Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Uso de técnicas de Inteligencia Artificial para la generación automática de contenidos en videojuegos

Use of Artificial Intelligence techniques for the automatic generation of content in video games

Vanessa Valentina Villalba Pérez

La Laguna, 13 de septiembre de 2022

D. **Rafael Arnay del Arco**, con N.I.F. 78569591-G profesor contratado Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **José Demetrio Piñeiro Vega**, con N.I.F. 43774048-B profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

C E R T I F I C A (N)

Que la presente memoria titulada:

”*Uso de técnicas de Inteligencia Artificial para la generación automática de contenidos en videojuegos*”

ha sido realizada bajo su dirección por Doña **Vanessa Valentina Villalba Pérez**, con N.I.F. Y5866526-E.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 13 de septiembre de 2022

Agradecimientos

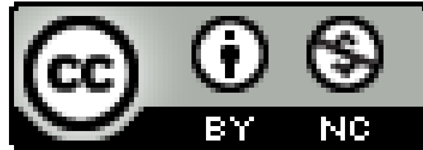
A mi familia, porque me han apoyado desde siempre y desde donde fuera, incluso cuando estaba indecisa de si era el momento de empezar una carrera estando lejos de casa, me impulsaron a seguir mis metas en todo momento.

A mis amigos y compañeros durante la carrera, de los que me llevo lo importante de trabajar en equipo. En especial a Marta y a Óscar, porque entre los tres hacíamos que hasta los días más duros fueran más llevaderos, confiando en nosotros y sobre todo acompañándonos durante largas horas de trabajo, han sido un pilar fundamental de mi paso por la carrera y espero que lo sigan siendo por mucho más, gracias.

A mis tutores, Arnay y Demetrio, por entusiasmarse tanto como yo con este proyecto desde el día uno, por haberme guiado y resuelto todas las dudas que tenía y finalmente por su paciencia.

Muchas gracias a todos.

Licencia



© Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial 4.0 Internacional.

Resumen

Este Trabajo de Fin de Grado, nace del interés de encontrar, estudiar y, utilizar un modelo de Inteligencia Artificial que sea capaz de generar contenidos en un escenario virtual de forma automática, debido a los beneficios que esto trae consigo. Para ello, se ha elegido la técnica que se propone en el paper de NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis publicado en el año 2020 [19], que permite la generación de vistas novedosas a partir de una red neuronal entrenada con imágenes de una escena e información referente a las poses de la cámara, logrando simular un modelo 3D dentro de una imagen. Y, una vez realizadas diferentes pruebas con NeRF, se propone su implementación dentro de un mundo virtual con el uso del motor gráfico Unity3D, ya que como línea futura se espera lograr una posible adaptación para videojuegos.

Palabras clave: videojuegos, Inteligencia Artificial, gráficos en tiempo real, técnicas de renderizado, motores gráficos

Abstract

This Final Degree Project is born from the interest in finding, studying, and using an Artificial Intelligence model capable of automatically generating content in a virtual environment, due to the benefits that this brings. To do this, the technique proposed in the NeRF paper: Representing Scenes as Neural Radiance Fields for View Synthesis published in 2020 [19] has been chosen, which allows the generation of novel views from a neural network trained with images of a scene and the camera poses, managing to simulate a 3D model within an image. And, once different tests have been carried out with NeRF, its implementation within a virtual world with the Unity3D graphics engine is proposed since it is expected to achieve a possible adaptation for video games as a future line.

Keywords: video games development, Artificial Intelligence, real-time graphics, rendering techniques

Índice general

1. **Introducción** 1
   - 1.1. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis . 1
   - 1.2. Gráficos en tiempo real . . . 2
     - 1.2.1. Ray Tracing . . . 2
     - 1.2.2. Volume Ray Casting . . . 3
   - 1.3. Objetivos . . . 4

2. **Trabajos relacionados** 5
   - 2.1. GANCraft: Unsupervised 3D Neural Rendering of Minecraft Worlds . . . 5
   - 2.2. GPT-3 . . . 6
   - 2.3. NeRF in the wild: Neural Radiance Fields for Unconstrained Photo Collections 6
   - 2.4. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding . 7
   - 2.5. pixelNeRF: Neural Radiance Fields from One or Few Images . . . 7
   - 2.6. Nerfies: Deformable Neural Radiance Fields . . . 8

3. **Material y métodos** 9
   - 3.1. Motivación . . . 9
   - 3.2. Funcionamiento de NeRF . . . 9
     - 3.2.1. Arquitectura del MLP . . . 10
   - 3.3. Tiny NeRf . . . 10
     - 3.3.1. Arquitectura del MLP . . . 11
     - 3.3.2. Tecnologías utilizadas . . . 11
     - 3.3.3. Datos de entrada y resultados . . . 11
   - 3.4. Comunicación entre Tiny NeRF y Unity3D . . . 13
     - 3.4.1. Servidor . . . 13
     - 3.4.2. Cliente . . . 14
   - 3.5. Escenario en Unity3D . . . 16
     - 3.5.1. Assets utilizados y diseño de la escena . . . 16
     - 3.5.2. Manejo de controles para el jugador y Tiny NeRF . . . 17
   - 3.6. Generación de dataset propio . . . 20

4. **Resultados obtenidos** 21
   - 4.1. Especificaciones . . . 21
   - 4.2. Optimización de Tiny NeRF . . . 21
     - 4.2.1. Prueba con NeRF . . . 22
   - 4.3. Escenario en Unity3D . . . 23

5. **Conclusiones y líneas futuras** 26

6. **Summary and Conclusions** 27

7. **Presupuesto** 28

A. **Repositorios y datasets disponibles** 29

A.1. Datasets naturales y sintéticos 29
A.2. Repositorio de Unity3D-Python-Communication 29
A.3. Repositorio asociado a este TFG 29
A.4. Repositorio de Local Light Field Fusion 29
A.5. NeRF para datasets propios. Resultados en vídeo 29

Índice de Figuras

1.1. Resumen de NeRF [19]. 2
1.2. Nociones básicas del Ray Tracing [10] 3
1.3. Etapas del volume ray casting [11] 4

2.1. Resultados de GANCraft [16] 5
2.2. Entrada y resultados para NeRF in the Wild [17] 6
2.3. Referencia y resultados para Instant NeRF con diferentes tamaños de tablas Hash [20] 7
2.4. Entrada y resultados para pixelNeRF [24] 8
2.5. Entrada y nerfie resultante [22] 8

3.1. NeRF pipeline [19] 10
3.2. Arquitectura del MLP 11
3.3. Escena del garage 17
3.4. Características de los datasets. 20

4.1. PSNR para diferentes Tiny NeRF. 21
4.2. PSNR para diferentes Tiny NeRF. 22
4.3. Resultados de NeRF durante el entrenamiento. 23
4.4. Tiny NeRF en Unity3D. 24
4.5. Tiny NeRF en Unity3D. 25

Índice de Tablas

7.1. Presupuesto del proyecto 28

Listados

3.1. Metadatos de la cámara. 12
3.2. Lógica del servidor. 14
3.3. Lógica del cliente. 15
3.4. Lógica para el movimiento de la cámara. 18
3.5. Thresholding a Tiny NeRF. 19

Capítulo 1

Introducción

Debido al avance en motores gráficos [21] y las diferentes técnicas de renderizado aplicadas dentro de los mismos, los resultados visuales al implementar un escenario dentro de un videojuego son impresionantes ¿Y si se le añade el uso de técnicas de Inteligencia Artificial? La experiencia del jugador mejora, se vuelve exquisita, aportando una sensación de realidad que le permite sumergirse en el mundo virtual. En la actualidad, existen diferentes técnicas que permiten, por ejemplo, modelar y simular el comportamiento de un personaje, obtener finales alternativos dependiendo de las decisiones tomadas a lo largo de la partida, trazar trayectorias óptimas de un punto a otro, entre otros [5].

Es por ello, que nace la motivación inicial de este Trabajo de Fin de Grado, la cual es encontrar, estudiar y utilizar un modelo de Inteligencia Artificial existente que sea capaz de generar contenidos en un escenario virtual de forma automática, por lo que se ha propuesto el uso del modelo de redes neuronales *NeRF* [19] para lograrlo. Asimismo, esta memoria está dividida en diferentes capítulos; el primer capítulo, es una breve introducción al problema así como conceptos básicos relacionados a los gráficos en tiempo real; el segundo capítulo, plantea los distintos trabajos relacionados a NeRF, que también fueron tomados en consideración al principio del desarrollo; el tercer capítulo, desglosa todo el desarrollo e implementación del proyecto en sí, explicación en detalle de la estructura del modelo de IA escogido, diferentes datasets utilizados, entrenamientos obtenidos, dificultades encontradas, puesta en escena con *Unity3D*[1], entre otros detalles; el cuarto capítulo, explica los resultados obtenidos durante la fase de experimentación; el quinto y sexto capítulo, son las conclusiones y líneas futuras del proyecto, en español e inglés respectivamente y, finalmente, el séptimo capítulo, el cual distingue el presupuesto necesario para llevar a cabo el proyecto.

1.1. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis

Los gráficos en tiempo real se han visto agitados con cambios que provienen de la evolución de la IA. Entre estos cambios destaca el modelo propuesto *NeRF*, cuyo algoritmo es capaz de, partiendo de un conjunto disperso de imágenes de una escena compleja tomadas desde diferentes perspectivas, codificar dicha escena y aprender a interpolar vistas desconocidas, teniendo que aprender así conceptos como iluminación, paralaje, oclusión, entre otros. Ver Figura 1.1.

[1] Unity3D: `https://unity.com/es`

Este modelo fue publicado por primera vez en el paper titulado *NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis* [19] para el año 2020 y actualmente se encuentra en auge debido a su planteamiento y variaciones del proyecto inicial existentes, por lo que se ha considerado como modelo base para el desarrollo de este proyecto. El uso de *NeRF* a lo largo de este proyecto se basa en sus resultados principalmente, ya que no solo permite obtener vistas desconocidas de un objeto, lo cual lo hace excepcional, si no que también permite plasmar un modelado 3D de dicho objeto en imágenes y se espera utilizar dichas imágenes dentro de un ambiente virtual, para poder comprobar los posibles beneficios que podría tener el uso de este modelo, por ejemplo, para un videojuego.

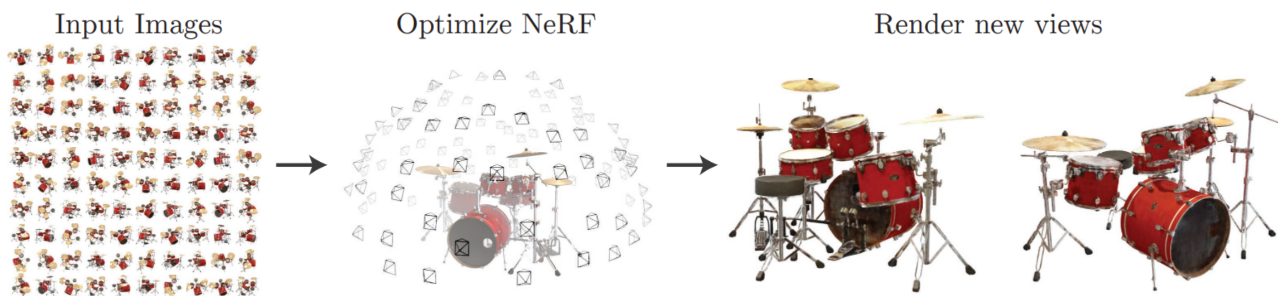


Figura 1.1: Resumen de NeRF [19].

1.2. Gráficos en tiempo real

En cuanto a generación de gráficos en tiempo real se refiere, existen diferentes técnicas de renderizado [1] que juegan un papel importante y, que además, son fundamentales para el entendimiento de fases que se llevan a cabo en la arquitectura de *NeRF*, por lo que serán resumidas a continuación.

1.2.1. Ray Tracing

También conocido como trazado de rayos en español o trazado de rayos de luz, tiene como objetivo simular la iluminación global, sombras y reflejos sobre las superficies, tal y como lo hace la vista humana pero dentro de un ambiente virtual, aplicando este efecto en proyectos cinematográficos, imágenes o videojuegos.

Su funcionamiento viene dado a partir de un algoritmo que genera rayos desde el punto de vista, es decir, la cámara, atravesando el plano en 2D de la escena en 3D, donde logra interceptar el objeto más cercano para luego determinar si los píxeles se encuentran sombreados o iluminados [23] (ver Figura 1.2). Es decir, una vez interceptado el objeto, se genera un árbol de rayos que permitirá determinar la refracción, reflexión o sombras arrojadas, basándose en el cálculo de la intensidad del píxel.

Sin embargo, este algoritmo exige mucha potencia de cálculo cuando se trata de ejecuciones en tiempo real, como por ejemplo, la de un videojuego a 60*fps*[2] o más. Si bien es cierto que este problema se solventa un poco al hacer uso de una tarjeta gráfica en

[2] *fps*: Frames por segundos

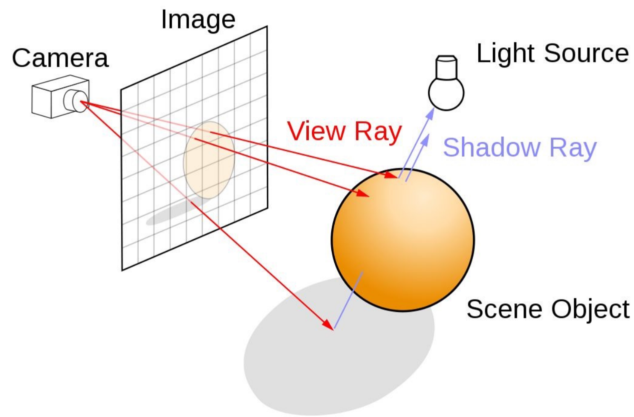


Figura 1.2: Nociones básicas del Ray Tracing [10]

condiciones, los cálculos computacionales para cada uno de los píxeles en cada momento siguen siendo demandantes, por lo que se han propuesto distintas alternativas para el mismo [15].

1.2.2. Volume Ray Casting

Es por ello que en pro de seguir evolucionando, se han empezado a aplicar técnicas de volume ray casting, que permiten incluso representar objetos de un mundo virtual que incluyen sistemas volumétricos y de partículas, proporcionando resultados de renderizado de alta calidad. Para generar una proyección en 2D de un dataset, se necesita definir previamente la posición de la cámara relativa al objeto y definir los valores RGBα para cada vóxel[3], a partir de una función de transferencia.

¿En qué se diferencia del ray tracing? Principalmente, debido a que el ray tracing solamente procesa datos de la superficie de cada objeto. Para esta variante volumétrica [11], el cálculo no se detiene en la superficie sino que atraviesa dicho objeto, muestreándolo a lo largo del rayo y, tampoco genera rayos secundarios. Adicionalmente, se trata de una técnica de representación de volumen basada en imágenes, ya que el cálculo proviene de una imagen de salida y no de los datos del volumen de entrada.

Este algoritmo se desglosa en cuatro etapas principales. Ver Figura 1.2.

1. **Ray casting**. Donde por cada píxel de la imagen inicial, se proyecta un rayo de vista a través del volumen, donde al momento de ser tocado, se encierra dentro de un objeto primitivo delimitador [8].

2. **Sampling**. Se seleccionan muestras equidistantes a lo largo del rayo.

3. **Shading**. Donde para cada punto de muestreo, se utiliza una función de transferencia capaz de recuperar un color de material RGBα y se calcula un gradiente de valores de iluminación. Luego, cada muestra se sombrea según la orientación de la superficie y la ubicación de la fuente de luz en la escena.

4. **Compositing**. Finalmente los puntos de muestreo se componen a lo largo del rayo de vista, dando como resultado el color final para el píxel que se está procesando actualmente. Este cálculo deriva principalmente de una ecuación de renderizado

[3]Vóxel: Unidad cúbica que compone un objeto tridimensional, equivalente del píxel para un objeto 2D.

donde se comienza el punto muestreado más alejado del observador y termina con el más cercana, o viceversa.

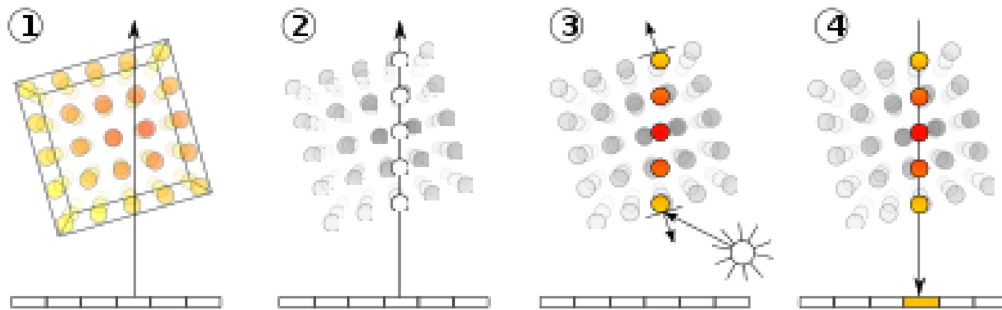


Figura 1.3: Etapas del volume ray casting [11]

Y, en cuanto a ejemplos de uso, es utilizado mayormente para la visualización de datos médicos tales como resultados de resonancias magnéticas, tomografías axiales computarizadas, e incluso también en el ámbito de los videojuegos, mejor conocido como ray marching, para la generación y aplicación del iluminado volumétrico.

1.3. Objetivos

El objetivo principal de este Trabajo de Fin de Grado, es el estudio y uso del modelo de redes neuronales *NeRF* dentro de un ambiente virtual implementado en Unity3D. Pero más en concreto, los objetivos específicos son los siguientes:

- Realizar una actividad previa de búsqueda de documentación relacionada a *NeRF* y modelos de IA similares.
- Familiarizarse con *NeRF* a partir del uso de modelos pre entrenados proporcionados por sus propios autores.
- Crear datasets propios para su uso con *NeRF*.
- Implementar conexión entre un modelo entrenado con un escenario en *Unity3D*.

Capítulo 2

Trabajos relacionados

2.1. GANCraft: Unsupervised 3D Neural Rendering of Minecraft Worlds

GANCraft [16] fue uno de los modelos considerados para utilizar a lo largo de este proyecto antes que el modelo *NeRF*. El objetivo principal que se plantea en este paper es el poder resolver un problema de traducción *world-to-world*, es decir, a partir de un escenario maquetado en *Minecraft*[1] poder generar un escenario fotorealista en 3D usando *Unity3D*, reduciendo de esa forma el proceso de modelar paisajes complejos y agilizar el desarrollo de un videojuego, además de no necesitar los conocimientos necesarios para reproducirlo más que saber jugar o usar *Minecraft*, lo cual es una ventaja.

Para lograrlo, a cada bloque en el mundo del *Minecraft* se le asigna una etiqueta semántica, como puede ser tierra, césped o agua, lo que permite al usuario tener un control sobre el estilo que se espera obtener, incluso el tipo de terreno esperado, si es áspero al estilo del desierto, si hay nieve, entre otros parámetros, y luego el escenario es tomado como una función volumétrica continua y es entrenado, generando el renderizado correspondiente. Se puede visualizar un ejemplo en la Figura 2.1.

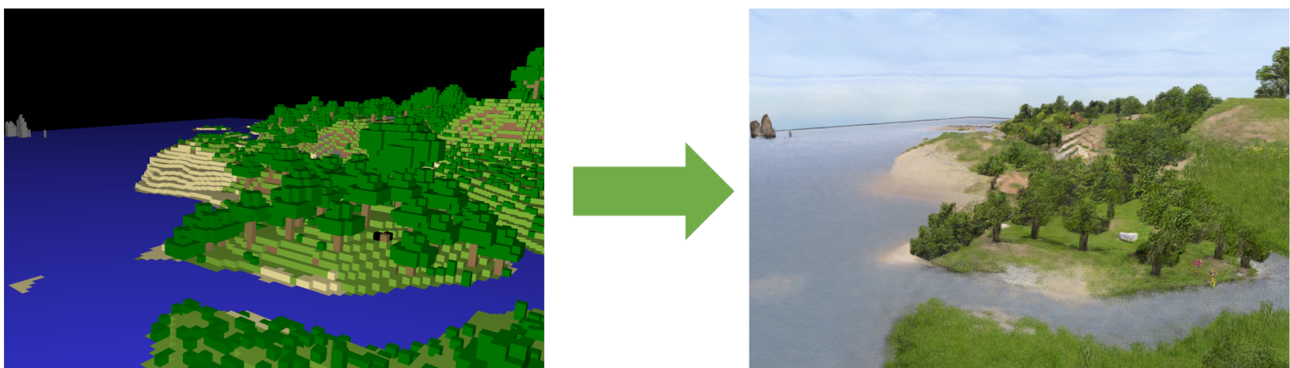


Figura 2.1: Resultados de GANCraft [16]

[1]Minecraft: `https://www.minecraft.net/es-es`

2.2. GPT-3

Por otro lado, también se propuso el uso del modelo original de *OpenAI*[2] llamado *Generative Pre-trained Transformer3* o también conocido por *GPT-3*, para la generación de un videojuego conversacional donde el final de la partida nunca fuera el mismo, a partir del procesamiento de lenguaje natural *NLP* que realiza el modelo entrenado con el jugador. Actualmente, es uno de los más potentes debido a lo difícil que resulta distinguir los textos generados de los escritos por un humano y compite con *Github Copilot*[3] o *Tabnine*[4] también publicado por *OpenAI*, aunque estos últimos estén enfocados principalmente en la sugerencia de código al momento de desarrollarlo.

2.3. NeRF in the wild: Neural Radiance Fields for Unconstrained Photo Collections

Por otro lado, existen muchas otras variaciones de Nerf desde el momento en el que el paper original fue publicado. Por ello fue propuesto el uso de algunos de ellos para el desarrollo de este proyecto, entre estos se encuentra *NeRF in the Wild* [17]. Tal y como su nombre lo indica, a diferencia de NeRF, este modelo renderiza un ambiente con vistas desconocidas a partir de datasets tomados en escenarios naturales, los cuales no necesariamente son estáticos como ocurre con NeRF.

Figura 2.2: Entrada y resultados para NeRF in the Wild [17]

Los datasets que acepta pueden tener variaciones en la iluminación o incluso obstáculos que puedan ocultar el objeto principal del cual se quiere obtener la inferencia, es decir, es ideal para poder renderizar edificios o monumentos históricos debido a que no es relevante si es de día o de noche y si hay personas alrededor, e incluso hace cuanto

[2]OpenAI: `https://openai.com/`
[3]Github Copilot: `https://github.com/features/copilot`
[4]Tabnine: `https://www.tabnine.com/`

tiempo fue tomada la imagen, ya que no interfiere con el resultado, por lo que podría ser implementado para alguna aplicación de Realidad Virtual, donde el usuario se sumerja en el ambiente y pueda recorrer dichos monumentos sin necesidad de viajar o movilizarse a ellos. Adicionalmente, también sería de gran ayuda para aquellos que se encarguen de modelar estos edificios, ya que se ahorran gran parte del proceso y los resultados que se obtienen son de muy buena calidad visual, por lo que hasta los más mínimos detalles se tomarían en cuenta. Un ejemplo de esta propuesta se puede ver en la Figura 2.2 donde se ven claramente la diferencia entre las imágenes dadas como entrada y el renderizado sigue siendo preciso con la realidad.

2.4. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding

Luego, en cuanto a calidad visual de resultado se refiere (Ver Figura 2.3.) se propuso el uso del modelo de *Instant NeRF* [20] que tiene como propósito reducir el tiempo y costo de entrenamiento y testeo. Permitiendo realizar un entreno en segundos y renderizar inferencias en decenas de milisegundos con una resolución de 1920x1080 píxeles.

Cuenta con exactamente la misma implementación e hiperparámetros que el NeRF original, pero la diferencia viene dada en el rendimiento ya que el entrenamiento se realiza paralelamente gracias al uso de kernels de *CUDA*[5]. Es decir, que es necesario contar con una tarjeta gráfica *NVIDIA* para poder realizar estos entrenamientos. No fue utilizado finalmente ya que a pesar de obtener resultados increíbles en tan poco tiempo, su publicación fue realizada cuando este proyecto ya se encontraba en curso y la curva de aprendizaje del mismo se incrementaba respecto al modelo inicial NeRF.

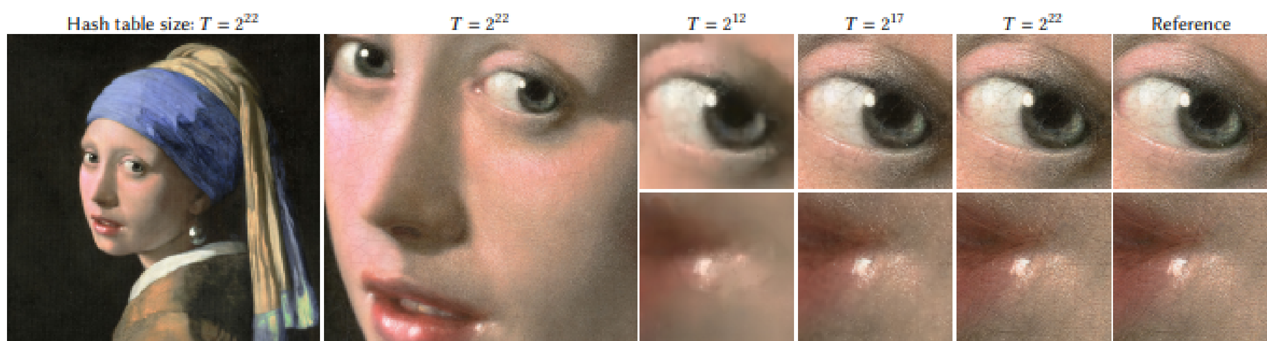


Figura 2.3: Referencia y resultados para Instant NeRF con diferentes tamaños de tablas Hash [20]

2.5. pixelNeRF: Neural Radiance Fields from One or Few Images

Uno de los retos que supone el uso del NeRF original, es que se debe contar con un dataset de imágenes estáticas para las cuales también se tenga disponible, la información

[5]CUDA: Plataforma de computación en paralelo desarrollada por NVIDIA para informática general en GPUs

referente a la posición de la cámara con la que fue tomada cada imagen en la escena, por lo que puede volverse un poco tedioso todo este proceso. Es por ello, que se propuso el uso de la variación *pixelNeRF* [24], ya que permite lograr el mismo resultado pero solo proporcionando un par de vistas para la escena, ya que el entrenamiento no es realizado para una escena de forma independiente, como se hace con NeRF sino que el modelo se encuentra previamente entrenado para una categoría de objetos similares, como por ejemplo, lámparas, por lo que puede inferir fácilmente al solo adicionar un par de imágenes nuevas. Este puede ser aplicado tanto a objetos sintéticos como escenas reales, como se puede observar en la Figura 2.4.

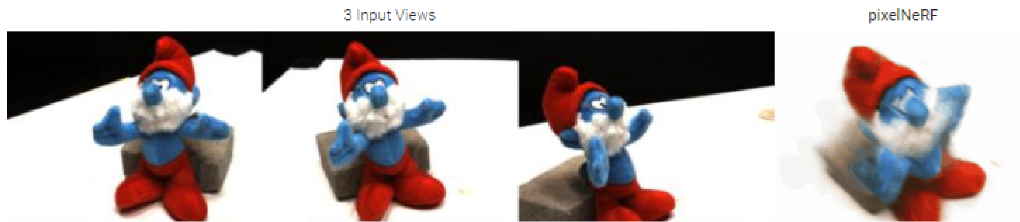


Figura 2.4: Entrada y resultados para pixelNeRF [24]

2.6. Nerfies: Deformable Neural Radiance Fields

Finalmente, se propone también el uso de otra variación del NeRF original, *Nerfies: Deformable Neural Radiance Fields* [22] que permite trabajar con escenas capturadas imágenes y vídeos de forma casual con un móvil, por ejemplo, y permite obtener igualmente inferencias de buena calidad. Ideal para generar inferencias de humanos, se puede ver un ejemplo en la Figura 2.5..

Esta implementación se considera innovadora respecto al resto de variaciones comentadas debido a la dificultad de modelar una persona y además, de hacerlo a partir de imágenes tomadas como lo haríamos habitualmente para tomarnos un *selfie*. Esto se debe a la complejidad de no contar con una estabilidad al momento de fotografiar la escena y lo que puede cambiar la visión de objetos variables como puede ser el cabello, gafas, entre otros detalles que definen a una persona. Para lograr esto, la diferencia en su implementación es que cuentan con un campo de deformación para cada una de las imágenes procesadas.

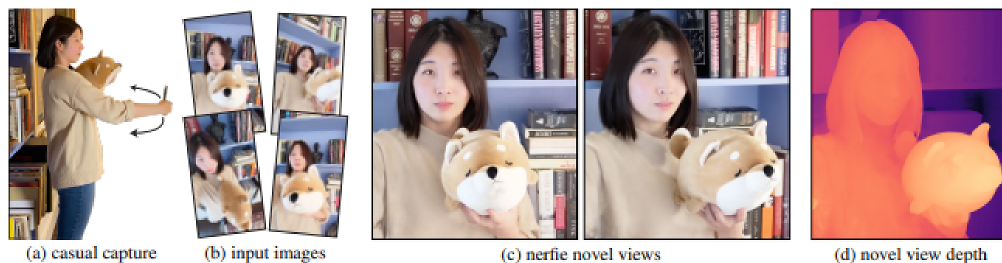


Figura 2.5: Entrada y nerfie resultante [22]

Capítulo 3

Material y métodos

3.1. Motivación

Uno de los beneficios que trae consigo trabajar con NeRF es su implementación, es el poder generar modelados con buena resolución a partir de un dataset de imágenes y además generar vistas novedosas ya que de esa forma se evita el costoso proceso de modelado 3D de las escenas, con tan solo necesitar de un dataset y las poses de la cámara para lograrlo. Adicionalmente, la cantidad de documentación disponible que existe es algo positivo, así como repositorios públicos [3] y diversos tutoriales para utilizarlo.

Es por ello, que para un entendimiento inicial del modelo se hizo un seguimiento de talleres propuestos por PyImageSearch [12] [13] [14] y la visualización de la presentación del paper original dado por uno de sus autores, John Barron, titulado *Understanding and Extending Neural Radiance Fields* [4].

3.2. Funcionamiento de NeRF

Para este modelo, se tiene como punto de partida un conjunto disperso de imágenes tomadas a una escena real o a un objeto modelado de forma sintética, a partir de allí se consideran los metadatos de la cámara con la que fueron tomadas las imágenes, en concreto la posición y la orientación de la misma para cada uno de esos instantes. Y, se llevan a cabo los siguientes pasos, simplificados en la Figura 3.1.

1. **Generación de rayos**, se trazan rayos dentro de la escena para cada uno de los píxeles de cada una de las imágenes, y estos se intersectan dentro de la escena en 3D.

2. **Muestreo**, se toman puntos de muestreos ubicados en cada uno de los rayos, por lo que estos contendrán información única referente a la posición en los ejes (x,y,z) y la dirección del rayo dada por (θ, ϕ).

3. **Deep Learning**, se optimiza una red neuronal del tipo *MLP*[1] con la información recolectada en el muestreo, es decir, los valores vectoriales 5D para cada imagen. Se obtiene una salida con la inferencia referente a el color emitido C y a la densidad de volumen σ, en formato RGBσ de cada punto que fue insertado.

[1] *MLP*: Multilayer perceptron

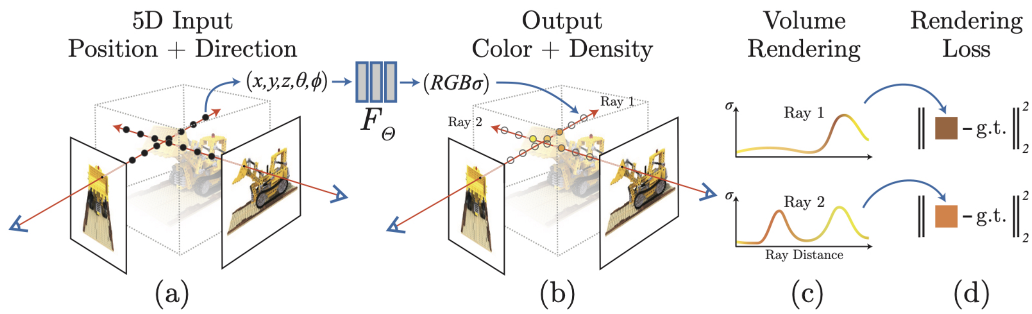


Figura 3.1: NeRF pipeline [19]

4. **Renderizado volumétrico**, se aplica esta técnica para obtener el color correspondiente a cada píxel en una imagen 2D, tal y como se comentó en la introducción del proyecto.

5. **Pérdida fotométrica**, se considera el ECM[2] entre el color inferido de cada píxel con el color real correspondiente para minimizar dicho error en las diferentes vistas generadas, permitiendo obtener una escena más coherente con la real ya que este parámetro es utilizado en la función de pérdida de la red neuronal para entrenarse.

3.2.1. Arquitectura del MLP

Para lograr la representación final de la escena, NeRF propone un MLP cuyas capas se encuentran totalmente conectadas entre sí con la siguiente arquitectura. Ver Figura 3.2.

- Se restringen las primeras 8 capas para que reciban como entrada los valores (x,y,z) del punto, para lograr la predicción de la densidad del volumen (σ). Estas capas utilizan funciones de transferencia *ReLU* [9] y 256 canales por cada una de ellas y la σ, viene dada en un vector de 256 dimensiones.

- Seguidamente, se concatena este último vector con (θ, ϕ) y es pasado a unas últimas capas adicionales. Estas capas finales también utilizan funciones de transferencia *ReLU* pero en este caso solo 128 canales, dando como resultado los valores RGB para la escena final.

Teniendo esto en cuenta, en base a los resultados expuestos en el paper original, para optimizar un NeRF se tarda entre un día o dos aproximadamente, dependiendo de la cantidad de repeticiones o épocas para las cuales se defina el entrenamiento, ya que mejora la resolución del resultado y solo necesita de una GPU para lograrlo. Luego, para renderizar una inferencia se tarda entre menos de un segundo a 30 segundos aproximadamente, dependiendo de la resolución. Sin embargo, para el caso de este proyecto, se hará uso principalmente de una versión simplificada del mismo denominada Tiny NeRF.

3.3. Tiny NeRf

Se trata de una versión simplificada de NeRF, ha sido utilizada por la rapidez en el entrenamiento respecto a la versión original y su facilidad de uso. A diferencia de

[2]Error cuadrático medio: Mide el promedio de los errores al cuadrado

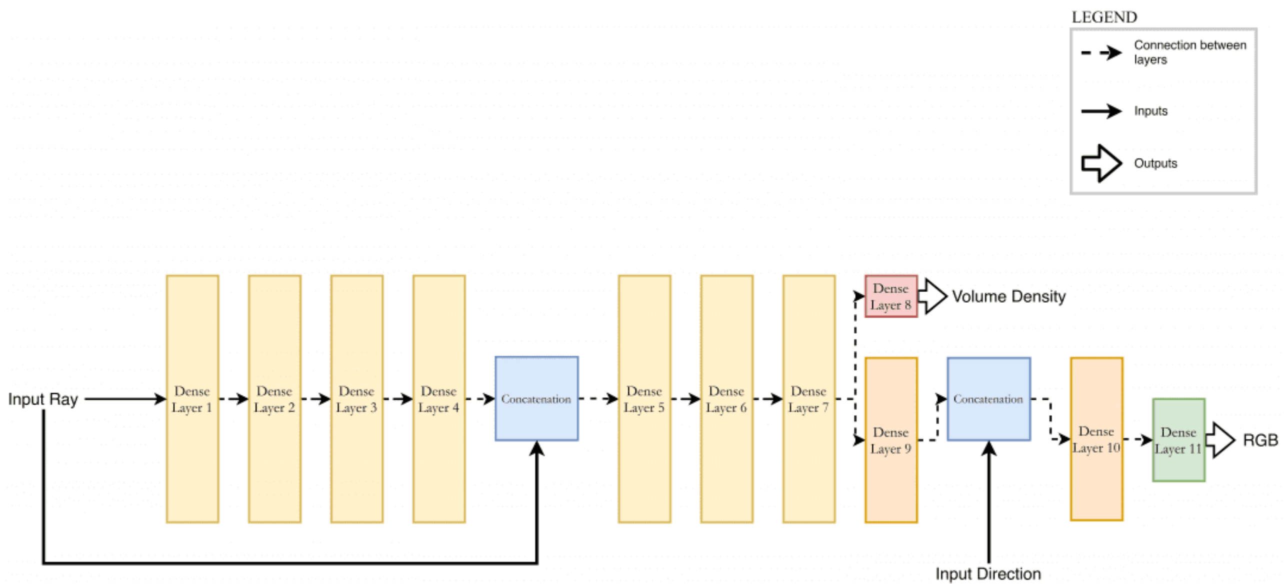


Figura 3.2: Arquitectura del MLP

NeRF, Tiny NeRF te permite visualizar cómo se va optimizando a medida que se va ejecutando cada tantas épocas y se genera una gráfica con el *PSNR*[3] obtenido. Este último parámetro es interesante ya que el *PSNR* está relacionado con el ECM que se comentaba anteriormente, por lo que se concluye que cuanto mayor sea el valor del *PSNR*, mejor se ha reconstruido la inferencia basada en la imagen original, y por lo tanto, ha disminuido el ECM.

3.3.1. Arquitectura del MLP

En general, la arquitectura MLP para Tiny NeRF, es exactamente la misma que para NeRF, a diferencia de la entrada. En este caso lo que se recibe como entrada son los valores correspondientes al RGB de la imagen, por lo que la inferencia del volumen para cada píxel viene dada a partir de imágenes totalmente estáticas.

3.3.2. Tecnologías utilizadas

A instancias prácticas, se hizo uso del Tiny NeRF en un *Google Colab Notebook*, con una sesión activa del backend de *Google Compute Engine (GPU)* que utiliza *Python 3* y las librerías necesarias para el notebook son: *Tensorflow2 y keras, numpy, matplotlib, PIL.image, ipywidgets, IPython.display, base64, json, time, imageio, os, sys, google colab*.

3.3.3. Datos de entrada y resultados

Se utilizaron datasets de objetos sintéticos proporcionados por los autores, los cuales contienen imágenes para el entrenamiento, validación y testeo de 100 imágenes cada uno. Los principales datasets para objetos sintéticos que se encuentran son: una silla antigua, un micrófono, un barco pirata, una batería, un ficus, un tractor hecho con legos, un perrito caliente y unas esferas de diferentes materiales. Asimismo, también se disponen de datasets para objetos en escenarios naturales, con ambiente de fondo e

[3]PSNR: *Peak Signal-to-Noise Ratio, utilizado para medir la calidad de una imagen digital generada.*

iluminación variable, aunque el enfoque durante este apartado será los objetos sintéticos principalmente. Todos estos datasets están disponibles en: Ver apéndice A.

Por otro lado, también se proporciona la matriz de transformación de cada imagen, es decir, su pose y la distancia focal, correspondiente a la vista de la cámara para el momento en el que se tomó cada imagen. Los ficheros se encuentran en formato JSON. Ver listing 3.1 con ejemplo.

Listados 3.1: Metadatos de la cámara.

```
{
    "camera_angle_x": 0.6911112070083618,
    "frames": [
        {
            "file_path": "./train/r_0",
            "rotation": 0.012566370614359171,
            "transform_matrix": [
                [
                    -0.9999021887779236,
                    0.004192245192825794,
                    -0.013345719315111637,
                    -0.05379832163453102
                ],
                [
                    -0.013988681137561798,
                    -0.2996590733528137,
                    0.95394366979599,
                    3.845470428466797
                ],
                [
                    -4.656612873077393e-10,
                    0.9540371894836426,
                    0.29968830943107605,
                    1.2080823183059692
                ],
                [
                    0.0,
                    0.0,
                    0.0,
                    1.0
                ]
            ]
        },
        ...
}
```

Para poder utilizar Tiny NeRF, a pesar de que tanto el notebook, como las imágenes y sus poses eran proporcionadas para que pudiera funcionar correctamente. Se realizaron modificaciones al código original debido a dificultades que se fueron encontrando.

1. **Imágenes de entrada.** El formato de las imágenes de entrada debían estar en

RGBα, por lo que fueron cargadas al notebook desde la carpeta de Drive donde se encontraban y se les extrajo ese valor con la librería *PIL*. Por otro lado, debido al espacio ocupado en RAM por los tensores, la optimización del modelo se ralentizaba y, en este caso, la sesión de colab se cancelaba por completo dejando de funcionar, por lo que como solución se tuvo que reducir la resolución de las imágenes, siendo el mayor tamaño posible de 100x100 píxeles.

2. **Imágenes de salida.** Una vez terminada la optimización, se podían visualizar de forma interactiva las vistas deseadas del objeto de forma inmediata, gracias a un slider generado con la librería *ipywidgets* que permite definir valores de: θ,ϕ, y radio. Adicionalmente, también se genera un vídeo con todas las vistas inferidas con la librería *imageio*, se pueden visualizar diferentes de los vídeos generados durante este proyecto. Y, en este caso, se añadió el guardado del modelo ya entrenado en ficheros con formato *.h5* para poder ser utilizado en el servidor implementado.

Finalmente, en comparación con NeRF, el entrenamiento tardaba alrededor de una hora o dos, sobretodo dependiendo en qué tan sobrecargada se encontraba la sesión de *Google Colab* en ese momento, por lo que efectivamente era mejor opción para esperar menos tiempo. Sin embargo, también es importante recalcar que Tiny NeRF, genera inferencias de menor calidad y la definición de los detalles para los objetos baja. Los objetos sintéticos para los cuales la resolución resultó ser más detallada en comparación al resto fueron la batería y el tractor hecho con legos, por lo que se decidió guardar el modelo entrenado de este último para continuar con la implementación del proyecto.

3.4. Comunicación entre Tiny NeRF y Unity3D

En esta etapa del proyecto, se buscó información acerca de diferentes plugins que pudieran existir para realizar esta comunicación para agilizar el proceso. En principio, no se encontraron muchas opciones que fueran funcionales a día de hoy o que fueran compatibles con *Python3* y la versión estable del 2020 para *Unity3D*. De hecho para este último, la librería *NetworkConnection*[4] se encuentra en una versión obsoleta y su siguiente versión sigue en desarrollo, por lo que no es recomendable su uso.

A pesar de ello, se encontró un plugin de uso público llamado *Unity3D-Python-Communication* (Disponible en: Ver apéndice A) que hace uso principalmente de la librería *ZeroMQ*[5] y se fue modificando progresivamente de acuerdo a las necesidades que se fueron encontrando durante el desarrollo del proyecto, que serán explicadas en los siguientes subapartados.

3.4.1. Servidor

Se implementó en *Python3* un servidor TCP, el cual es ejecutado de forma local en el puerto 5555. Y además, cuenta con las siguientes características.

- **Tiny NeRF.** Permite la lectura de Tiny NeRF previamente entrenado almacenado en un fichero de extensión *.h5*. Adicionalmente, cuenta con la lógica de inferencia para Tiny NeRF.

[4] NetworkConnection:`https://docs.unity3d.com/2020.3/Documentation/Manual/class-NetworkConnection.html`

[5] ZeroMQ: `https://zguide.zeromq.org/`

- **Conexión con el cliente.** Permite conectarse con el cliente y se queda a la escucha, debido a que el cliente es capaz de enviarle los valores de θ,ϕ, y radio, tal y como se comentó en el apartado de Tiny NeRF.

Listados 3.2: Lógica del servidor.

```
1    if __name__ == "__main__":
2        context = zmq.Context()
3        socket = context.socket(zmq.REP)
4        socket.bind("tcp://*:5555")
5        model = load_model('./TrainedModels/tiny_nerf_lego.h5')
6
7        while True:
8            coordinates = socket.recv(1024)
9            coordinates = array.array('f', coordinates).tolist()
10           print("\nSpherical coordinates received: %s" % coordinates
                 )
11
12           start = time.time()
13           img = f(**{"theta": coordinates[0], "phi": coordinates[1],
                  "radius":coordinates[2]})
14           end = time.time()
15           print("\nThe execution time of NeRF was of: ", end - start
                 , " seconds\nSending encoded img... ")
16           socket.send(img)
```

- **Obtención de inferencia y envío.** La inferencia que ha sido obtenida en valores RGB en valores decimales, es transformada a valores enteros del 0 al 255 y ha sido generada con un fondo negro. Es codificada en bytes y se envía al cliente esta información para ser plasmada en la escena de Unity3D.

Para obtener mayor detalle de la arquitectura y cómo ejecutar el servidor, su implementación se encuentra disponible en el fichero `Server/server_v2.py` dentro del repositorio asociado a este TFG (Disponible en: Ver apéndice A. Para poder utilizarlo, es recomendable preparar un entorno virtual, por ejemplo, con *pyenv*[6] con el fichero `requirements.txt` también disponible en el repositorio.

3.4.2. Cliente

Se implementó un cliente para el escenario en *Unity3D*, que fuera capaz de enviar información referente a la posición en la que esperaba obtener una inferencia de Tiny NeRF. Como fue comentado anteriormente, se hizo uso de la librería *ZeroMQ*, pero en el caso del cliente, como se iba a trabajar con C# se hizo uso de la librería *NetMQ*[7] ya que proporciona una abstracción para colas de mensajes asíncronos, uso de sockets, entre otras funcionalidades que la librería *NetworkConnection* de *Unity3D* no puede proporcionar actualmente. La implementación del cliente cuenta con la siguiente estructura de clases dentro del proyecto de Unity3D.

[6]pyenv: url:https://github.com/pyenv/pyenv
[7]NetMQ: `https://github.com/zeromq/netmq`

- **RunAbleThread.** Clase abstracta que permite la activación de un hilo con el método *Run()* y cancelación de la ejecución de un hilo para la escena con el metodo *Stop()*.
- **NeRFRequester.** Clase hija de RunAbleThread que implementa el funcionamiento del método Run(), donde se realiza la conexión vía sockets al puerto 5555 y recoge las coordenadas para ser enviadas e implementa el funcionamiento del método TaskOnClick(RequestSocket client, bool gotMessage) que recibe la inferencia de TinyNeRF y la aplica como Sprite a la escena.

Listados 3.3: Lógica del cliente.

```
1      public class NeRFRequester2 : RunAbleThreadV2
2  {
3      private byte[] inputs;
4      private Texture2D texture;
5      private bool gotMessage = false;
6      protected override void Run()
7      {
8          ForceDotNet.Force();
9          using (RequestSocket client = new RequestSocket())
10         {
11             client.Connect("tcp://localhost:5555");
12             sendButton.onClick.AddListener(delegate{TaskOnClick(client,
                  gotMessage);});
13             while (Running) {
14                 gotMessage = false;
15
16                 float theta = CameraMovement.Instance.thetaValue;
17                 float phi = CameraMovement.Instance.phiValue;
18                 float radius = CameraMovement.Instance.radiusValue;
19                 float[] coordinatesValues = { theta, phi, radius };
20
21                 inputs = new byte[coordinatesValues.Length * 4];
22                 Buffer.BlockCopy(coordinatesValues, 0, inputs, 0, inputs.
                      Length);
23             }
24         }
25         NetMQConfig.Cleanup();
26     }
27
28     public void TaskOnClick(RequestSocket client, bool gotMessage)
29     {
30         try
31         {
32             byte[] message = null;
33             client.SendFrame(inputs);
34             while (Running)
35             {
36                 gotMessage = client.TryReceiveFrameBytes(out message);
```

```
                if (gotMessage)
                {
                    Debug.Log("Texture applied..." +  System.Text.Encoding
                       .UTF8.GetString(message));
                    texture = new Texture2D(100,100);
                    texture.LoadImage(message);
                    texture.name = "NeRF Texture";
                    NeRFImg.texture = texture;
                    break;
                };
            }
            if (gotMessage) Debug.Log("Image bytes received");
        }
        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}
```

Una vez ya se tiene la conexión Cliente-Servidor funcionando, el orden de ejecución sería el siguiente: ejecutar el servidor, seguidamente ejecutar el cliente que sería ejecutar el play de la escena en *Unity3D*, mover la cámara para obtener la vista deseada y pulsar el botón de **Send** ubicado en la pantalla, el servidor recibe la petición y procesa la inferencia para enviarla como respuesta, finalmente es recibida en el cliente y se actualiza por pantalla el sprite correspondiente, coincidiendo con la escena en 3D.

3.5. Escenario en Unity3D

Una de las principales razones por las que se planteó la implementación del servidor era para evitar la acción de guardar en memoria las inferencias obtenidas para cada una de las vistas posibles y evitar tener que asignar dependiendo de la posición de la cámara, cada uno de los diferentes sprites posibles, debido a que sería muy extenso y poco modular en la implementación. Por otro lado, también se esperaba que la escena fuera a tiempo real, mejorando la experiencia del jugador o que al menos se encontrase con un entorno virtual donde el tiempo de respuesta sea corto.

Una vez verificada que la conexión del Cliente-Servidor funcionaba, el enfoque del proyecto fue dado a un escenario en 3D en el que se pudiera adicionar un sprite proveniente de Tiny NeRF, como lo es habitual en los videojuegos 2.5D en la actualidad [7]. Debido a que se estaba usando el dataset de tractor de lego, la escena debería ir acorde, por lo que se propuso la representación de un garage con diferentes herramientas.

3.5.1. Assets utilizados y diseño de la escena

Para agilizar el proceso de desarollo de la escena, se utilizó el asset *Simple Garage* [6] del cual se adicionaron varios de sus prefabs[8]. Se trata de un asset gratuito, disponible

[8]Prefab: Objetos reutilizables, y creados con una serie de características dentro de la vista proyecto

en la *AssetsStore*[9] de *Unity*, en este caso, compatible con la versión estable del 2020 de *Unity3D* que se está utilizando.

Figura 3.3: Escena del garage

3.5.2. Manejo de controles para el jugador y Tiny NeRF

Este subapartado está relacionado al planteamiento propuesto para el Cliente-Servidor, ya lo importante no era solo que funcionara esta conexión sino que para el jugador fuera intuitiva. Para ello se fueron haciendo diferentes propuestas de forma iterativa.

- **Sliders.** En primer lugar, se replicó la acción que se tiene dentro del notebook, de poder mover sliders dentro de un rango determinado para determinar los valores de (θ, ϕ, radio) de los que se quiere obtener un Tiny NeRF.

  En principio, adicionar a la escena esta interfaz fue sencillo debido a que se hace uso del *Canvas*[10] dentro la escena, y para obtener los valores del slider se utilizó la propiedad *PlayerPrefs* de esa manera se guardaban las últimas coordenadas entre una sesión y otra.

- **Movimiento de la cámara.** Este paso fue uno de los fundamentales para el proyecto, ya que daría con la versión final. Para evitar el uso de sliders y que se diera la impresión de un movimiento dentro de un videojuego a tiempo real, se decidió utilizar el movimiento de la cámara y, tanto su posición como rotación respecto al centro de la escena (donde se mostraría la inferencia), para obtener los datos que se necesitaban, (θ, ϕ, radio). Theta, viene dado por la coordenada y en ángulos de Euler; Phi, viene dada por la coordenada x en ángulos de Euler y el radio, viene dado por un valor definido por defecto, el cual se puede modificar con el movimiento de la cámara. Para mover la cámara de un lado a otro, se utilizan los controles del teclado que vienen definidos por defecto en la clase *Input* con el método *GetAxis*

[9]AssetsStore: https://assetstore.unity.com/

[10]Canvas: Área donde todos los elementos UI deben estar dentro de un escenario en Unity3D

y, para ajustar el radio se utilizan las teclas; *E*, para disminuir su valor y *Q*, para aumentar su valor.

Sin embargo, llegado a este punto, para obtener los valores dentro de la clase *NeRFRequester* se tuvieron que crear variables estáticas provenientes de la clase *CameraMovement*, ya que si no se generaban errores debido a que el requester, no hereda de *MonoBehaviour* sino de *RunAbleThread* como se comentó anteriormente, eliminando por completo la implementación anterior del *PlayerPrefs* y las clases relacionadas a ella.

Se adicionó el objeto de la cámara al objeto del cliente de NeRF en el inspector, para que siempre se tuviera una visión fiel con el plano, aunque surgió otro problema y es que a pesar que la conexión vía sockets funcionaba, lo que devolvía Tiny NeRF no correspondía con la visión que se tenía en la escena, es por ello que se movió toda la escena ya no solo la cámara, al punto (0,0,0) dentro del mundo de Unity3D. Y, se tuvo que multiplicar por *-1* el valor correspondiente a phi. Una vez allí, se adicionó una mesa donde se "posaría"el tractor de lego.

La lógica sería la siguiente, la cámara, ubicada en la misma posición en el mundo que el objeto vacío del cliente NeRF dentro de la escena en el punto (0,0,0), se trasladaría tantas posiciones como el radio indicado por defecto para Tiny NerF fuera hacia atrás, y una vez allí, con la visión entera de la mesa y ubicado dónde se posaría el tractor, se movería la cámara como el jugador quisiera y enviaría la petición al servidor. Pero, al realizar pruebas del movimiento sobrepasando lo que sería la parte superior del tractor y yendo dirección a su parte trasera, la cámara se quedaba volteada por lo que daba la impresión que la escena se encontraba del revés. Para solucionarlo se limitó el movimiento de la cámara a 80º, desde el borde de la mesa hasta la parte exacta superior del tractor, de esta manera si se quisiera visualizar la parte trasera el tractor, se debería llegar al límite y moverse hacia los lados para luego bajar, parecido a la lógica que tendría un cubo Rubik para mover sus piezas.

Listados 3.4: Lógica para el movimiento de la cámara.

```
1    public class CameraMovement: Monobehaviour
2    {
3        ...
4        void Update()
5        {
6            float thetaRotation = Input.GetAxis("Horizontal") *
                rotationSpeed * Time.deltaTime;
7            float phiRotation = Input.GetAxis("Vertical") *
                rotationSpeed * Time.deltaTime;
8
9            xAngle -= phiRotation;
10           yAngle -= thetaRotation;
11
12           // use Mathf.clamp to limit the rotation
13           xAngle = Mathf.Clamp(xAngle, 0, 80);
14           transform.rotation = Quaternion.Euler(xAngle, yAngle, 0);
15           UpdateNeRFValues();
```

```
16          }
17
18          void UpdateNeRFValues()
19          {
20          // Multiplied by -1 because of views resulting with Tiny Nerf
               and those coordinates
21          phiValue = transform.eulerAngles.x * -1;
22          thetaValue = transform.eulerAngles.y;
23          radiusValue = CameraScale.Instance.distanceToTarget;
24          }
25      }
```

- **Inferencia de Tiny Nerf.** Para este momento, se decidió cambiar la escala de todos los objetos en la escena de escala 1 a escala 2 en todas sus coordenadas para el mundo de Unity, debido a que daba la impresión que el Tiny NeRF devuelto era enorme.

  Luego, visualizando mejor la inferencia, esta se devolvía con el fondo negro comentado anteriormente y mucho ruido en el contorno del tractor, por lo que para dar una impresión más realista dentro del escenario en 3D, la imagen fue procesada desde el servidor con un procedimiento de threshold proporcionado por la librería de *opencv*. Dentro de este procedimiento aquellos píxeles cuyo valor fuera mayor a 50, se consideraba como 255 para poder ser eliminado el fondo. Luego, como se puede ver en el Listing 3.5 se obtiene el canal alpha para las transparencias a partir del método de *threshold* y es fusionado con los valores RGB de la inferencia, aunque en este caso, el canal B y el canal R están intercambiados porque *opencv* los reconoce de esa forma, luego se codifica en bytes y es enviado al cliente como respuesta.

Listados 3.5: Thresholding a Tiny NeRF.

```
1   def f(**kwargs):
2       c2w = pose_spherical(**kwargs)
3       rays_o, rays_d = get_rays(HEIGHT, WIDTH, focal, c2w[:3,:4])
4       rgb, depth, acc = render_rays(model, rays_o, rays_d, near=2.,
           far=6., N_samples=N_samples)
5       rgb_255 = rgb * 255
6       rgb = tf.dtypes.cast(rgb_255, tf.uint8)
7
8       tmp = cv2.cvtColor(rgb.numpy(), cv2.COLOR_BGR2GRAY)
9       _,alpha = cv2.threshold(tmp,50,255,cv2.THRESH_BINARY)
10      b, g, r = cv2.split(rgb.numpy())
11      rgba = [b,g,r, alpha]
12      dst = cv2.merge(rgba,4)
13      cv2.imwrite("lego.png", dst)
14
15      dst_tensor = tf.convert_to_tensor(dst, dtype=tf.uint8)
16      rgb_encoded_array = tf.io.encode_png(dst_tensor)
17
18      return rgb_encoded_array.numpy()
```

3.6. Generación de dataset propio

Finalmente, para no solamente verificar los resultados obtenidos con datasets proporcionados por los autores de NeRF, se han tomado datasets propios con el móvil con una cámara de 48mp[11], tanto a objetos estáticos como naturales. Estos datasets cuentan con las siguientes características. Ver Figura 3.4

Nombre del dataset	Cantidad de imágenes	Resolución en píxeles
Orquids	255	2088x4640
Phasma	154	1280x720
Keyboard	148	1280x720

Figura 3.4: Características de los datasets.

Adicionalmente, por si fuera necesario se implementó un script capaz de obtener los frames capturados en un vídeo y generar un dataset, ubicado en el fichero *Videos2Frames/video2frame.py* dentro del repositorio del TFG (Disponible en: Ver apéndice A).

Luego, para poder utilizar estos datasets propios, tomando en cuenta los requerimientos para NeRF como para Tiny NeRF, se necesitan de los metadatos de la cámara para cuando las imágenes fueron tomadas. En ese caso, se hace uso de COLMAP[12] siguiendo los pasos dados en el repositorio del paper *Local Light Field Fusion: Practical View Synthesis with Prescriptive Sampling Guidelines* [18] para obtener las poses correspondientes (Disponible en: Ver apéndice A).Y se repite el proceso para utilizar Tiny NeRF, ajustándolo a los parámetros para los datasets nuevos.

[11]mp: Megapíxeles
[12]COLMAP: https://colmap.github.io/

Capítulo 4

Resultados obtenidos

4.1. Especificaciones

Para el desarrollo de todo el proyecto y en concreto la realización de las diferentes pruebas, cuando no se utilizaba el notebook de *Google Colab*, se contaba con un ordenador con las siguientes especificaciones:

- Procesador Intel(R) Core(TM) i5-7200U. Con velocidad de reloj de 2.5Ghz.
- Sistema operativo Windows 10 de 64 bits con procesador basado en x64.
- Tarjeta Gráfica Integrada Intel HD Graphics 620.
- Memoria RAM de 8GB.

4.2. Optimización de Tiny NeRF

Tal y como se comentó en apartados anteriores, cuando se optimiza un Tiny NeRF, no solo resulta el modelo entrenado, sino que por cada 50 epocs que pasan, se guarda el valor del PSNR calculado. Es por ello que a modo de experimentación, se va a realizar una comparativa entre PSNRs al optimizar un Tiny NeRF para los siguientes datasets: Tractor de lego, orquids, phasma y keyboard.

Para optimizar un Tiny NeRF, se hace uso del notebook de colab, por lo que no se considerará en la comparativa el tiempo que tarda en optimizarse ya que para todos tardó 30 minutos. Los resultados se pueden ver en la siguiente tabla. Ver figura 4.1.

Dataset	epochs	PSNR
Tractor de lego	1000	~24
Orquids	1000	~16
Phasma	1000	~18
Keyboard	1000	~16

Figura 4.1: PSNR para diferentes Tiny NeRF.

Al ver los resultados obtenidos para la misma cantidad de épocas, se puede confirmar que cuanto mayor sea el PSNR, menor será el ECM por lo que la similitud entre la inferencia generada y lo que sería la imagen original es mayor, y es más cercano a la realidad. También es importante destacar, que a pesar que la diferencia entre cada valor de PSNR obtenido no sea mucha, visualmente se puede comprobar que para los datasets propios el entrenamiento no se obtuvieron buenos resultados. Para el tractor de lego, se puede ver el objeto perfectamente e incluso da la impresión que hubiese sido modelado con *PixelArt*[1], pero para los casos de los datasets propios, son imágenes distorsionadas de las cuales no se les puede observar la silueta a los objetos.

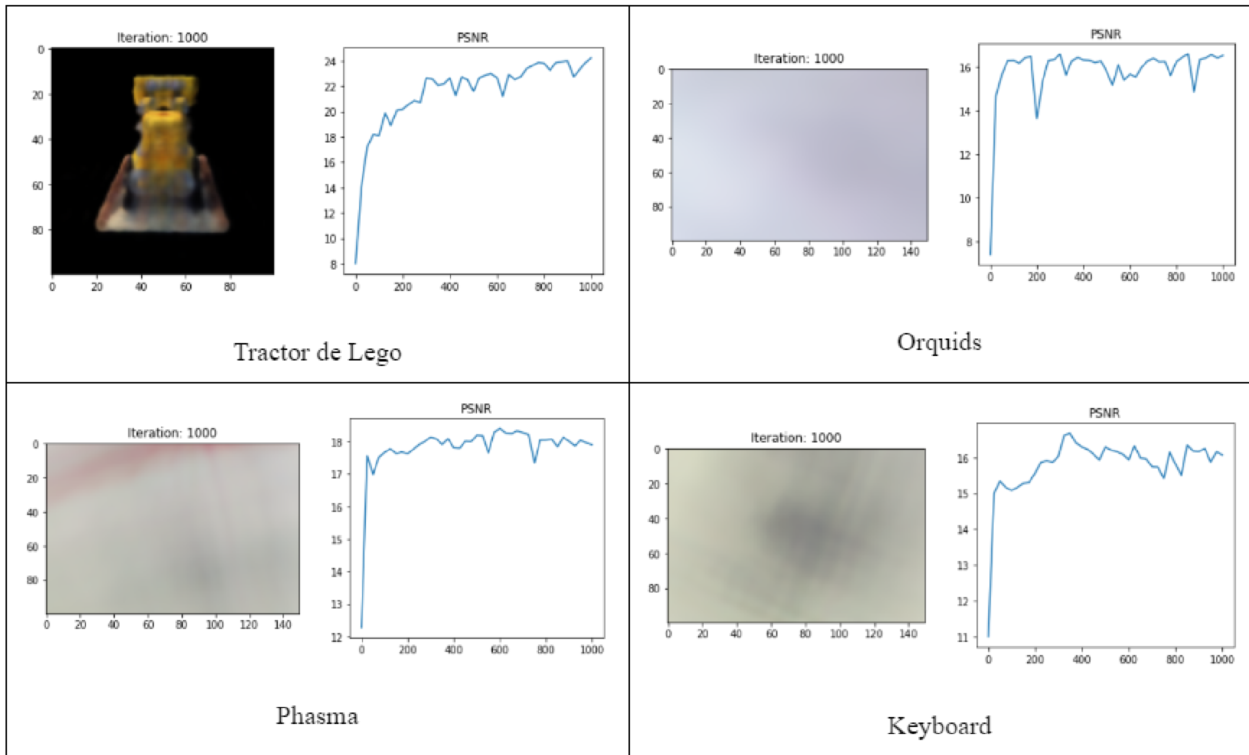


Figura 4.2: PSNR para diferentes Tiny NeRF.

Existe una posibilidad en la que estos resultados se hayan visto afectados debido a que el dataset del tractor de lego eran imágenes en formato *.png* con fondo transparente y las imágenes del dataset propio se encuentran en un ambiente real y con iluminación variable.

4.2.1. Prueba con NeRF

En vista de los resultados poco alentadores con los datasets propios, a pesar de que el enfoque principal del proyecto fue realizado para Tiny NeRF, se decidió realizar una pequeña prueba de generar un NeRF para cada dataset propio y obtener un vídeo de referencia, siguiendo los pasos en el repositorio oficial [3] y ejecutándolo en un servidor proporcionado por la Universidad de La Laguna.

Nuevamente, los resultados fueron regulares, pero solamente al momento de exportar en vídeo los resultados obtenidos disponibles en: Ver Apéndice A. A continuación se pueden observar los resultados que se iban obteniendo a lo largo del entrenamiento, a

[1] PixelArt: Arte digital, donde las imágenes son creadas y editadas al nivel del píxel.

diferencia de NeRF se logran ver los objetos con mayor detalle aunque también es cierto que llevan mayor cantidad de épocas en comparación. Ver figura 4.3.

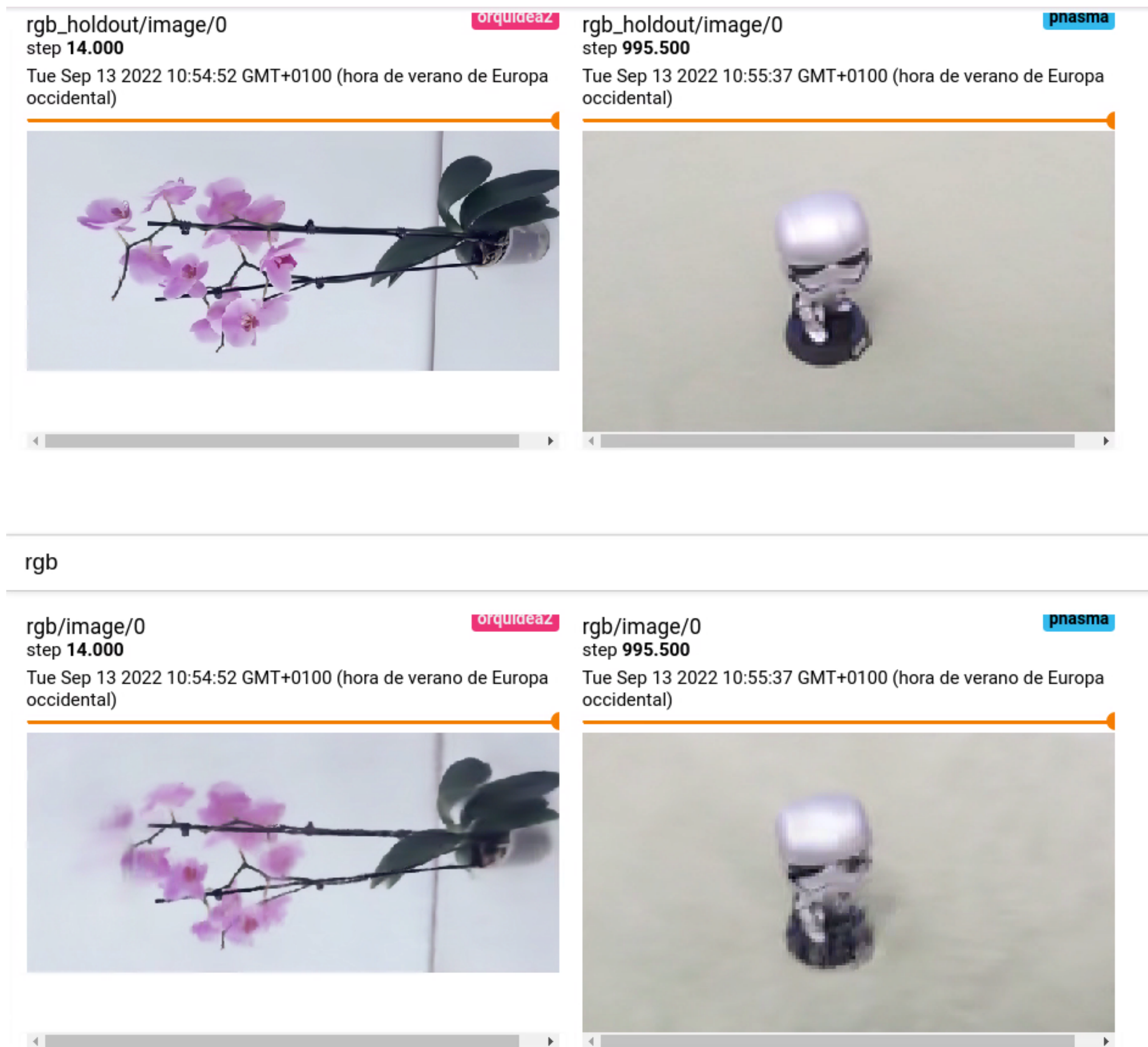


Figura 4.3: Resultados de NeRF durante el entrenamiento.

4.3. Escenario en Unity3D

Por otro lado, en la escena en Unity3D sí se pudo integrar correctamente el Tiny NeRF como fue comentado anteriormente, logrando alcanzar uno de los objetivos de este Trabajo de Fin de Grado. En este caso, se pudo realizar una pequeña experimentación en donde se realizaba la inferencia con el servidor ejecutándose en el ordenador del que las especificaciones fueron dadas en el apartado anterior y en donde se realizaba la misma inferencia en un ordenador que contara con una GPU.

Este punto es importante, ya que es lo que permitiría dar la impresión de encontrarse en un ambiente virtual que se ejecuta en tiempo real.

- **Para el primer ordenador, la inferencia tardaba entre 15 a 20 segundos**, dependiendo en qué tan sobrecargado de ejecuciones se encontraba en ese momento, ya que en ocasiones, durante el desarrollo del servidor este dejaba de funcionar, se congelaba la ejecución en Unity3D o incluso se apagaba, mostrando un gran consumo de CPU en el administrador de tareas y ralentizando el avance del proyecto.
- **Para el segundo ordenador, la inferencia tardaba alrededor de 0.2 segundos**, por lo que se pudo confirmar lo importante de contar con una GPU para la ejecución de este proyecto, ya que el esfuerzo computacional se reducía notablemente y no llegaba a congelarse la ejecución en ningún momento.

Finalmente, así se vería el escenario entero una vez inferido el Tiny NeRF. Ver figuras 4.4 y 4.5.

Figura 4.4: Tiny NeRF en Unity3D.

Figura 4.5: Tiny NeRF en Unity3D.

Capítulo 5

Conclusiones y líneas futuras

Una vez culminado el proyecto se puede confirmar que tanto el objetivo principal como los objetivos específicos fueron alcanzados. Considero que se debe principalmente, a la cantidad de documentación disponible para NeRF y todas sus variantes. El desarrollo con NeRF y en general de diferentes modelos que infieren contenido con un par de imágenes o descripciones, como es el caso DALL·E 2 [2] se encuentran en auge actualmente, muchos desarrolladores hablan del tema a diario y el contenido relacionado es impresionante, por lo que el proceso del desarrollo se ha vuelto más ameno e interesante. Asimismo, considero que he podido asentar conocimientos respecto al uso de modelos de IA más complejos y, mejorar mis habilidades desarrollando con Unity3D y C# debido a los obstáculos que tuve que atravesar y solucionar, que fueron totalmente diferentes a experiencias previas que había tenido con el motor gráfico para la asignatura de Interfaces Inteligentes.

Sin embargo, a pesar de esto las expectativas que tenía respecto al uso de datasets propios eran altas y los resultados no han sido muy favorables, por lo que puede ser un punto a contemplar en el futuro, bien sea por la cantidad de imágenes que se utilizan, la cantidad de épocas definida, la resolución de las imágenes o incluso contar con un ordenador con una GPU con las características necesarias para soportar estos entrenamientos y obtener mejores resultados. Por lo que también me gustaría recalcar que debido al punto anterior y las características de mi ordenador actual, considero que el desarrollo de este proyecto se ralentizó un poco, en ocasiones podía pasar horas esperando a que la conexión vía sockets respondiera o incluso el ordenador dejaba de funcionar y debía empezar desde cero, por lo que el proceso no era muy ágil y se necesitaba mucha paciencia.

Luego, considero que como líneas futuras se podrían plantear los siguientes escenarios:

- Encontrar los parámetros necesarios para obtener buenos resultados tanto en NeRF como Tiny NeRF para datasets propio.

- Probar la implementación con otras variantes como NeRF in the Wild para el cambio de iluminación en un videojuego o Instant NeRF que promete mucho debido a su rapidez de entrenamiento y calidad.

Para culminar, considero que el desarrollo de videojuegos junto con técnicas de IA nos va a seguir sorprendiendo debido a la velocidad en la que van avanzando estos modelos. Y, personalmente el desarrollo de este proyecto me ha gustado mucho, a pesar del tiempo que ha tomado, por lo que estoy segura que seguiré testeando otras variantes de NeRF o otros modelos por mi cuenta a partir de ahora.

Capítulo 6

Summary and Conclusions

Once the project is completed, it can be confirmed that both the main objective and the specific objectives were achieved. I think it is mainly due to the amount of documentation available for NeRF and all its variants. development with NeRF and in general different models that infer content with a couple of images or descriptions, as is the case with DALL·E 2 [2], are currently on the rise, many developers talk about it daily and the related content is awesome, so what the development process has become more enjoyable and interesting. Also, I consider that was able to establish knowledge regarding the use of more complex AI models and, improve my development skills with Unity3D and C# due to the obstacles that I had to go through and solve, which were different from previous experiences had with the graphics engine for the Intelligent Interfaces subject.

However, despite this, the expectations I had regarding the use of my datasets were high and the results have not been very favorable, so it may be a point to contemplate in the future, either because of the number of images used, the number of defined epochs, the resolution of the images or even having a computer with a GPU with the necessary characteristics to support these training and obtain best results. So I would also like to emphasize that due to the previous point and the characteristics of my current computer, I consider that the development of this project slowed down a bit, sometimes I could spend hours waiting for the connection via sockets would respond or even the computer would stop working and I would have to start from zero, so the process was not very agile and a lot of patience was needed. Then, I consider that the following scenarios could be proposed as future lines:

- Finding the necessary parameters to obtain good results in both NeRF like Tiny NeRF for own datasets.
- Test the implementation with other variants like NeRF in the Wild for change of lighting in a video game or Instant NeRF that promises a lot due to its Speed of training and quality.

To conclude, I consider that the development of videogames together with AI techniquesIt will continue to surprise due to the speed at which these models are advancing. And also, I liked the development of this project a lot, despite the time it took, so I'm sure I'll keep testing other variants of NeRF or ther models on my own from now on.

Capítulo 7

Presupuesto

Para poder estimar el valor del desarrollo de este Trabajo de Fin de Grado, se han dividido las tareas en los bloques más importantes y estimado el valor de cada bloque a partir de un precio por hora fijo en relación a las horas necesitadas. El presupuesto estimado puede ser consultado a continuación. Ver Tabla 7.1

Concepto	Coste por hora	Horas	Coste total
Estudio de modelos de IA	20€	20	400€
Estudio de NeRF	20€	30	600€
Aplicación de Tiny NeRF	20€	60	1.200€
Conexión del servidor	20€	70	1.400€
Escenario en Unity	20€	60	1.200€
Creación de datasets	20€	10	200€
Estudio del requerimiento computacional	20€	20	400€
Documentación del proyecto	20€	60	1.200€
	Total	**330**	**6.600€**

Tabla 7.1: Presupuesto del proyecto

Apéndice A

Repositorios y datasets disponibles

A.1. Datasets naturales y sintéticos

https://drive.google.com/drive/folders/128yBriW1IG_3NJ5Rp7APSTZsJqdJdfc1?usp=sharing

A.2. Repositorio de Unity3D-Python-Communication

https://github.com/off99555/Unity3D-Python-Communication

A.3. Repositorio asociado a este TFG

https://github.com/vanessavvp/TFG-nerf-implementation

A.4. Repositorio de Local Light Field Fusion

https://github.com/fyusion/llff

A.5. NeRF para datasets propios. Resultados en vídeo

https://drive.google.com/drive/folders/1Emk3U0UIPg6RLqqRMFdMx2LDcYZvLxC2?usp=sharing

Bibliografía

[1] Descubre qué es el renderizado 3d, imagen y vídeo. 2018. URL: https://3dalia.com/que-es-el-renderizado-3d/.

[2] Dalle 2 by openai, 2022. URL: https://openai.com/dall-e-2/.

[3] nerf: Code release for nerf (neural radiance fields), 2022. URL: https://github.com/bmild/nerf.

[4] Vision Graphics Seminar at MIT. Jon barron - understanding and extending neural radiance fields, 2021. URL: https://github.com/bmild/nerf.

[5] Edier Bernal Rozo, Renso Cardona Montoya, and Jaime Páez. Videojuegos: Avances tecnológicos en aplicación de física e inteligencia artificial. *Letras ConCiencia TecnoLógica*, pages 61–78, ago. 2018. URL: https://revistas.itc.edu.co/index.php/letras/article/view/114, doi:10.55411/26652544.114.

[6] DKV BY. Simple garage, 2021. URL: https://assetstore.unity.com/packages/3d/props/interior/simple-garage-197251.

[7] Wikipedia contributors. What are 2.5d games? how they differ from 2d and 3d games. 2021. URL: https://www.makeuseof.com/what-are-2-5d-games-2d-3d/.

[8] Wikipedia contributors. Bounding volume. 2022. URL: https://en.wikipedia.org/w/index.php?title=Bounding_volume&oldid=1064206790.

[9] Wikipedia contributors. Rectifier (neural networks). 2022. URL: https://en.wikipedia.org/w/index.php?title=Rectifier_(neural_networks)&oldid=1101096036.

[10] Wikipedia contributors. Trazado de rayos. 2022. URL: https://es.wikipedia.org/wiki/Trazado_de_rayos.

[11] Wikipedia contributors. Volume ray casting. 2022. URL: https://en.wikipedia.org/w/index.php?title=Volume_ray_casting&oldid=1097049152.

[12] Ritwik; Gosthipaty, Aritra Roy; Raha. Computer graphics and deep learning with nerf using tensorflow and keras: Part 1. 2021. URL: https://www.pyimagesearch.com/2021/11/10/computer-graphics-and-deep-learning-with-nerf-using-tensorflow-and-keras-part-

[13] Ritwik; Gosthipaty, Aritra Roy; Raha. Computer graphics and deep learning with nerf using tensorflow and keras: Part 2. 2021. URL: https://www.pyimagesearch.com/2021/11/17/computer-graphics-and-deep-learning-with-nerf-using-tensorflow-and-keras-part-

[14] Ritwik; Gosthipaty, Aritra Roy; Raha. Computer graphics and deep learning with nerf using tensorflow and keras: Part 3. 2021. URL: https://www.pyimagesearch.com/2021/11/24/computer-graphics-and-deep-learning-with-nerf-using-tensorflow-and-keras-part-

[15] Donald P Greenberg Hank Weghorst, Gary Hooper. Improved computational methods for ray tracing. *ACM transactions on graphics*, 3:52–69, 1984. URL: https://dl.acm.org/doi/pdf/10.1145/357332.357335, doi:10.1145/357332.357335.

[16] Zekun Hao, Arun Mallya, Serge Belongie, and Ming-Yu Liu. GANcraft: Unsupervised 3D Neural Rendering of Minecraft Worlds. In *ICCV*, 2021.

[17] Ricardo Martin-Brualla, Noha Radwan, Mehdi S. M. Sajjadi, Jonathan T. Barron, Alexey Dosovitskiy, and Daniel Duckworth. NeRF in the Wild: Neural Radiance Fields for Unconstrained Photo Collections. In *CVPR*, 2021.

[18] Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz-Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics (TOG)*, 38:1–14, 2019. URL: https://bmild.github.io/llff/, doi:10.1145/3306346.3322980.

[19] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020. URL: https://www.matthewtancik.com/nerf.

[20] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4):102:1–102:15, July 2022. doi:10.1145/3528223.3530127.

[21] David Muñoz. ¿qué es un motor gráfico? 2022. URL: https://es.digitaltrends.com/videojuego/que-es-un-motor-grafico/.

[22] Keunhong Park, Utkarsh Sinha, Jonathan T. Barron, Sofien Bouaziz, Dan B Goldman, Steven M. Seitz, and Ricardo Martin-Brualla. Nerfies: Deformable neural radiance fields. *ICCV*, 2021.

[23] Scratchapixel. An overview of the ray-tracing rendering technique (overview of the ray-tracing rendering technique). 2014. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview.

[24] Alex Yu, Vickie Ye, Matthew Tancik, and Angjoo Kanazawa. pixelNeRF: Neural radiance fields from one or few images. In *CVPR*, 2021.