Escuela de Doctorado
y Estudios de Posgrado
Universidad de La Laguna

Trabajo de Fin de Máster

Máster Universitario en Desarrollo de Videojuegos

Captura volumétrica en videojuegos

Volumetric capture in videogames

Germán Alfonso Teixidó

La Laguna, 28 de junio de 2021

Dª **Isabel Sánchez Berriel**, con N.I.F. 42.855.838-S profesora Contratada Doctora adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutora

C E R T I F I C A

Que la presente memoria titulada:

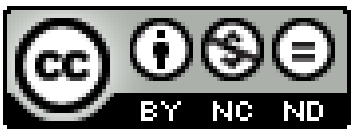
"Captura volumétrica en Videojuegos"

ha sido realizada bajo su dirección por D. **Germán Alfonso Teixidó**,

con N.I.F. 42238605W.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 28 de junio de 2021.

Licencia



Resumen

Este TFM ha sido realizado como máster de prácticas de empresa en Arquimea Centro de Investigaciones Avanzadas. El objetivo inicial de las prácticas consiste en crear diferentes aplicaciones con el fin de explorar mundos virtuales y/o modelos de objetos en realidad virtual capturados en el mundo real mediante el uso del método llamado captura volumétrica neural que está siendo desarrollado actualmente en la compañía.

Para ello se han desarrollado un conjunto de diferentes pruebas de concepto que se han utilizado para estudiar la viabilidad de las técnicas utilizadas en el desarrollo de videojuegos, así como en otros campos donde el uso de un motor gráfico sea necesario y pueda verse beneficiado por la nueva tecnología.

Abstract

This TFM has been carried out as an internship at Arquimea Research Center. The initial goal of the internship is to create different applications in order to explore virtual worlds and/or models of objects in virtual reality captured in the real world by using the method called neural volumetric capture that is currently being developed in the company.

For this purpose, a different set of proof-of-concept tests have been developed and used to study the feasibility of the techniques used in the development of video games, as well as in other fields where the use of a graphics engine is necessary and can be benefited by the new technology.

Índice

Tabla de Imágenes:

Capítulo 1
Introducción

Este TFM ha sido realizado como máster de prácticas de empresa en **Arquimea Centro de Investigaciones Avanzadas**. El objetivo de las prácticas consiste en crear diferentes aplicaciones con el fin de explorar mundos virtuales y/o modelos de objetos en realidad virtual capturados en el mundo real mediante el uso del método llamado **captura volumétrica neural (en sus siglas en inglés, NVC)**, que está siendo desarrollado actualmente en la compañía. Este novedoso método de captura de la volumetría se basa en el uso de modelos de Inteligencia Artificial (AI) – específicamente, de aprendizaje profundo – con el fin de aprender la geometría, comportamiento de la luz, y color de escenas reales tridimensionales (3D), para luego poder renderizar vistas de esta desde puntos de vista novedosos arbitrarios.

1.1 Objetivos

El objetivo del proyecto es el diseño y estudio de rendimiento de diferentes pruebas de concepto (PoC) que permitan visualizar en un motor gráfico y desde puntos de vista arbitrarios, objetos o escenas capturadas en un modelo volumétrico neural; estos modelos son construidos entrenando redes neuronales profundas. Nuestra motivación es insertar estas capturas neurales de objetos o escenarios del mundo real en mundos virtuales, y visualizarlos de una manera integrada con los elementos y experiencia del entorno virtual. Se trabajará usando el motor gráfico Unreal Engine 4 (UE4).

Para sobrellevar la imposibilidad de ejecutar el modelo en el orden de los milisegundos y por lo tanto en tiempo real, hemos tomado la aproximación de pre-renderizar un conjunto de vistas de la escena, de forma que luego puedan ser seleccionadas según el punto de vista del observador del objeto u escena. El número de imágenes (vistas noveles) y las posiciones y ángulos o poses de cámara de cada una de estas vistas pre-generadas dependerá del tipo de captura y del tipo visor / experiencia que queramos reproducir en el mundo virtual.

1.2 ¿Qué es NVC?

NVC propone el uso de una red neuronal profunda y técnicas de aprendizaje profundo para aprender escenas tridimensionales reales, con el fin de renderizar vistas noveles de la misma, desde puntos de vista arbitrarios. La red mapea puntos del espacio tridimensional (ubicación espacial) y la dirección desde la que lo observamos (pose de la cámara), como una entrada 5D, al color RGB y la opacidad (salida 4D) de ese punto en el espacio tridimensional capturado.

NVC nos permite crear modelos 3D implícitos, basados en redes neuronales profundas (no modelos 3D convencionales, como los basados en mallas o vóxeles) a partir de un número reducido de fotografías de la escena tridimensional; más tarde usaremos ese modelo para generar imágenes nóveles arbitrarias de ese mismo espacio. Sus aplicaciones son similares a las de la fotogrametría, ya que nos permite capturar modelos tridimensionales a partir de fotografías para luego situarlos en otros espacios. Cada modelo (red neuronal profunda) entrenado permite generar diferentes vistas del modelo que representan, con mayor o menor flexibilidad y calidad en función del número y resolución de imágenes de entrada, y el tipo y duración del entrenamiento realizado. Otro factor clave es el ángulo de captura de la escena. Podemos entrenar diferentes perspectivas, desde un escenario a una visión en 180º a un objeto en 360º.

Ejemplo de vistas generadas a partir de modelos NVC

1.2.1: A la derecha, el ejemplo de un pinar visto desde diferentes ángulos y posiciones. A la izquierda, un modelo de una estatua vista desde diferentes ángulos. La libertad de ángulo y posición que podemos requerir en nuestra nueva vista y la calidad y fidelidad de la imagen renderizada dependerá del tiempo de entrenamiento del modelo, vistas con las que ha sido entrenado, etc. Si no entrenamos nuestros modelos adecuadamente, las vistas producirán artefactos.

NVC está implementado en Python y sobre el framework de aprendizaje profundo PyTorch. Tanto el entrenamiento como la pre-generación de imágenes se realiza en estaciones de trabajo de alto rendimiento con unidades de procesamiento gráfico (GPUs) Nvidia Geforce RTX3090.

En este proyecto no profundizaremos en el funcionamiento o prestaciones de NVC, sino en la aplicabilidad y utilidad del método en la generación de contenido para realidad virtual (VR); así, usaremos NVC como una suerte de caja negra para la implementación de las pruebas de concepto requeridas por la empresa. Para la comprensión de los objetivos, resultados y conclusiones del presente proyecto, tan solo es necesario entender que lo utilizaremos para construir un modelo de una escena u objeto, al partir del cual pre-generaremos n vistas del mismo, para simular un objeto tridimensional en un entorno virtual con la intención de lidiar con un menor coste de GPU que con el render (que implicaría la ejecución en tiempo real del modelo neural) de una escena u objeto 3D.

NVC en acción

1.2.2: Algunos ejemplos de vistas generadas por modelos NVC.
Estas vistas han sido generadas por nuestra IA, entrenada al partir de algunas fotos de la escena.

1.3 Formato del visor

Gracias al método NVC es posible entrenar un modelo neural (red neuronal artificial) para aprender la volumetría y posteriormente renderizar vistas de escenas u objetos desde puntos de vista arbitrarios. En realidad, no solo tenemos que limitarnos a crear modelos NVC con capturas del mundo real, sino que también podríamos aprender la volumetría y generar vistas de objetos y escenas CGI. Así, en teoría, se plantea que podríamos llegar a utilizar un modelo neural -implícito por naturaleza- en vez del modelo explícito 3D convencional (basado en mallas o vóxeles), dentro de un motor gráfico, mostrando las vistas generadas por nuestro modelo NVC mediante renderizado neural en lugar de usar el renderizando 3D tradicional. Una de las principales ventajas sería que nuestro modelo neural sería mucho más ligero que un objeto tridimensional al uso y sería mucho más ligero de ejecutar en GPU al mostrarse por pantalla.

Un modelo NVC puede tardar horas (o días) en ser entrenado y optimizado, por lo que no es factible que su entrenamiento se realice *in-engine* (dentro del motor gráfico). Por ende, la idea original consiste en importar a Unreal Engine 4 un modelo ya entrenado que ejecutaríamos para obtener las vistas que deseemos mostrar en tiempo de ejecución. Este modelo nos devolvería imágenes con las vistas que solicitemos, por lo que tendremos que solicitarle la vista desde la cámara a dicho objeto en cada *frame*/ciclo de reloj (Denominado '*tick*' en Unreal Engine 4).

Idealmente, el flujo de trabajo debería simplificarse a: entrenar un modelo NVC, importarlo a UE4 y añadirlo a la escena como un Actor. Preferiblemente, deberíamos desarrollar un Actor especializado en interpretar el modelo generado y devolver la vista correspondiente respecto a la actual posición y rotación del objeto, la posición de la cámara, etc. A la hora de generar una vista en tiempo real, existen dos aproximaciones:

- A cada *tick*, generamos la imagen correspondiente a la vista que queremos mostrar a partir de nuestro modelo, y la mostramos desde nuestro Actor (mayor consumo de tiempo de reloj y vista más precisa).
- Al comenzar la ejecución, generamos una gran cantidad de vistas de nuestro modelo y las guardamos en una matriz de vistas. A cada *tick*, mostramos la imagen de nuestra matriz que más se aproxime a la vista que queremos mostrar (menor consumo de tiempo de reloj y vista menos precisa).

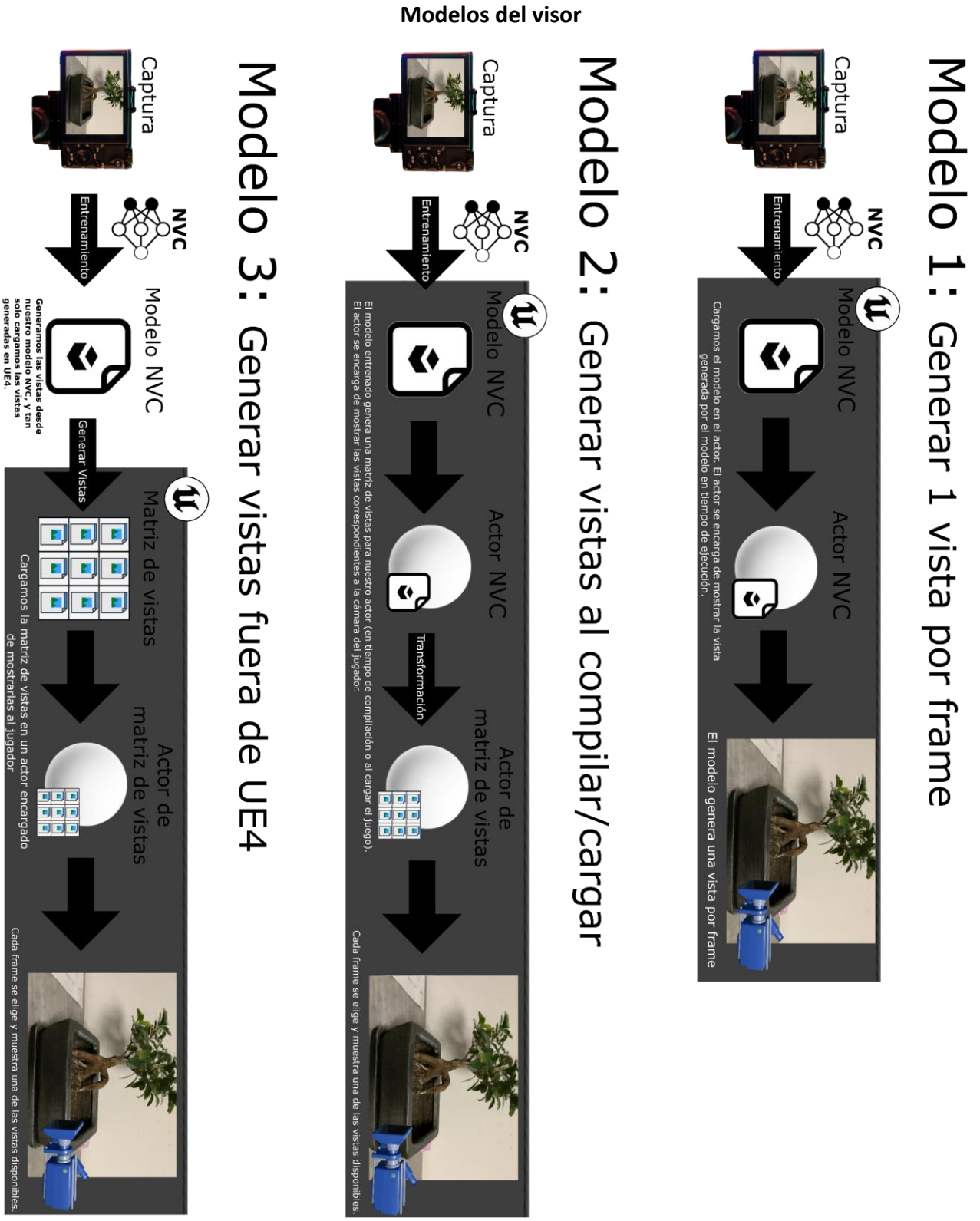
Lamentablemente, durante la realización del periodo de prácticas de TFM, ninguna de estas aproximaciones era posible:

- Por un lado, la inferencia o ejecución del modelo para renderizar vistas no es tiempo real; la mayor velocidad que se ha obtenido para generar una imagen 1080p de la vista es de 500-600 milisegundos en una GPU RTX3090. Por lo que la primera aproximación era inútil en esos momentos.
- La segunda aproximación era más viable, pero también requeriría de un desarrollo de código para adaptar los modelos generados a UE4 y poder generar, guardar y mostrar las vistas en el motor gráfico, para lo cual no había tiempo para su implementación durante la duración de las prácticas.

Afortunadamente, es posible imitar la segunda implementación planteada pre-generando las vistas de la matriz desde el modelo fuera de UE4, importando estas vistas al motor gráfico, y trabajando con estas imágenes. En resumen, nuestra solución consiste en:

- Importamos una matriz de imágenes al motor gráfico, que cargamos en un actor.
- Programamos este actor para que a cada *tick* muestre la imagen de dicha matriz que más se aproxime a la vista que queremos mostrar.

En la siguiente imagen se explica de forma más detallada la implementación de cada modelo:



1.3.1: Resumen visual de los tres posibles modelos considerados.
(El Modelo 3 es el que aplicaremos a nuestras pruebas de concepto en esta defensa.
El Modelo 1 sería el objetivo final si se alcanzase una generación de vistas en tiempo real).

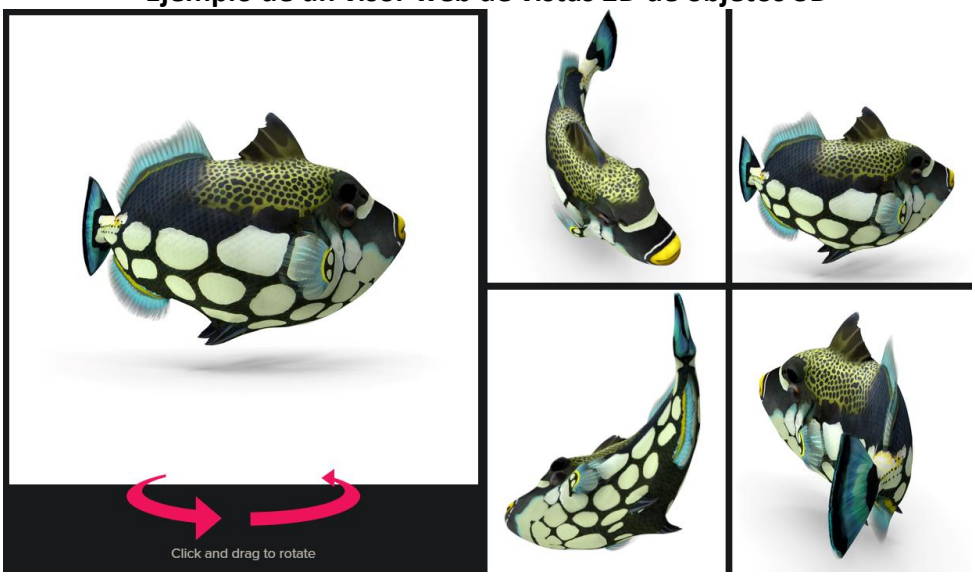
En definitiva, el “modelo” que terminaremos importando a nuestro motor gráfico simplemente consistirá en una matriz de imágenes pre-generadas de una escena, de una resolución variable, que serían generadas previamente mediante la ejecución (inferencia) de un modelo neural NVC entrenado con imágenes de dicha escena.

Es más, esta aproximación nos permite además modelar un objeto 3D en Blender y realizar capturas de este desde diferentes ángulos para emular el resultado de las vistas generadas por un modelo NVC entrenado, ahorrándonos la necesidad de entrenar un modelo neural a la hora de realizar pruebas de la solución en las diferentes pruebas de concepto.

Generalmente, trabajaremos con una matriz de imágenes de tamaño 20x20. Nuestro objetivo es crear actores en Unreal Engine 4 encargados de mostrar a cámara/el jugador una vista de un objeto o un escenario, y cambiar dicha vista cuando sea necesario al reaccionar frente al input del jugador (ya sea al desplazarse, interactuar con este, etc).

Mostrar diferentes vistas de un objeto 3D en un visor no es una idea nueva, y ya existen diferentes visores que usan este formato con diferentes fines. La meta no es reinventar la rueda, sino adaptar este formato a un motor gráfico para simular escenarios y objetos 3D capturados por nuestro modelo para obtener un menor coste de GPU.

Ejemplo de un visor web de vistas 2D de objetos 3D



1.3.2: En la imagen podemos apreciar el visor de vistas 2D usado por PixelSquid. Mostrar diferentes ángulos de un objeto puede servir para ofrecer a artistas imágenes de un objeto que buscan en un ángulo determinado, o previsualizar un objeto 3D en una librería online sin los costes asociados a tener que descargar el modelo ni renderizarlo en un navegador.

El alcance del proyecto está orientado principalmente a *VR* (realidad virtual), donde residía el interés de la empresa. Por ende, nuestras pruebas de concepto se aprovecharán de datos de entrada y valores obtenidos a través de sensores exclusivos de dispositivos VR como los de un *HMD (Head-motion display)*. No obstante, las bases de estas pruebas de concepto son fácilmente aplicables para ser utilizadas en cualquier otro tipo de espacio tridimensional.

Capítulo 2
Visores volumétricos

Tras un estudio de las actuales capacidades de NVC, así como el formato de visor a implementar, identificaremos una serie de alternativas o pruebas de concepto (PoC) para estudiar la sustitución de los modelos 3D por modelos neurales en la producción de contenido para realidad virtual. Nuestros esfuerzos terminaron centrándose en las tres siguientes:

- Visor de imágenes 2D estándar en un plano (adaptado a VR) **[*PoC* 2.0]**
- Visor de escenarios panorámicos en 180º/360º envolviendo al jugador **[*PoC* 2.1]**
- Visor de imágenes *object centric* 360º en VR **[*PoC* 2.2]**

Nuestra *PoC* 2.0 es más bien una simple demo o prototipo de nuestra idea inicial, donde traemos los visores de vistas 2D de objetos 3D ya existentes y los adaptamos a realidad virtual, de modo que podamos desplazar la vista siendo mostrada deslizando nuestra mano por la imagen.

Las siguientes *PoC* son las evoluciones lógicas de nuestra primera iteración. En nuestra *PoC* 2.1 utilizaremos el mismo concepto de cambio de vistas según una condición para mostrar un escenario envolvente en 180 o 360 grados, similar a un visor de captura volumétrico ya existente, pero rastreando el movimiento de la cabeza del jugador a través de la pose del *HMD* utilizado para cambiar la vista mostrada con el fin de obtener un efecto de cambio de perspectiva y profundidad. En nuestra *PoC* 2.2 mostraremos un objeto en el espacio tridimensional a través de un actor, con dicho actor cambiando la vista siendo mostrada según el ángulo desde el cual lo observemos, de forma que parezca que se comporta como cualquier otro objeto tridimensional en la escena.

Nuestras *PoC* no tienen como propósito ser una implementación final, sino valorar la viabilidad de cada propuesta en el caso de que la empresa desee centrarse en una ruta orientada al desarrollo VR en un futuro. Por ende, no estarán cuidadosamente optimizadas e implementadas en el motor. En lugar de crear nuestros propios actores optimizados para su labor utilizando C++, usaremos actores genéricos cuyo comportamiento modificaremos mediante el empleo de Blueprints (lenguaje de *scritping* visual de Unreal Engine 4), que luego nativizaremos para mejorar su rendimiento. Por otro lado, estas implementaciones sobre NVC tienen severas limitaciones: Al mostrar la imagen de una vista, no es posible aplicarles físicas complejas o interactuar con la luz. Por ende, a la hora de realizar comparativas de rendimiento, deberemos tener en cuenta sus limitaciones y su uso real, y comparar su costo respecto al de objetos 3D con las mismas limitaciones y casos de uso (usando, por ejemplo, mallas y luces estáticas, y no aplicando geometría de físicas a nuestros Actores).

2.1 PoC 2.0

PoC 2.0

2.1.1: Cargamos la matriz en un Actor compuesto por un Widget. Podremos interactuar con dicho widget en un espacio VR a través de un WidgetComponent enlazado a las manos del jugador. Esto permite que el jugador pueda "arrastrar" la imagen para cambiar la vista que está mostrando.

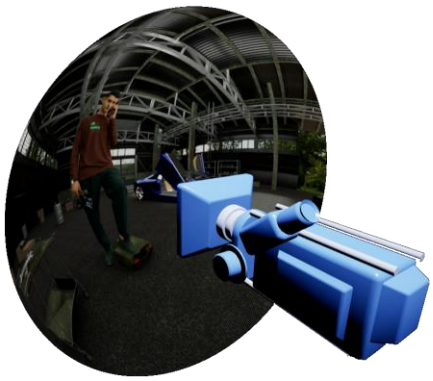
Para diseñar nuestro primer visor, utilizaremos un Widget compuesto de una imagen, a la que sustituiremos su *brush* para mostrar diferentes vistas al detectar que el jugador arrastra su mano sobre la imagen mientras presiona un botón.

Para ello, enlazaremos un WidgetInteractor a las "manos" de nuestro jugador, para poder detectar colisiones con nuestro widget. Cuando presionamos nuestro widget, almacenamos las coordenadas donde hemos presionado, y mientras deslicemos nuestra mano por el plano, calculamos la distancia recorrida desde la posición actual hasta el origen en el eje X y el eje Y. Usamos dichas distancias para sustituir el *brush* de la imagen del widget con una de las imágenes de nuestra matriz.

La sensación final es el ver una imagen desde un pequeño plano, y poder desplazar el punto de vista desde donde la estamos observando al mover nuestra mano sobre dicho plano.

2.2 PoC 2.1

PoC 2.1



2.2.1: Las fotos volumétricas en 180º o 360º son un recurso conocido y utilizado en el desarrollo de juegos y aplicaciones VR, pero tienen la desventaja de que son una captura de un punto estático, donde un cambio en la posición del usuario no cambia la perspectiva de la imagen. Aplicando NVC, podemos generar diferentes vistas y generar la ilusión de cambio de perspectiva cuando el usuario se desplaza en el mundo real.

La siguiente iteración por testear es la aplicación de la anterior PoC en visores volumétricos en realidad virtual. Los principios son los mismos, pero en lugar de mostrar la escena en un plano, utilizaremos una esfera o semiesfera con normales invertidas a las que le aplicaremos como textura la imagen que queramos mostrar, cambiando la variable de Texture2D encargada de mostrar la imagen por otra cuando un cambio de vista sea necesario.

En lugar de desplazar la escena con nuestras manos, usaremos la información obtenida de nuestro HMD. En lugar de vincular automáticamente el HDM a una cámara virtual en el espacio virtual, usaremos una cámara con una posición fija. Usando la pose del dispositivo, rotaremos dicha cámara virtual respecto a la rotación de la cabeza del espectador. Sin embargo, usaremos la posición de las gafas para calcular la distancia respecto a un origen en el eje X y Z (horizontal y altura); usaremos esta distancia para cambiar la vista mostrada por nuestra (semi)esfera para producir el efecto de profundidad y cambio en la perspectiva al desplazarnos.

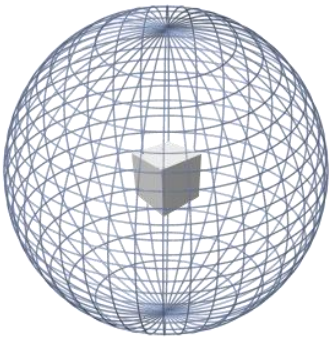
2.3 PoC 2.2

PoC 2.2

2.3.1: Un plano en una escena muestra un modelo enfocado a cámara, y cambia la vista de dicho modelo respecto a la posición y ángulo de la cámara al plano. De esta forma, podemos ver dicho modelo desde diferentes ángulos al desplazarnos por la escena, de forma que da la impresión de ser un verdadero modelo tridimensional que se encuentra en el espacio virtual.

Nuestra última prueba de concepto se enfoca en mostrar objetos en lugar de escenas. El objetivo es poder sustituir un modelo 3D detallado por un modelo NVC que muestre vistas de un objeto desde cualquier ángulo. De esta forma se pueden sustituir objetos decorativos con los que no interactuamos por imágenes con una mayor calidad gráfica con un menor coste.

Para calcular la vista a mostrar, tendremos que imaginar una suerte de esfera dividida en fragmentos. Cada uno de estos fragmentos contiene una vista del objeto capturado. Al trazar un rayo desde la cámara a nuestro objeto, la casilla atravesada será la imagen mostrada por nuestro actor (que es un simple actor vacío con un widget mostrando la imagen de la vista que deseamos).

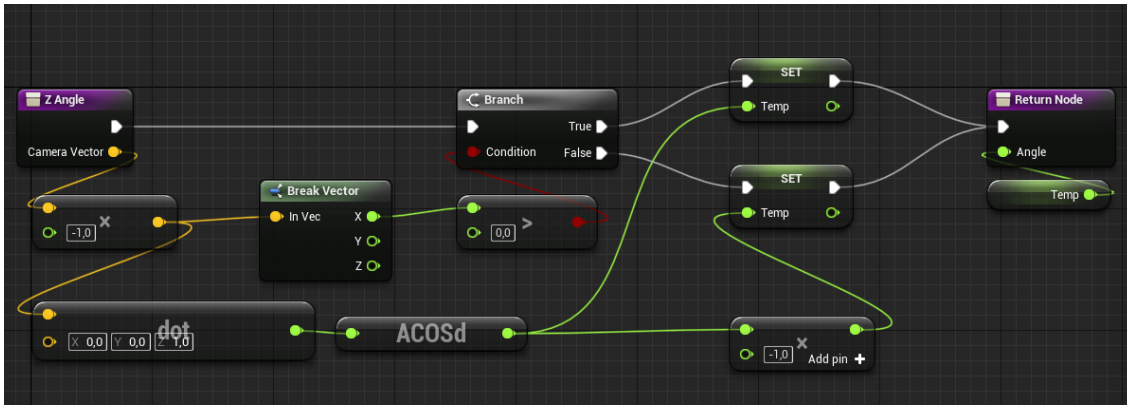


Esfera de vistas

2.3.2: Podemos imaginar que cada una de las rejillas contiene una vista de nuestro cubo. En cada *tick* de reloj, la cámara del usuario traza un rayo desde su posición hasta el centro de esta esfera imaginaria (nuestro actor). Las coordenadas de la rejilla atravesada serán equivalentes a las coordenadas de la imagen que tenemos que mostrar en nuestra matriz de vistas.

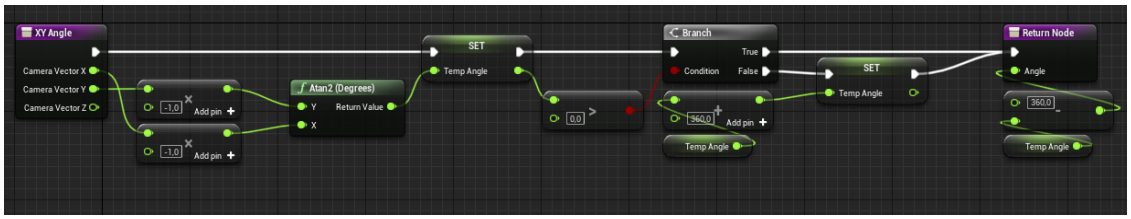
Por supuesto, no existe una esfera imaginaria que atravesar con un rayo para calcular la vista a generar. En su lugar, trazamos el vector unitario desde la cámara del jugador a nuestro objeto (nos referiremos a este como vector *vu*), y calculamos su ángulo respecto al eje Z.

Para calcular la rotación respecto al eje Z, tan solo necesitamos un valor de 0º a 180º, por lo que tan solo tenemos que calcular el producto escalar del vector Z y *vector vu* para hallar posteriormente su arcocoseno. Para saber con exactitud si estamos mirando a nuestro objeto de frente o por atrás, multiplicamos su valor por -1 cuando x sea negativa.



2.3.3: Blueprint para calcular el ángulo en Z

Para calcular la rotación respecto al eje X necesitamos un valor de 0º a 360º. No podemos utilizar el mismo método que para el eje Z, ya que el ángulo calculado tiene que ser sobre el plano XY perpendicular a Z, por lo que en su lugar calculamos su rotación sobre el eje X respecto al plano XY del mundo.



2.3.4: Blueprint para calcular el ángulo en XY

2.4 Problemas texturizando profundidad en las vistas

Una de las ideas iniciales a la hora de renderizar vistas como objetos 3D era utilizar el mapa de profundidad generado por el modelo NVC como textura normal en el material encargado de mostrar el objeto/escena para producir un mejor efecto de volumen. Los mapas son generados identificando el píxel más cercano a donde se toma la foto y coloreándolo en negro, y el píxel más lejano en blanco. Posteriormente, el resto de las distancias en cada píxel se mapea en una escala de blanco a negro. Lamentablemente, esta opción no pudo ser implementada debido a los siguientes problemas:

- Algunos elementos en las vistas no se mapean correctamente en lo que respecta su profundidad. Elementos reflectivos o translúcidos son especialmente problemáticos, ya que se identifican como elementos muy lejanos.
- Los mapas de profundidad de cada imagen se generan individualmente para cada vista, y no están normalizados los unos con los otros, dando diferentes escalas de profundidad en cada vista. Esto se debe a que no tenemos una escala de profundidad real, y que generamos estos mapas al partir del rango del píxel más cercano y lejano de la posición donde se toma la vista, en lugar de generarlo en base a una escala común.

Necesitaríamos de una herramienta más especializada a la hora de mapear la profundidad si quisiésemos continuar con esta aproximación.

Profundidades erróneas

2.4.1 (Izquierda): El mapa de profundidad extraído no corresponde con la realidad en el agua o las ventanas.

2.4.2 (Derecha): Los mapas de profundidad generados en vistas muy cercanas no casan entre sí, mostrando valores de profundidad muy diferentes.

Capítulo 3
Estudio de rendimiento

Nuestro objetivo es el estudio de rendimiento de los tres modelos explicados anteriormente frente a modelos 3D que podrían sustituir. Los experimentos han sido planteados y desarrollados para realidad virtual, utilizando como hardware el dispositivo Oculus Quest 2. Todas las pruebas de rendimiento han sido desarrolladas en UE4.

3.1 Limitaciones y planificación

Limitaciones del modelo

Al trabajar con nuestros modelos NVC, debemos de tener en cuentas varias cosas:

- No tienen propiedades físicas.
- No pueden reaccionar a iluminación.
- Materiales estáticos.

Es por tanto un factor relevante que, a la hora de comparar el rendimiento entre ambos tipos de objetos, debemos optimizar los objetos 3D omitiendo la necesidad de reaccionar a la luz (usando *lightmappings* de luz estática) o simular físicas (el coste de rendimiento debe venir del render de polígonos, no del cálculo de físicas).

Por otra parte, nuestros modelos NVC también estarán afectados por su implementación actual, ya que se usa un widget o una *mesh* al que superponer imágenes o texturas para producir el efecto de cambio de perspectiva. Esto significa que posiblemente el coste de CPU al cambiar una vista sea mayor del necesario. Por otro lado, en una hipotética implementación del Modelo 1, la generación de vistas podría aumentar el uso de GPU. Ambos factores no se tendrán en cuenta durante el *profiling*, pero deberían ser considerados en el futuro.

Limitaciones del hardware

Trabajaremos utilizando el dispositivo Oculus Quest 2 como hardware donde ejecutar nuestras demos. Estas gafas de realidad virtual no son excesivamente potentes, pero son capaces de renderizar juegos a 72 fotogramas por segundo (*fps* – *frames per second*). Para ello, requeriremos configurar los ajustes de nuestro proyecto para poder renderizar escenas en 3D con el menor coste posible y mantener los dichos 72 fps. Toda esta optimización se unirá a la optimización realizada ya anteriormente en los objetos 3D, por lo que requeriremos de objetos relativamente complejos para poder realizar comparativas donde podamos notar un impacto en el rendimiento.

3.2 Profiling y comparación de datos

Utilizaremos el *software* oficial de Oculus (OVR Metric Tools) para desarrollar el *profiling* de nuestras pruebas. Para instalarlo, deberemos descargárnoslo desde la página web de Oculus e instalarlo manualmente a través del PC a nuestras gafas de realidad virtual utilizando el comando 'adb'. Podemos encontrar el enlace a este recurso en la bibliografía de esta memoria.

OVR Metric Tools nos permite visualizar en tiempo real el consumo de CPU, GPU, la cantidad de fps y otro tipo de información mientras depuramos o testeamos nuestro videojuego. Además, podemos almacenar los datos recopilados de una sesión de testeo en un archivo .csv para poder analizarlos y compararlos como consideremos necesario.

Aunque existen una gran cantidad de datos que podemos tener en cuenta durante el profiling, nos centraremos en estos seis valores*:

FPS	**Media de FPS.**	Muestra la ratio de fotogramas en fotogramas por segundo.
APP T	**Media del tiempo de uso de la GPU de la aplicación.**	Muestra el tiempo de la GPU de la aplicación (en µs, 1/1000 de ms), que es la cantidad de tiempo que la aplicación emplea en renderizar un solo fotograma.
STALE	Media de fotogramas anticuados.	Muestra el número de fotogramas antiguos. Indica el número de veces que un fotograma no se entregó a tiempo y se utilizó el fotograma anterior en su lugar.
CPU U	Uso de CPU.	Muestra el porcentaje de utilización de la CPU. Sin embargo, esta métrica representa el núcleo de peor rendimiento y, dado que la mayoría de las aplicaciones son multihilo y el programador asigna los hilos a medida que están disponibles, el hilo principal de la aplicación puede no estar representado en esta métrica.
GPU U	Uso de GPU.	Muestra el porcentaje de utilización de la GPU. Si este valor está al 100%, el cuello de botella se encuentra en la GPU. Los problemas de rendimiento pueden comenzar a aparecer sobre el 90%.
U MEM	Memoria disponible (Reservada por la ap.).	Indica la memoria disponible en MB según lo informado por Android. En Android, la memoria se maneja de una manera un tanto opaca de cara al desarrollador, lo que hace que este valor sea útil sólo como orientación general. Este valor es principalmente útil para controlar si la memoria se asigna más rápido de lo esperado, o si la memoria no se libera cuando se espera que lo haga.

(*A la hora de presentar nuestros datos, mostraremos la moda obtenida de estos valores durante la ejecución del juego/programa. Si estos valores no son fieles a la realidad debido a no ser constantes, contendrán una (i) a su lado, indicando que son inestables.)

De estos datos, los valores más importantes a considerar serán la media de fotogramas por segundo y el tiempo de GPU de la aplicación. Para poder renderizar 72 fotogramas por segundo, tendremos que renderizar cada fotograma de nuestra aplicación en un lapso de 13.8 milisegundos (o 13800 µs). Si nuestra aplicación se ejecuta a menos de 72 fps y el tiempo de GPU de la aplicación es mayor a 13800 µs, estaremos sufriendo un cuello de botella en la GPU, pero si el tiempo de GPU de la aplicación es menor, estaremos lidiando con un cuello de botella en la CPU.

Simplificando:

- Si “FPS < 72” y “APP T > 13800”, sobrecarga de GPU.
- Si “FPS < 72” y “APP T < 13800”, sobrecarga de CPU.

Generalmente, comenzaremos a notar algunos problemas de rendimiento cuando nuestra GPU U esté al 90%, que se agravarán si alcanza el 100%. El CPU U es más engañoso, ya que tan solo nos muestra el uso del núcleo con peor rendimiento, por lo que es mejor fijarse en el número de fotogramas y el tiempo de aplicación de la forma mencionada en el párrafo anterior.

La memoria disponible no suele ser un dato excesivamente fiable en Android debido a como el sistema operativo la gestiona de cara al desarrollador, pero puede ser un dato útil para comprobar el impacto en memoria de un modelo 3D frente a una matriz de vistas.

3.3 PoC 2.0

Situamos nuestra PoC 2.0 en un espacio tridimensional, e interactuaremos con la misma a través de unas manos modeladas en 3D que nos permitirán interactuar con el widget de la forma que se ha explicado anteriormente.

TEST	FPS	APP T	STALE	CPU U	GPU U	U MEM
PoC 2.0	72	3970	0	42	45	520

Nótese que parte del coste de GPU, CPU y memoria vienen de renderizar el espacio tridimensional donde se sitúan, así como de los controladores que utilizamos para interactuar con este (cuyo coste es bastante alto). Podemos dividir los costes en una escena en la que tan solo existe nuestro Actor, y otra en la que existen todos los elementos de la escena original salvo nuestra PoC 2.0.

TEST	FPS	APP T	STALE	CPU U	GPU U	U MEM
PoC 2.0 (A)	72	1200	0	35	20	470
Escena 3D	72	3705	0	35	39	390

La conclusión que obtenemos de estos datos es que nuestro Actor NVC tiene un coste relativamente bajo respecto a la escena final, pero genera un mayor uso de memoria respecto a usar una *mesh*. Los usos de este actor en un espacio tridimensional como sustituto de un modelo 3D son muy específicos, por lo que no realizaremos una comparativa con ninguna escena similar para comparar rendimiento. Delegaremos esa labor en las PoC 2.1 y 2.2.

3.4 PoC 2.1

Una de las mayores ventajas de esta prueba de concepto es que no tenemos que preocuparnos por renderizar el resto del mundo tridimensional (ni su iluminación, post procesamiento, etc), la propia imagen renderiza toda la escena. Además, mientras mantengamos la misma resolución de imagen, los costes de la prueba de concepto serán muy similares independientemente de la escena que mostremos.

A cambio, tendremos que generar una vista de una imagen panorámica en lugar de una vista 2D, lo que dificulta el proceso. Además, en esta prueba de concepto, necesitamos cargar la vista como una textura que añadir a el material de nuestro visor (una esfera con las normales invertidas).

Escena 3D frente a PoC 2.1

3.4.1: Nuestra PoC 2.1 no solo tiene ventajas de rendimiento: sabemos que, al compilar, no obtendremos texturas diferentes *in-engine* frente al software final.

Aunque también cuenta con defectos apreciables, como ángulos borrosos en el suelo y el cielo (un problema típico de las imágenes panorámicas) así como todas las limitaciones de ser una imagen frente a un objeto 3D.

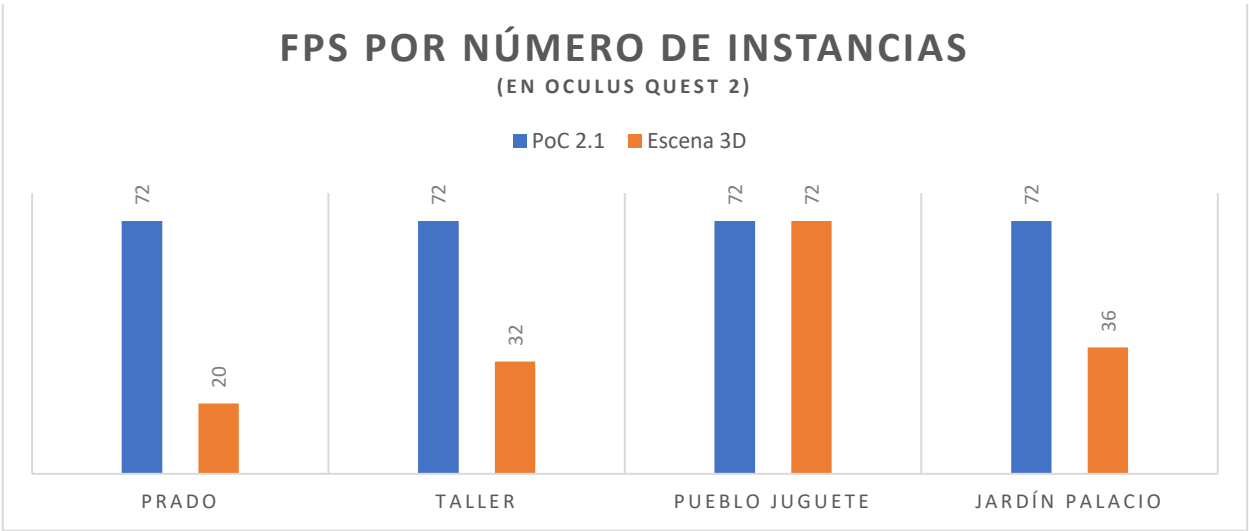
Los resultados obtenidos al testear dierentes modelos son prometedores. Si nuestro objetivo es mostrar una escena tridimensional rica en detalles, nuetra PoC cumple con creeces con un menor uso de CPU y GPU que utilizando modelos tridimensionales tradicionales, incluso tras ser cuidadosamente optimizadas. Además, la configuración de nuestra PoC requiere de mucho menos tiempo de optimización y sus resultados son mucho más predecibles, ya que siempre actuará igual.

TEST	FPS	APP T	STALE	CPU U	GPU U	U MEM
PoC 2.1	72	3800	0	30	39	1390
Prado 3D	20	45200 (*i*)	72	72 (*i*)	91 (*i*)	460
Taller 3D	32	26050 (*i*)	72	49	99	510
Pueblo de juguete 3D	72	4970	0	36	48	480
Jardín de palacio 3D	36	24160 (*i*)	72	33	99	589

Diferentes escenas de prueba utilizadas

3.4.2: Las escenas 3D con materiales simples y pocos polígonos no suponen un problema, pero a mayor complejidad, mayores problemas de rendimiento.

Incluso en escenas simples y pulidas (como el pueblo de juguete), podemos ver como renderizar la escena en nuestra prueba de concepto implica un menor gasto de recursos. En escenas más complicadas y con mayor cantidad de detalles y formas y texturas más complejas, los resultados son mucho más diferenciados. Además, usar imágenes nos permite lograr una mayor calidad y realismo en las sombras, reflejos, etc. Sin embargo, no todo son ventajas: algunos ángulos de visión situados abajo y arriba de nuestra cámara son borrosos; nuestras imágenes no pueden reaccionar a cambios de luz dinámica ni simular físicas, y los objetos 3D estáticos pueden simular movimiento a través de texturas y *shading* (simular el movimiento del césped al moverse por el viento, las olas del mar al arremeter contra la orilla, etc.).

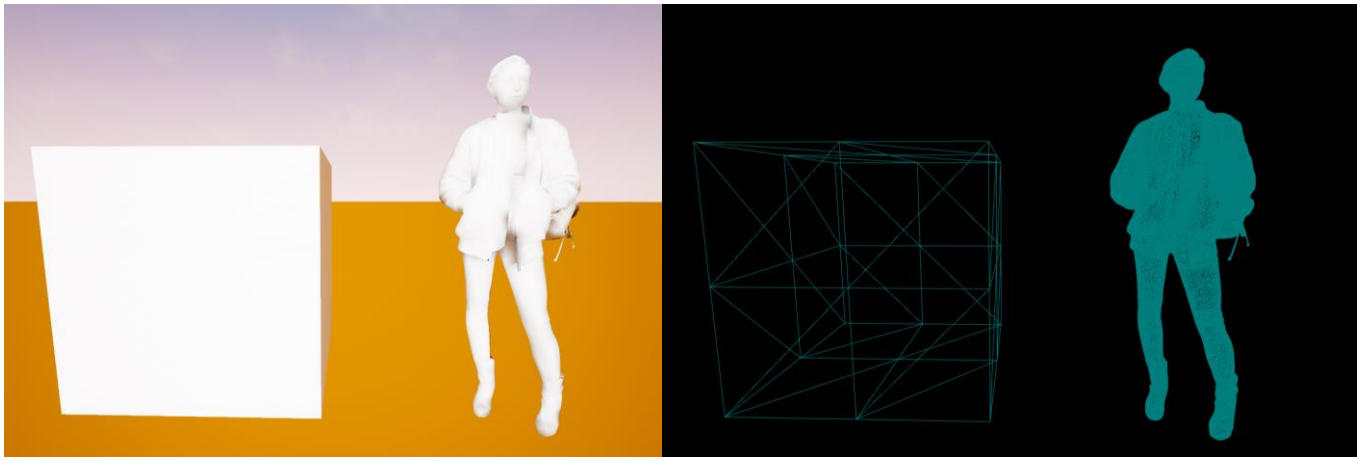


3.5 PoC 2.2

Nuestra última PoC propone simular objetos en 3 dimensiones utilizando un modelo entrenado para generar vistas de un objeto. Cuando realizamos comparativas entre nuestros modelos frente a objetos 3D sencillos o modelos 3D con mayor carga de polígonos. Esta PoC tendrá resultados más similares a las PoC 2.0 que a las de las 2.1, ya que también tenemos que preocuparnos de renderizar el resto de mundo 3D, iluminación (estática), etc. No obstante, no inicializaremos los actores WidgetInteractor, cuyo consumo es considerable, por lo que los resultados estarán menos alterados.

TEST	FPS	APP T	STALE	CPU U	GPU U	U MEM
PoC 2.2	72	3500	0	36	40	509
PoC 2.2 sin mundo 3D	72	1700	0	33	22	458
Simple3D	72	4800	0	30	47	395
Complex3D	72	7300	0	30	66	414

Diferentes modelos 3D para comparar



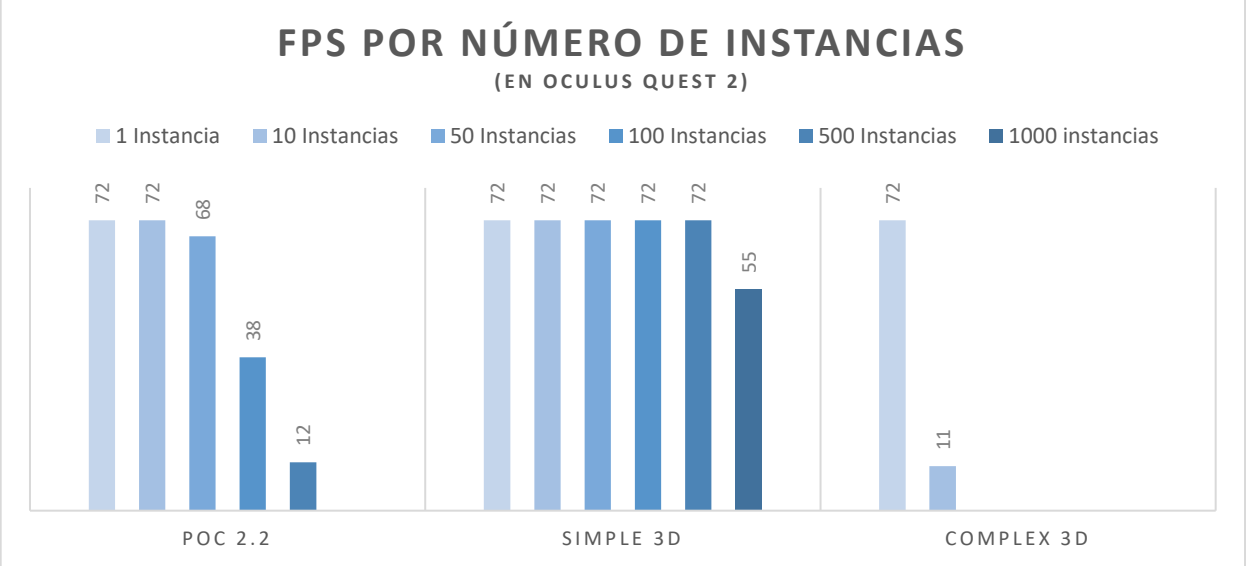
3.5.1: Compararemos el rendimiento de nuestra PoC con un modelo de un cubo (48 polígonos) y un modelo más complejo, el cuerpo de una mujer (≈100k polígonos).

A priori, los resultados parecen beneficiosos para nuestra PoC, pero en cualquier escena real no nos limitaremos a usar un único actor, en su lugar, crearemos decenas (centenares, en muchas ocasiones) de objetos. Nuestra PoC tiene que calcular una vista para cada objeto NVC en cada fotograma. Comprobemos como se comportan estos modelos al instanciar un gran número de estos.

PoC 2.2	FPS	APP T	STALE	CPU U	GPU U	U MEM
1 instancia	72	3500	0	36	40	509
10 instancias	72	4000	0	47	41	514
50 instancias	68	6200	5	80	51	568
100 instancias	38	8500	55	90	45	578
500 instancias	12	25800	72	100	37	855
Simple3D	**FPS**	**APP T**	**STALE**	**CPU U**	**GPU U**	**U MEM**
1 instancia	72	4800	0	30	47	395
10 instancias	72	5750	0	33	60	421
50 instancias	72	5800	0	35	62	426
100 instancias	72	7181	0	35	71	445
500 instancias	72	8900	0	47	83	433
1000 instancias	55	15000	52	52	99	446
10000 instancias	10	65535	72	75	99	585
Complex3D	**FPS**	**APP T**	**STALE**	**CPU U**	**GPU U**	**U MEM**
1 instancia	72	7300	0	30	66	414
2 instancias	72	11800	0	32	95	414
4 instancias	70	12300	6	29	99	429
6 instancias	50	16800	65	36	99	407
10 instancias	11	56000	72	10	99	430

A la hora de instanciar objetos con un gran número de polígonos, nuestra PoC ofrece mayor cantidad de instancias en la escena antes de causar un impacto en el rendimiento que un modelo 3D tradicional. Nuestras PoC podrán soportar hasta 50 actores antes de comenzar a sobrecargar la CPU de las Oculus Quest, mientras que el modelo complejo causará un cuello de botella en la GPU al partir de las 4 instancias.

A la hora de instanciar una gran cantidad de objetos simples, está claro que el modelado 3D tradicional se sigue imponiendo: Podemos llegar a instanciar hasta 500 objetos sin llegar a sufrir un impacto en el rendimiento.



3.6 Consideraciones a futuro

Si bien la metodología actual implica pre-generar las vistas fuera del motor y exportar estas vistas a UE4 (Modelo 3), la meta final consistiría en la implementación del Modelo 1, donde las vistas se generarían en cada fotograma del proceso. Esto implicaría un futuro sistema donde el gasto de GPU***[1]** y el APP T es mucho mayor, pero el gasto de CPU es menor, así como la memoria utilizada por la aplicación.

La viabilidad de la implementación del Modelo 1 dependerá de la capacidad de NVC de generar vistas con un menor coste de CPU y un menor tiempo de ejecución que el de un modelo 3D, lo cual, dado el actual estado del arte, parece una visión muy optimista: se tarda más de 500 milisegundos en generar una vista de un modelo entrenado en equipos con GPUs de alta gama: el consumo de GPU de un modelo para generar una vista es demasiado alto y su tiempo de respuesta demasiado lento.

No obstante, existen diferentes ramas de estudio que buscan aumentar la velocidad de generación de vistas de NVC (de 30 a 120 fps) y reducir su coste computacional, así como solucionar algunos de sus problemas, como otorgar a los modelos la capacidad de generar vistas con una entrada de iluminación diferente al de las capturas con el que el modelo ha sido entrenado, o entrenar modelos que recuerden el movimiento de una escena entrenada. La usabilidad del Modelo 1 dependerá en gran medida del éxito o fracaso de estas tecnologías.

En general, podemos concluir que, pese a que los resultados obtenidos son relativamente buenos, nos encontramos frente a una propuesta en una fase de desarrollo muy temprana. Es difícil de predecir si una futura implementación en motores gráficos para el desarrollo de videojuegos sería una ruta viable o útil, ya que requiere una gran mejora en el tiempo de renderizado y reducir costes de GPU a la hora de generar una vista sin aumentar demasiado el coste de CPU; enfrentándose a unos sistemas de render de modelos 3D que hoy en día son muy eficientes.

***[1] El gasto de GPU y CPU se reducirían ligeramente al eliminar la necesidad de renderizar en un widget/plano/malla/etc. y renderizar en el pipeline gráfico, pero es difícil que coste de generar una vista a través del modelo NVC sea menor que este ahorro**

Otra opción que explorar podría ser el mantener nuestro modelo actual de pre-generar la(s) vista(s) fuera del motor, e importar aquellas que deseemos en este. Esto limita su uso en aplicaciones como videojuegos, así como la integración de la tecnología en el motor, pero puede tener un uso puntual en la creación de fondos falsos para cinemáticas, platós virtuales, etc.

Un factor de peso a tener en cuenta es la reciente publicación de la versión de acceso anticipado de Unreal Engine 5 (UE5). Este nuevo motor gráfico incluye Nanite, una forma de virtualización de geometría que permite renderizar miles de instancias de mallas estáticas con una gran cantidad de polígonos sin impacto en el tiempo de ejecución. Nanite todavía no está disponible para aplicaciones de realidad virtual, por lo que no se realizarán comparativas frente a esta nueva tecnología. No obstante, Epic Games ha comunicado su intención de que Nanite esté disponible para estos dispositivos en la versión final de UE5. Por ello, es necesario que nuestros modelos NVC planteen mejoras de rendimiento u ofrezcan algunos usos que las mallas estáticas Nanite no nos puedan ofrecer.

Nanite

3.6.1 Es difícil imaginar a nuestros modelos NVC siendo una alternativa frente a una malla Nanite en lo que respecta a potencia y facilidad de uso, cuando estas pueden interactuar con el sistema de físicas e iluminación del motor gráfico, e instanciarse en gran cantidad sin importar el número de polígonos en escena.

(Imagen extraída de la documentación de Unreal Engine)

Un modelo NVC no tiene que lidiar con límites de polígonos, almacenar ni mapear texturas complejas, etc; pero a efectos prácticos, es difícil de imaginar mallas Nanite que presenten un problema en el entorno de desarrollo de un videojuego, y ofrece una gran ventaja que nuestros modelos NVC carecen: la interacción con el resto de los elementos del motor (luz, físicas…). Por si fuera poco, Nanite tiene muy pocos problemas a la hora de renderizar una gran cantidad de actores diferentes, mientras que el rendimiento de nuestros modelos NVC escalan bastante mal cuanto mayor sea su cantidad, ya que cada modelo debe calcular individualmente el ángulo en el que están siendo observados, generar una vista, y mostrarla en la cámara del usuario. En otras palabras, es difícil imaginar nuestra PoC 2.2 ofreciendo una solución competitiva frente a Nanite hoy en día.

Un desafío considerable podría ser la integración de nuestros modelos NVC en Unreal Engine (con el fin de poder interactuar con el sistema de iluminación, físicas, etc), o la creación de un motor gráfico compatible con estos modelos. Este es un desafío considerable, y su viabilidad dependerá en gran medida de los avances en rendimiento de los modelos NVC, así como la identificación de casos de usos particulares donde estos modelos puedan solucionar problemas que no puedan resolverse a través de una malla 3D Nanite o una captura volumétrica tradicional.

Por ejemplo:

- Nanite no es perfecto: a mayor cantidad de polígonos, complejidad texturas, etc. de una *mesh,* mayor será su peso. Un modelo 3D tradicional ocupa mucho más espacio que un modelo NVC entrenado para generar vistas. A la hora de reducir espacio, un modelo NVC podría ser mucho más deseable que una malla. Esto es un factor más relevante de lo que puede parecer a primera vista, ya que los juegos se vuelven cada vez más y más pesados y el almacenamiento comienza a ser un problema. Al contrario que un modelo 3D, nuestro modelo NVC consistiría en una única red neuronal entrenada, ocupando mucho menos espacio y sin necesidad de almacenar texturas pesadas ni mallas complejas. Además, un modelo NVC tiene una capacidad de deformación potencialmente mayor que el de una malla estática.

- Las capturas volumétricas tradicionales - mediante escaneo 3D (por ejemplo, mediante fotogrametría) - requieren de una gran cantidad de fotos para generar un modelo estático 3D, muy pesado y con una gran cantidad de polígonos, que muchas veces hay que tratar para eliminar artefactos o geometría no deseada. Además, requiere de equipo con un coste económico elevado y largos periodos de procesamiento para obtener modelos de alta calidad. Estos modelos son estáticos y conservan la iluminación de la escena donde han sido tomadas, salvo que sean modificadas manualmente tras ser convertidas en una mesh. Poder reducir costes a la hora de capturar modelos fotorrealistas podría ser un buen incentivo para usar modelos NVC pese a sus limitaciones actuales.

Es complicado predecir el porcentaje de éxito o fracaso de dichas ramas de investigación u otras totalmente diferente. La tecnología y sistemas tras NVC aún está en constante desarrollo, y su calidad y coste, así como su flexibilidad y posibilidad de integración con otros sistemas aún tiene mucho margen de mejora. Cualquiera de los casos de uso mencionados en esta memoria podría cambiar radicalmente en cualquier momento con algún nuevo avance o descubrimiento a la hora de entrenar o interpretar la red.

Capítulo 4
Conclusiones y líneas futuras

Como se ha mencionado previamente, la utilidad de un modelo NVC dependerá, en gran medida, de la rapidez en la que sea capaz de generar una vista dados unos parámetros de entrada, así como su coste en la GPU.

Es difícil imaginar a la actual iteración de NVC enfrentarse a nuevos modelos de mallas estáticas como pueden ser Nanite, así como a sistemas de captura volumétrica tradicional. Pero existen diferentes rutas de investigación donde sería posible que estos modelos pudiesen suplir carencias que la tecnología actual no puede solucionar.

Los resultados obtenidos por nuestras pruebas de concepto son prometedores dadas las circunstancias en las que han sido realizadas, pero dependen en gran medida del avance del estado del arte en esta tecnología.

La viabilidad de los modelos NVC en computación gráfica dependerán, en última medida, de su evolución en los próximos años, así como en solucionar algunas de sus limitaciones, como pueden ser la iluminación en tiempo real.

Capítulo 5
Summary and Conclusions

As previously mentioned, the usefulness of a NVC model will depend, to a large extent, on how fast it is able to generate a view given some input parameters, as well as its cost on the GPU.

It is difficult to imagine the current NVC iterations facing new static mesh models such as Nanite, as well as traditional volumetric capture systems. But there are different research paths where it would be possible that these models could fill gaps that current technology cannot address.

The results obtained by our proof-of-concept tests are promising given the circumstances in which they have been performed but their future is highly dependent on the state of the art in this new technology.

The viability of NVC models in computer graphics will ultimately depend on their evolution in the coming years, as well as on solving some of their limitations, such as real-time lighting.

Bibliografía

- Oculus Quest Documentation:
  https://developer.oculus.com/documentation/?device=QUEST

- Oculus Quest for Unreal Engine:
  https://developer.oculus.com/documentation/unreal/?device=QUEST

- OVR Metric Tools:
  https://developer.oculus.com/documentation/tools/tools-ovrmetricstool/

- Unreal Engine:
  https://www.unrealengine.com

- Documentación de Unreal Engine 4:
  https://docs.unrealengine.com/en-US/index.html

- Nanite (Unreal Engine 5):
  https://docs.unrealengine.com/5.0/en-US/RenderingFeatures/Nanite/

- PyTorch:
  https://pytorch.org/

- Documentación de PyTorch:
  https://pytorch.org/docs/stable/index.html