completada. Puesto que se está trabajando con imágenes con una resolución relativamente alta, 1024x512 píxeles o 1024x1024 píxeles dependiendo del modelo, es poco eficiente guardar cada una de las adquisiciones que se realizan, puesto que solo unos pocos píxeles de cada adquisición son válidos. Por esto, en este tipo de cámaras no se tendrá un *cubemap* o similar guardado, sino que se obtendrán las imágenes una a una.

Puesto que se tienen dos modelos muy diferenciados, panoramas y catadióptricas, se van a especificar cómo se consiguen por separado, definiendo los pasos y cómo se implementan los modelos matemáticos de cada uno de ellos.

5.1 Panorama no central

Las imágenes panorámicas no centrales se definen como una cámara de línea con el centro óptico distribuido en un círculo alrededor de un eje. Partiendo de la imagen final, se calcula la rotación de la cámara (5.1) y la posición desde donde se debe hacer la adquisición de la imagen (5.2). Se puede observar que el eje de giro que se utiliza es el eje z, vertical, y el círculo queda paralelo al suelo, dejando como trabajo futuro implementar esta inclinación respecto al suelo.

$$\theta = \left(\frac{2x}{x_{max}} - 1 \right) \pi \tag{5.1}$$

$$x_{foco} = X_{centro} + R_{adio} \cos\theta \tag{5.2}$$

$$y_{foco} = Y_{centro} + R_{adio} \sin\theta \tag{5.3}$$

$$z_{foco} = Z_{centro} \tag{5.4}$$

Situada la cámara en el centro óptico, se hace la adquisición en C como se ve en la figura 5.1. Esta adquisición no es una imagen completa, sino una franja de píxeles donde poder obtener la información necesaria para completar la imagen final. Además, tomar solo la franja de píxeles necesaria acelera el proceso de adquisición y composición, teniendo que almacenar una menor cantidad de datos, aunque sea solo de forma temporal.

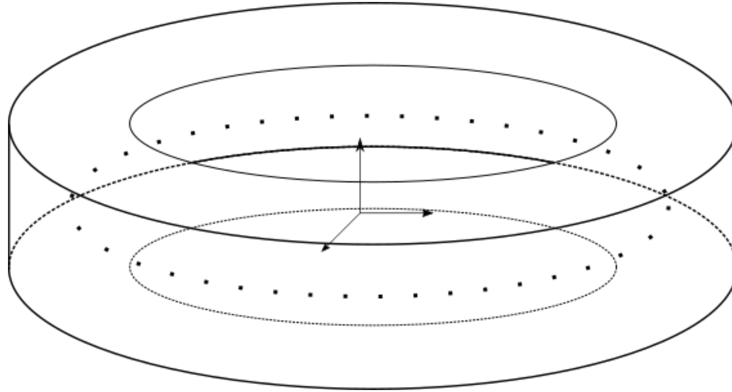


Figura 5.1: Panorama no central: perfil en C de revolución

La adquisición realizada es de 3 imágenes para cada posición de captura, una en el plano horizontal en la dirección de la cámara, definida por θ, y otras dos en las direcciones verticales, $+z$ y $-z$. Para saber de qué imagen se debe obtener la información, se sigue un proceso similar al definido en el simulador de cámaras centrales 4.2.6. Definiendo unas coordenadas esféricas θ, (5.1), y φ, (5.6), se obtiene el vector que "sale" de la cámara en cada centro óptico.

$$\theta = \left(\frac{2x}{x_{max}} - 1\right)\pi \tag{5.5}$$

$$\varphi = \left(1/2 - \frac{y}{y_{max}}\right)\pi \tag{5.6}$$

Obtenido este vector 3D, se va a la 'zona de búsqueda', la cual tendrá 2 zonas, x $\pm z$, en lugar de 3, x $\pm y$ $\pm z$, como pasaba en las cámaras centrales. Además, puesto se está orientando la cámara en cada θ, no se tendrá una componente y significativa en el vector que salga de la cámara, puesto está siendo anulada en cada adquisición.

Las condiciones definidas hacen que sea fácil continuar con la composición llegados este punto. Para una componente de θ definida por cada coordenada x de la imagen final, se tiene una adquisición de imágenes, definidas en franja, que corresponden con la variación de la componente esférica φ.

La repetición de este proceso, como se aprecia en las figuras 5.2 y 5.3, permite obtener una imagen panorámica no central alrededor de un punto con un radio definido. Su implementación queda reflejada en el anexo B, puesto que no existe una función principal que defina este modelo, sino que es un conjunto de funciones que entrelazan el modelo y la adquisición de las imágenes.

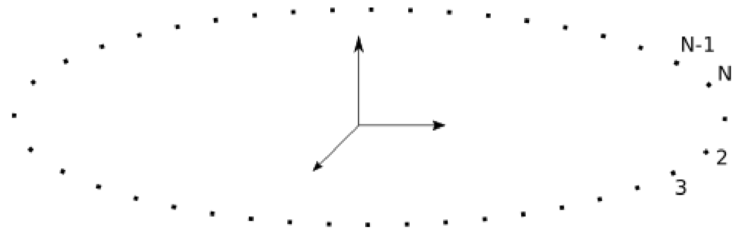


Figura 5.2: Recorrido de los centros ópticos de la cámara en la adquisición

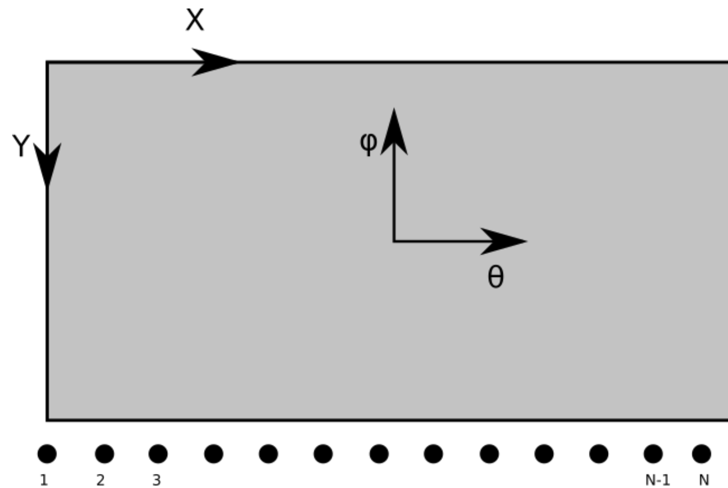


Figura 5.3: Coordenadas del panorama en la imagen junto a su adquisición

5.2 Catadióptricos no centrales

En 3.2.2 se ha explicado el modelo de las imágenes catadióptricas no centrales, explicando cómo se define y comportan los rayos de luz. En los espejos cónicos se ha definido una circunferencia por la que cruzan todos los rayos de luz, mientras que en el esférico no se ha podido determinar dónde se agrupan los focos. En el caso del simulador, se tiene que conocer dónde están dichos focos para poder usarlos como punto de adquisición de las imágenes para la composición de la imagen final.

Estudiando el modelo de catadióptricos no centrales, se puede apreciar que todos los rayos de luz atraviesan en eje de revolución del espejo. La intersección de este eje con el rayo de luz permite obtener un punto desde donde se pueden adquirir imágenes perspectivas para realizar una composición, y es este punto el que se va a buscar para realizar la adquisición y composición de las imágenes. Para ambos espejos, este punto se ha definido con la distancia Z_r desde el sistema de referencia absoluto. A partir de las ecuaciones que se han obtenido de los modelos, se va a obtener dicha distancia.

Para espejos cónicos, la obtención de la distancia Z_r se puede resolver como un problema geométrico. Puesto que es conocida la distancia Z_m al vértice del espejo y el ángulo de apertura del espejo, τ.

$$\cot\phi = \frac{1 + r\tan(2\tau)}{\tan(2\tau) - r} \tag{5.7}$$

$$Z_r = Z_c + R_c \cot\phi \tag{5.8}$$

donde R_c y Z_c se definieron en (3.17) y (3.18) respectivamente.

En el caso de los espejos esféricos, la matemática se complica un poco. En este caso se igualan los rayos que llegan al espejo y la cámara, ecuación (5.9), para obtener la intersección de estos. Esta intersección permite calcular los mismos parámetros que en el espejo cónico.

$$\begin{pmatrix} x\delta \\ y\delta \\ -\zeta \\ -yZ_s(\delta+\varepsilon) \\ xZ_s(\delta+\varepsilon) \\ 0 \end{pmatrix} = \begin{pmatrix} \sin\phi\cos\theta \\ \sin\phi\sin\theta \\ \cos\phi \\ -Z_r\sin\phi\sin\theta \\ Z_r\sin\phi\cos\theta \\ 0 \end{pmatrix} \tag{5.9}$$

Manipulando ligeramente estos vectores, dividiendo el primero por δ y el segundo por $\sin\phi$, se obtiene una relación directa de parámetros como se puede ver en (5.12) y (5.13).

$$x = \cos\theta \tag{5.10}$$

$$y = \sin\theta \tag{5.11}$$

$$\cot\phi = \frac{-\zeta}{\delta} \tag{5.12}$$

$$Z_r = Z_s\frac{(\delta+\varepsilon)}{\delta} \tag{5.13}$$

Los valores de δ (3.21), ε (3.22) y ζ (3.23) se definieron en la sección 3.2.2.

Obtenido el parámetro Z_r, se procede a la adquisición de imágenes. En este modelo, para cada posición de adquisición se crea un *cubemap* de el que se obtendrá la información para la composición de la imagen final. De cada punto de adquisición se utilizará únicamente un anillo de píxeles, pero como este anillo variará dependiendo del espejo y de la altura desde donde se tomen las imágenes, este *cubemap* será similar al de cámaras centrales, no pudiendo reducir su resolución para acelerar el proceso. Este inconveniente hace que la adquisición y composición de este tipo de imágenes sea mucho más lenta que el resto de las que se han explicado en este proyecto.

Debido al tipo de adquisición que se realiza y su obtención por anillos, la forma de composición de este tipo de imagen es totalmente diferente a las anteriores comentadas. En lugar de recorrer la imagen final píxel a píxel, se hace un recorrido radial. Se itera desde el centro de la imagen, radio unidad, hasta el borde, incrementando el valor del radio píxel a píxel. Para cada incremento, se calculan los parámetros $\cot\phi$ y Z_r y se hace la adquisición del *cubemap*. Posteriormente se hace un barrido angular para cada radio, donde se itera sobre el parámetro θ. De esta forma, se obtienen las coordenadas esféricas de forma previa a las coordenadas píxel de la imagen final. Para hacer un barrido lo más uniforme posible, se adapta el paso de iteración del ángulo con el radio, haciendo un paso más corto cuanto mayor es el radio, para evitar saltarse algún píxel. Con estas coordenadas esféricas se procede a obtener las coordenadas píxel, ecuaciones (5.14) y (5.15), y las coordenadas del vector 3D que saldría del espejo hacia el entorno, ecuación (5.16).

$$u = \frac{u_{max}}{2} + r\cos\theta \tag{5.14}$$

$$v = \frac{v_{max}}{2} - r\sin\theta \tag{5.15}$$

$$vec = \begin{pmatrix} \sin\phi\cos\theta \\ \sin\phi\sin\theta \\ \cos\phi \end{pmatrix} \tag{5.16}$$

Una vez obtenido el vector 3D se procede como en las cámaras centrales, es decir, se rota el vector en la dirección de la cámara dentro del entorno y se selecciona la imagen de la 'zona de búsqueda' de dónde obtener la información de píxel para cada vector.

Como en las imágenes panorámicas, no se tiene una pequeña función que recoja la implementación de este modelo, por lo que su programación completa se verá reflejada en el anexo C.

Capítulo 6

Evaluaciones

6.1 Evaluación

Llegado a este punto, se quieren evaluar las imágenes que se pueden obtener del simulador descrito. Para ello se van a utilizar dos herramientas de visión por computador:

- **CFL**: Corners for Layout [13] es un sistema de detección basado en Deep-learning que obtiene la distribución de una habitación a partir de una imagen equirectangular. Se compararán las imágenes sintéticas equirectangulares que se obtengan con imágenes de bases de datos existentes.
- **uncalibLineToolbox**: Se utilizará una herramienta de calibración y extracción de rectas en imágenes implementada en Matlab para comparar los parámetros de calibración de las imágenes catadióptricas y de ojo de pez sintéticas, conocidos, con los resultados de esta *Toolbox*

Con estos experimentos, no solo se quiere comprobar la capacidad de estas herramientas, sino comprobar que las imágenes sintéticas que se obtienen en este proyecto pueden competir con imágenes reales en su uso en algoritmos de visión por computador.

6.1.1 Corners for Layout

CFL es una red neuronal que toma como entrada imágenes equirectangulares de habitaciones o entornos cerrados y provee como salida la situación aproximada de las paredes que componen dicha habitación y sus esquinas. Siendo esta red entrenada con bases de imágenes existentes, se ha probado con pequeñas muestras de imágenes generadas con este simulador. Para poder hacer una comparación directa, se han tomado muestras de otras bases de imágenes, tanto de imágenes reales, *SUN360* [16], como de imágenes sintéticas, *STD2D3D* [15].

Un punto a tener en cuenta es la definición del *ground truth* de las imágenes. En las bases de imágenes *SUN360* y *STD2D3D*, la definición y etiquetado del *ground truth* se ha realizado de forma manual, hecho por una persona. En *SB3*, base de imágenes obtenidas en este proyecto, el *ground truth* se ha hecho de forma automática y sin supervisión. Esto permite obtener imágenes de forma más eficiente y sin probabilidad de error, puesto se tiene una precisión de píxel en toda la imagen, algo impensable si tiene que realizarlo una persona.

Otra consideración viene por las imágenes con las que CFL ha sido entrenado. Hasta el momento, solo se habían utilizado imágenes equirectangulares centradas en la habitación donde habían sido tomadas. Esto implica que no existen grandes distorsiones en las habitaciones, se mantienen las paredes a una distancia aproximadamente igual y se tiene una forma de cuboide, o caja de zapatos, más intuitiva, menos distorsionada. Además, todas las imágenes que se ha encontrado hasta ahora mantienen la vertical de la cámara con el eje de la gravedad.

Sin embargo, las imágenes que se han generado en este proyecto no siguen todas estas reglas. Se han hecho imágenes distribuidas por toda la habitación, a distintas alturas y con las paredes a diferentes distancias. Esto hace que la habitación adquiera distorsiones distintas y más variadas. Se han hecho rotaciones de cámara y se podrían hacer en cualquier eje, definiendo imágenes donde el techo y suelo no siempre estén en las partes superior e inferior de la imagen respectivamente. Sin embargo, esto último no se ha hecho en las imágenes a evaluar, puesto sabemos con certeza antes de realizarlo que los resultados no van a ser correctos.

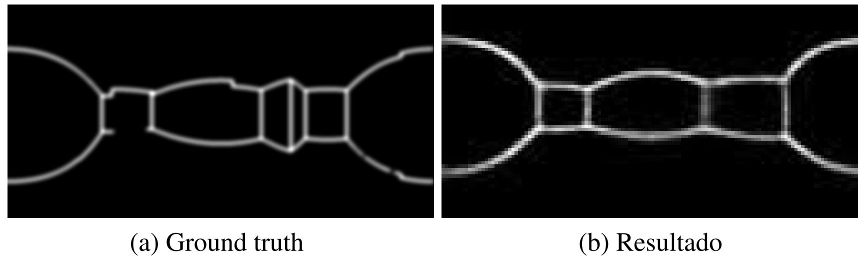
Para hacer la evaluación, se ha construido una muestra de imágenes desde el simulador, buscando una cierta variedad. Estas imágenes equirectangulares se han tomado desde las esquinas de la habitación y una en el centro de esta, variando la altura y orientación. Sin embargo, como se sabe a priori que únicamente está entrenada la red para imágenes en el mismo plano del suelo, no se han compuesto imágenes con rotaciones en ejes distintos al de la gravedad. Además de las imágenes, se ha generado automáticamente el *ground truth* para la intersección de paredes, EDGES, y para las esquinas, CORNERS.

Aunque se tiene una precisión de pixel en la generación del *ground truth*, se genera como una distribución Gaussiana para mejorar la precisión en los resultados, puesto la salida de CFL también es una distribución Gaussiana, marcando la incertidumbre de los resultados encontrados.

Obtenidas las imágenes y el *ground truth*, se procede a evaluar las imágenes sintéticas obtenidas de este simulador, obteniendose resultados como se ven en las figuras 6.2 y 6.3, obtenidos del panorama equirectangular 6.1.

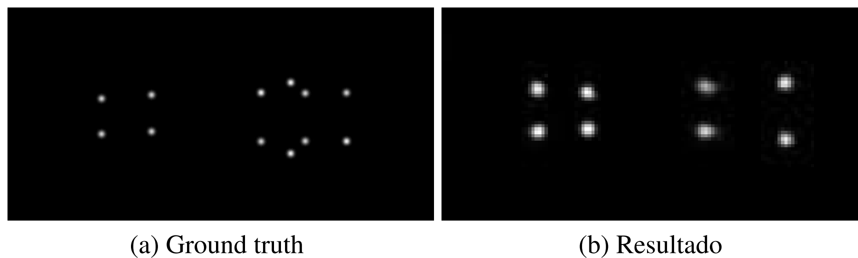
Figura 6.1: Imagen panorámica equirectangular evaluada en CFL

Se puede apreciar que los resultados obtenidos utilizando las distintas bases de imágenes son similares. Sin embargo, debido a que CFL no ha podido ser entrenada con un gran número de habitaciones de más de 4 paredes, encontrar más paredes le resulta complicado. En la figura



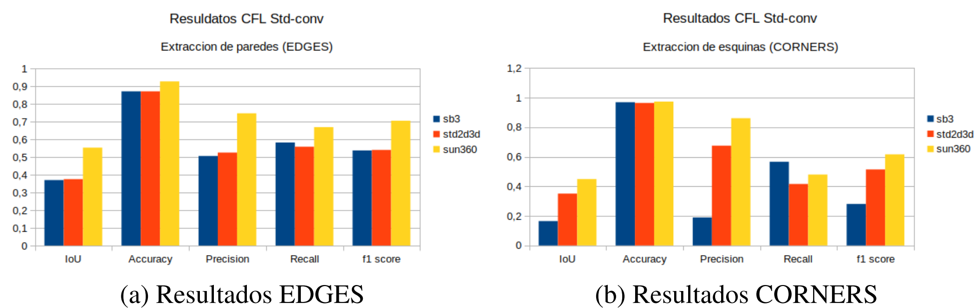
(a) Ground truth (b) Resultado

Figura 6.2: Evaluación de las paredes (*EDGES*) en CFL. (a) *ground truth*; (b) Resultado de la evaluación



(a) Ground truth (b) Resultado

Figura 6.3: Evaluación de las esquinas (*CORNERS*) en CFL. (a) *ground truth*; (b) Resultado de la evaluación



(a) Resultados EDGES (b) Resultados CORNERS

Figura 6.4: Comparación de resultados de la evaluación en CFL de imágenes de distintas bases

6.4 se puede ver una comparación directa de resultados entre las distintas bases de imágenes. Se aprecia que *SUN360* tiene los mejores resultados, lo cual es comprensible puesto que se ha entrenado la red con esta base de imágenes, y tiene ejemplos similares en las imágenes de evaluación. Por otro lado se aprecia que *Std-2D-3D* y *SB3*, la base de imágenes generada en este proyecto, presentan resultados similares en la parte de extracción de intersección de paredes, EDGES. En este caso, ambas bases son de imágenes virtuales, lo que permite hacer una comparación directa entre la herramienta presentada y bases de imágenes existentes.

En la extracción de esquinas, CORNERS, al tener un menor número de píxeles solución, si estos no son obtenidos de forma correcta se acusan más los fallos, haciendo que los resultados empeoren. Sin embargo, se puede ver que los resultados reales no distan mucho del *ground truth*.

Los resultados, figura 6.4, refuerzan la premisa de este proyecto: la falta de imágenes para

entrenamiento de redes neuronales y, en general, algoritmos de visión por computador.

Con una variedad suficiente de escenarios, se podría haber preparado una base de imágenes con una mayor variedad de entornos y configuraciones. Obtener imágenes de habitaciones de más de 4 paredes; distintas localizaciones dentro de las habitaciones, imágenes desde las esquinas, a ras de suelo o desde el techo; realizar giros de cámara en distintos ejes, distorsionando la línea de horizonte de las imágenes. Con una base de imágenes lo suficiente extensa y con estas variaciones en las imágenes, se podría haber entrenado la red neuronal para estar preparada para estas distorsiones.

6.1.2 Toolbox de calibración

Evaluadas las imágenes equirectangulares, se pasa a la evaluación de imágenes catadióptricas y dióptricas de ojo de pez. Para ello se ha utilizado una *Toolbox* diseñada en *Matlab* de extracción de rectas y parámetros de calibración a partir de las imágenes de una cámara de gran angular [18]. Esta herramienta carga una imagen y la procesa en función del modelo de proyección de la cámara que se quiera calibrar. Tras un primer pre-procesado, realiza una extracción de rectas en la escena. Dichas rectas aparecen deformadas por el modelo de cámara que se esté estudiando. En un postprocesado se obtiene un parámetro relacionado con la calibración de la cámara, desde el que se puede obtener la distancia focal a partir de las relaciones que aparecen en el cuadro 6.1.

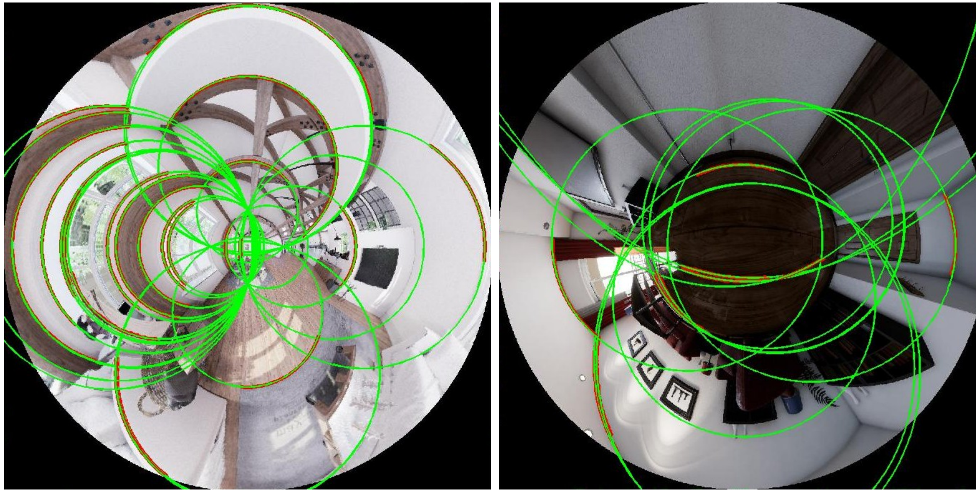
Tipo de cámara	$\hat{r}_{vl}$
Para-catadioptrica	$2fp$
Hiper-catadioptrica	$f\tan\chi$
Equi-angular ojo de pez	$f\frac{\pi}{2}$
Estereográfica ojo de pez	$2f$
Ortogonal ojo de pez	f
Equi-ángulo sólido ojo de pez	$f\frac{\sqrt{2}}{2}$

Cuadro 6.1: Relación entre $\hat{r}_{vl}$ y la distancia focal para cada tipo de cámara dióptrica y catadióptrica

Este parámetro $\hat{r}_{vl}$ se obtiene de la deformación de las rectas extraídas. Esto quiere decir que, cuanto mayor sea la deformación de las rectas que aparecen en la imagen, más precisa será la obtención de este parámetro. En las imágenes con las que se han hecho las pruebas, hay una variedad de localizaciones y parámetros para cada uno de los tipos de cámaras implementados. Esto permitirá observar qué precisión tiene esta herramienta, puesto que en estas imágenes sintéticas se puede controlar perfectamente su calibración, y se comparará con imágenes reales.

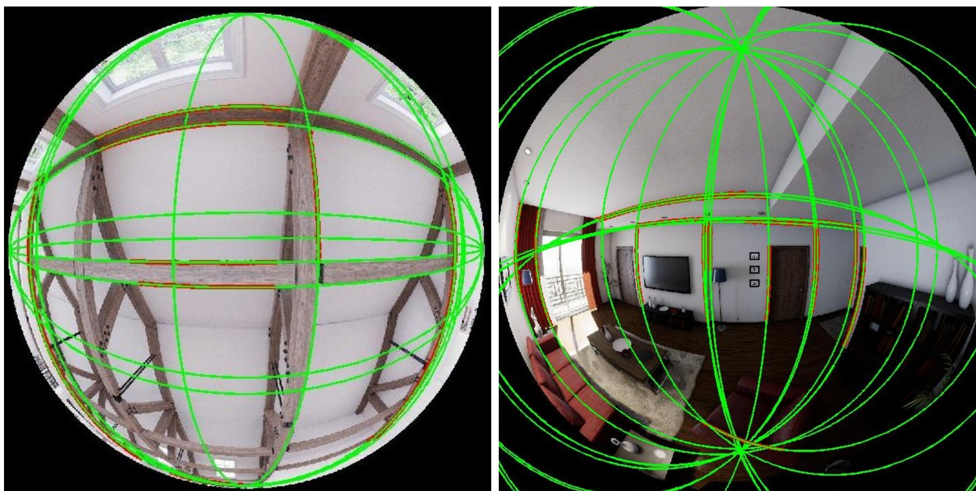
Empezando por las imágenes catadióptricas, la distancia al espejo y el *latus-rectum* los debe definir el usuario. Con estos parámetros se puede calcular p y χ. El parámetro f se ha definido para obtener resultados en el rango de estudio y poder compararlos con los obtenidos en [18].

Conocida la calibración de la cámara virtual, se procede a usar la *toolbox*, la cual hará una extracción de rectas y obtendrá los parámetros de calibración a partir de estas. En la figuras 6.5 y 6.6 se pueden ver ejemplos de las imágenes obtenidas tras la extracción de rectas para cámaras catadióptricas y de ojo de pez respectivamente.



(a) Para-catadióptrica con la extracción de lineas (b) Hiper-catadióptrica con la extracción de lineas

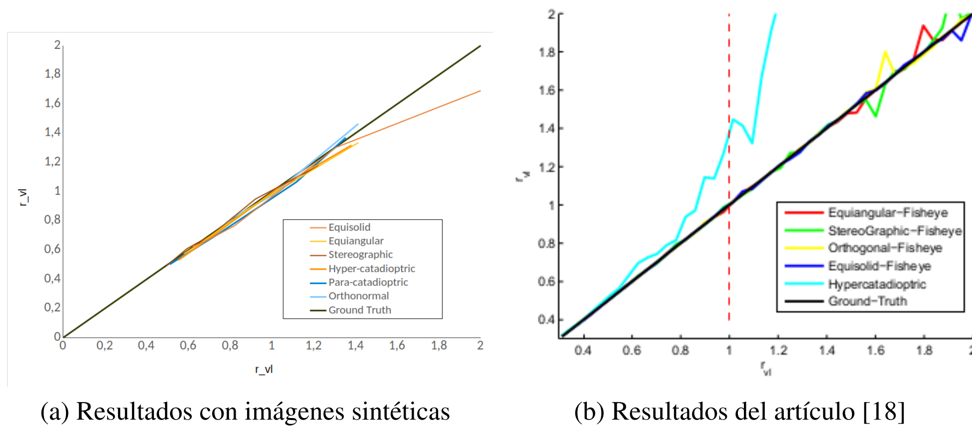
Figura 6.5: Ejemplo de extracción de líneas en imágenes catadióptricas



(a) Ojo de pez con la extracción de lineas (b) Ojo de pez con la extracción de lineas

Figura 6.6: Ejemplo de extracción de líneas en imágenes de ojo de pez

Realizada la calibración, se realiza una normalización del parámetro $\hat{r_{vl}}$ para poder comparar fácilmente los resultados obtenidos con los del artículo [18]. Esta normalización consiste en dividir por la diagonal de la imagen para conseguir $\hat{r}_{vl} = 1$ en la esquina de la imagen, permitiendo así hacer una comparación entre imágenes de distinta resolución.



(a) Resultados con imágenes sintéticas (b) Resultados del artículo [18]

Figura 6.7: Comparativa de resultados: a) Resultados obtenidos con imágenes sintéticas de este simulador; b) Resultados expuestos en [18]

A la vista de los resultados obtenidos en la calibración, expuestos en la figura 6.7, se puede determinar que las imágenes sintéticas obtenidas pueden ser fácilmente comparables con las imágenes reales para realizar una calibración de cámaras catadióptricas y dióptricas de ojo de pez.

Sin embargo, en las cámaras hiper-catadióptricas se tienen discrepancias entre los resultados reales y los sintéticos. Pueden existir dos razones para estas discrepancias:

1º En las imágenes sintéticas no se tiene la reflexión de la cámara, mientras que en las imágenes reales en inevitable. Esto significa que en las imágenes sintéticas catadióptricas se tiene más información del entorno que en las imágenes reales, puesto no queda ninguna dirección ocluida por la cámara que toma la imagen. Tener más información puede derivar en mejores resultados en la calibración.

2º En las pruebas realizadas con cámaras y espejos reales, se utilizó una mayor variedad de espejos hiperbólicos, utilizando algunos con $\xi < 0{,}7$, mientras que en las imágenes sintéticas se han utilizado valores de $\xi > 0{,}8$. En la realización de imágenes sintéticas se acotó este parámetro porque se tenía constancia previa de peores resultados cuanto menor era el parámetro ξ.

Estos resultados muestran que las imágenes sintéticas realizadas en este proyecto son comparables con imágenes de cámaras reales en cuanto a calibración.

Capítulo 7

Conclusiones

7.1 Conclusión

La principal contribución de este trabajo es la creación de una herramienta para la composición de imágenes de cámaras omnidireccionales y 360° centrales y no centrales. Con su implementación se puede generar bases de imágenes sintéticas fotorrealistas, con etiquetado automático, para su utilización en algoritmos de visión por computador. En este simulador se han implementado una gran variedad de modelos de proyección de cámaras centrales y no centrales, superando a las bases de imágenes y otras herramientas que se puedan encontrar en el estado del arte. Se han incluido modelos de imágenes panorámicas equirectangulares y cilíndricas centrales y panoramas no centrales; modelos catadióptricos para espejos con un centro óptico único, espejos parabólicos e hiperbólicos, y para espejos con el centro óptico distribuido en una trayectoria o superficie, espejos cónicos y esféricos; modelos dióptricos para lentes de ojo de pez, divididos en distintos modelos de proyección; y un modelo empírico que aproxima imágenes catadióptricas, el modelo de Scaramuzza.

La evaluación de diversos algoritmos de visión utilizando las imágenes generadas ha evidenciado la necesidad de bases de imágenes especializadas en el campo de la visión por computador. Se ha visto que la falta de variedad en algunas bases de imágenes dificulta o limita el rendimiento de algunos algoritmos de visión por computador, como se ha mostrado en la sección 6.1.1. En muchas ocasiones, las bases de imágenes quedan limitadas por la necesidad del etiquetado manual de cada una de las imágenes, limitante que se puede solventar con la herramienta presentada.

Por otro lado, partir de un motor gráfico que ofrece la posibilidad de generar espacios fotorrealistas permite utilizar las imágenes creadas como sustituto de imágenes reales. En la sección 6.1.2 se ha demostrado que las imágenes sintéticas consiguen iguales resultados, incluso mejores, en herramientas de calibración de cámaras de gran angular y algoritmos de extracción de rectas. De esta forma, su uso puede ser determinante para la evaluación de futuras herramientas de calibración de cámaras y algoritmos de extracción de rectas.

7.2 Trabajo Futuro

Una línea de trabajo futuro pasaría por optimizar el código. En el simulador de cámaras centrales, la composición es un proceso altamente paralelizable. Actualmente, la carga de trabajo ha sido casi exclusivamente para la CPU del ordenador. Aprovechando la capacidad que tiene una GPU para realizar tareas en paralelo, reprogramar el simulador para que pueda trabajar en este medio aceleraría el proceso de composición de las imágenes, reduciendo el tiempo de computación total. Además de mejorar la implementación del simulador, se podría incluir una inclinación en el modelo de panorama no central.

Por otro lado, el diseño modular de la implementación permite la inclusión de nuevos modelos de proyección centrales, entre los cuales se está considerando incluir el modelo de ojo de pez de Kannala-Brandt. Además, crear nuevas funciones de etiquetado y generación de *ground truth* para distintos algoritmos de visión permitirá a un mayor número de investigadores el uso de este simulador.

Otra línea de trabajo es la generación de una base de imágenes lo suficientemente grande para poder ser utilizada por algoritmos basados en deep-learning. Para ello es necesario adquirir nuevos escenarios, con la mayor diversidad de objetos y distribución en planta de habitaciones. Estudiar un procedimiento donde modificar los escenarios de forma semi-aleatoria, cambiando distribuciones en planta y posición del mobiliario y demás objetos sería una línea de trabajo interesante.

Bibliografía

[1] M. Johnson-Roberson, C. Barto, R. Mehta, S. N. Sridhar, K. Rosaen, and R. Vasudevan, "Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks?," *arXiv preprint arXiv:1610.01983*, 2016.

[2] A.-D. Doan, A. M. Jawaid, T.-T. Do, and T.-J. Chin, "G2d: from gta to data," *arXiv preprint arXiv:1806.07381*, 2018.

[3] M. Angus, M. ElBalkini, S. Khan, A. Harakeh, O. Andrienko, C. Reading, S. Waslander, and K. Czarnecki, "Unlimited road-scene synthetic annotation (ursa) dataset," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pp. 985–992, IEEE, 2018.

[4] S. R. Richter, V. Vineet, S. Roth, and V. Koltun, "Playing for data: Ground truth from computer games," in *European conference on computer vision*, pp. 102–118, Springer, 2016.

[5] F. S. Saleh, M. S. Aliakbarian, M. Salzmann, L. Petersson, and J. M. Alvarez, "Effective use of synthetic data for urban scene semantic segmentation," in *European Conference on Computer Vision*, pp. 86–103, Springer, 2018.

[6] S. R. Richter, Z. Hayder, and V. Koltun, "Playing for benchmarks," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2213–2222, 2017.

[7] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," *arXiv preprint arXiv:1711.03938*, 2017.

[8] G. Ros, L. Sellart, J. Materzynska, D. Vazquez, and A. M. Lopez, "The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3234–3243, 2016.

[9] J. McCormac, A. Handa, S. Leutenegger, and A. J. Davison, "Scenenet rgb-d: Can 5m synthetic images beat generic imagenet pre-training on indoor segmentation?," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2678–2687, 2017.

[10] W. Qiu, F. Zhong, Y. Zhang, S. Qiao, Z. Xiao, T. S. Kim, Y. Wang, and A. Yuille, "Unrealcv: Virtual worlds for computer vision," *ACM Multimedia Open Source Software Competition*, 2017.

[11] L. Puig, Y. Bastanlar, P. Sturm, J. J. Guerrero, and J. Barreto, "Calibration of central catadioptric cameras using a dlt-like approach," *International Journal of Computer Vision*, vol. 93, no. 1, pp. 101–114, 2011.

[12] L. Puig, J. Bermúdez, P. Sturm, and J. J. Guerrero, "Calibration of omnidirectional cameras in practice: A comparison of methods," *Computer Vision and Image Understanding*, vol. 116, no. 1, pp. 120–137, 2012.

[13] C. Fernandez-Labrador, J. M. Facil, A. Perez-Yus, C. Demonceaux, J. Civera, and J. J. Guerrero, "Corners for layout: End-to-end layout recovery from 360 images," *arXiv preprint arXiv:1903.08094*, 2019.

[14] A. Rituerto, L. Puig, and J. J. Guerrero, "Visual slam with an omnidirectional camera," in *2010 20th International Conference on Pattern Recognition*, pp. 348–351, IEEE, 2010.

[15] I. Armeni, S. Sax, A. R. Zamir, and S. Savarese, "Joint 2d-3d-semantic data for indoor scene understanding," *arXiv preprint arXiv:1702.01105*, 2017.

[16] J. Xiao, K. A. Ehinger, A. Oliva, and A. Torralba, "Recognizing scene viewpoint using panoramic place representation," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2695–2702, IEEE, 2012.

[17] A. Eichenseer and A. Kaup, "A data set providing synthetic and real-world fisheye video sequences," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1541–1545, IEEE, 2016.

[18] J. Bermudez-Cameo, G. Lopez-Nicolas, and J. J. Guerrero, "Line extraction in uncalibrated central images with revolution symmetry.," in *BMVC*, 2013.

[19] V. Autores, "Unreal engine 4 documentation." `https://docs.unrealengine.com/en-US/index.html`, 2019.

[20] J. Bermudez-Cameo, G. Lopez-Nicolas, and J. J. Guerrero, "Automatic line extraction in uncalibrated omnidirectional cameras with revolution symmetry," *International Journal of Computer Vision*, vol. 114, no. 1, pp. 16–37, 2015.

[21] J. Bermudez-Cameo, G. Lopez-Nicolas, and J. J. Guerrero, "Fitting line projections in non-central catadioptric cameras with revolution symmetry," *Computer Vision and Image Understanding*, vol. 167, pp. 134–152, 2018.

[22] J. B. Cameo, *Line-projection in omnidirectional vision: modelling, extraction and calibration in central and non-central cameras*. PhD thesis, Universidad de Zaragoza, 2016.

[23] J. Bermudez-Cameo, G. Lopez-Nicolas, and J. J. Guerrero, "A unified framework for line extraction in dioptric and catadioptric cameras," in *Asian Conference on Computer Vision*, pp. 627–639, Springer, 2012.

[24] D. Scaramuzza, A. Martinelli, and R. Siegwart, "A flexible technique for accurate omnidirectional camera calibration and structure from motion," in *Fourth IEEE International Conference on Computer Vision Systems (ICVS06)*, pp. 45–45, IEEE, 2006.

[25] D. Scaramuzza, A. Martinelli, and R. Siegwart, "A toolbox for easily calibrating omnidirectional cameras," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5695–5701, IEEE, 2006.

[26] J. Bermudez-Cameo, O. Saurer, G. Lopez-Nicolas, J. J. Guerrero, and M. Pollefeys, "Exploiting line metric reconstruction from non-central circular panoramas," *Pattern Recognition Letters*, vol. 94, pp. 30–37, 2017.

[27] J. Bermudez-Cameo, G. Lopez-Nicolas, and J. J. Guerrero, "Line-images in cone mirror catadioptric systems," in *2014 22nd International Conference on Pattern Recognition*, pp. 2083–2088, IEEE, 2014.

[28] J. Guerrero-Viu, C. Fernandez-Labrador, C. Demonceaux, and J. J. Guerrero, "What's in my room? object recognition on indoor panoramic images," *arXiv preprint arXiv:1910.06138*, 2019.

[29] J. B. Cameo, "Calibración de sistemas catadióptricos de visión omnidireccional," 2009.