Generación automática de entornos de interacción basada en LLMs
LLM-based interactive enviroment automatic generation

Trabajo de Fin de Grado
Curso 2024–2025

Autores

Gonzalo Bertolín Díez
Alfonso García-Carrasco Puerta
David Toral Castiñeira

Director

Carlos León Aznar

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Generación automática de entornos de interacción basada en LLMs
LLM-based interactive enviroment automatic generation

Trabajo de Fin de Grado en Ingeniería Informática

Autores
Gonzalo Bertolín Díez
Alfonso García-Carrasco Puerta
David Toral Castiñeira

Director
Carlos León Aznar

Convocatoria: *Junio 2025*
Calificación Gonzalo: *8,2*
Calificación Alfonso: *8,2*
Calificación David: *8,2*

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

26 de mayo de 2025

Agradecimientos

A mis amigos Jose, Argenta, Rodrigo, Pablo y Gaby que estuvieron ahí en mis múltiples bloqueos durante este trabajo, para apoyarme y darme ánimos. También a mis dos compañeros de este trabajo y amigos David y Gonzalo por hacer este año más llevadero, incluso en las ausencias. Por último, pero no menos importante, a mi familia, por haberme cuidado todo este tiempo y ayudarme a ser quien soy hoy en día. *Alfonso*

A mis amigos por ayudarme a probar el sistema y ser pacientes con él, en especial a Rafa por dejarme un servidor dónde poder ejecutar las LLMs. A mis compañeros Alfonso y David por lo fácil y cómodo que es trabajar con ellos y a Carlos por animarme a hacer un TFG sobre generación procedural, y por último a mi familia. *Gonzalo*

Le agradezco a todas las personas que me han ayudado a avanzar durante toda la carrera, desde personal docente a los compañeros de curso. Un recordatorio especial para Yoha un amigo que ya no está entre nosotros. Y en especial a mi madre que es la persona a la que le debo todo en esta vida. *David*

Resumen

Generación automática de entornos de interacción basada en LLMs

El uso de los LLM o Modelos Grandes de Lenguaje en la actualidad se ha vuelto de lo más cotidiano, desde la redacción y clasificación de textos, generación de código o respuestas respecto a una base de conocimientos, son algunas de las funcionalidades que hacen que sean tan populares en una vasta selección de campos que abarcan desde la medicina hasta los videojuegos, donde este último es el campo en el cual se centrará este proyecto. Se ha experimentado bastante con su uso en los distintos aspectos que lo componen. Por ejemplo, en la generación de entornos o narrativas dinámicas, en la interacción con Personajes No Jugadores o PNJ o en los *Game Master*.

El objetivo de este estudio es crear un generador de niveles mediante el uso de un LLM con el propósito de ayudar a los diseñadores a crear niveles de una manera rápida y efectiva. Para ello, el usuario insertará una breve descripción del nivel que querría generar, para tras ser procesado por el LLM, recibir un nivel o mundo fiel a la descripción dada. Una vez generado, el usuario podrá volver a pedir a la aplicación que genere cambios dentro del mismo nivel, o si no está conforme, pedir uno nuevo.

Se ha evaluado el sistema mediante pruebas con distintos usuarios y los resultados demostraron que el estudio es útil para el propósito que fue creado pero, aun así, se requiere más investigación para llegar a resultados conclusivos.

Palabras clave

Modelos grandes de lenguaje (LLM), inteligencia artificial (IA), niveles, Unreal Engine 5 (UE5), texturas, diseño

Abstract

LLM-based interactive enviroment automatic generation

The use of LLMs or Large Language Models has become commonplace today, from text writing and classification to code generation and responses to a knowledge base. These are just some of the features that make them so popular in a wide range of fields, from medicine to video games, the latter being the field on which this project will focus. There has been considerable experimentation with their use in various aspects of this field. For example, in the generation of dynamic environments or narratives, in interaction with non-player characters (NPCs) or in game masters.

The objective of this study is to create a level generator using an LLM to help designers create levels quickly and effectively. To do this, the user will insert a brief description of the level they would like to generate, which will then be processed by the LLM to receive a level or world that is faithful to the description given. Once generated, the user can ask the application to generate changes within the same level or, if they are not satisfied, request a new one.

The system has been evaluated through testing with different users, and the results showed that the study is useful for the purpose for which it was created. However, further research is needed to reach conclusive results.

Keywords

Large language models (LLM), artificial intelligence (AI), levels, Unreal Engine 5 (UE5), textures, design

Índice

1. Introducción **1**
1.1. Motivación 2
1.2. Hipótesis 2
1.3. Objetivos 3
1.4. Plan de trabajo 4
1.5. Estructura del resto del documento 6

2. Estado de la Cuestión **7**
2.1. Inteligencia Artificial 7
2.2. Modelos Grandes de Lenguaje (LLM) 9
2.3. Motores de Videojuegos 10
2.4. Generación Procedimental 11
2.5. Sistemas que usan LLMs para generar contenido en videojuegos . . . 14

3. Diseño y Arquitectura de un sistema de generación procedimental en base a peticiones del usuario con LLM **19**
3.1. Diseño 19
3.2. Arquitectura 19

4. Diseño del motor de generación procedimental de niveles basado en LLM **25**
4.1. Prototipo del sistema con LLM en local 25
4.2. Prototipo del sistema usando llamafiles 28

5. Desarrollo del sistema de generación procedimental en base a peticiones del usuario con LLM **31**
5.1. Implementación del entorno en Unreal Engine 5 31
5.2. Sección de texturas 34

6. Conclusiones del análisis de métricas aplicado a la salida del LLM **41**
6.1. Objetivos 42
6.2. Casos de prueba 42

6.3. Método de obtención de datos 43
6.4. Método de volcado de datos 43
6.5. Resumen del proceso 43
6.6. Llama 3.2 (8gb) 45
6.6.1. Resultados y valores obtenidos al procesar un texto corto y concreto 45
6.6.2. Resultados y valores obtenidos al procesar un texto largo y concreto 47
6.6.3. Resultados y valores obtenidos al procesar un texto imaginativo 50
6.7. Phi 2 orange (3-4 gb) 53
6.7.1. Resultados y valores obtenidos al procesar un texto corto y concreto 53
6.7.2. Resultados y valores obtenidos al procesar un texto largo y concreto 56
6.7.3. Resultados y valores obtenidos al procesar un texto imaginativo 59
6.8. Llama 3.2 (1gb) 62
6.8.1. Resultados y valores obtenidos al procesar un texto corto y concreto 62
6.8.2. Resultados y valores obtenidos al procesar un texto largo y concreto 65
6.8.3. Resultados y valores obtenidos al procesar un texto imaginativo 68
6.9. Conclusión final 70

7. Evaluación con usuarios del sistema de generación 71
7.1. Descripción de la evaluación 71
7.2. Evaluación cualitativa del proceso de diseño de usuarios 73
7.2.1. Usuario 1 73
7.2.2. Usuario 2 75
7.2.3. Usuario 3 76
7.2.4. Usuario 4 78
7.2.5. Usuario 5 80
7.2.6. Usuario 6 82
7.2.7. Usuario 7 83
7.2.8. Usuario 8 85
7.2.9. Usuario 9 86
7.2.10. Usuario 10 88
7.2.11. Sugerencias hechas por los encuestados en el cuestionario . . . 89
7.3. Evaluación cuantitativa del proceso de diseño de usuarios 90
7.3.1. Datos del cuestionario 90
7.3.2. Otras métricas 94
7.3.3. Conclusiones finales del experimento 95

8. Discusión 97

9. Conclusiones y Trabajo Futuro **99**
9.1. Líneas de trabajo futuro 100
9.1.1. Renderizado en modo editor 100
9.1.2. Base de texturas o *sprites* escalables 101
9.1.3. Entrenamiento de un LLM para que se especialice en el área de generación de niveles 102
9.1.4. Cargar niveles pre-hechos y posibilidad de guardar niveles generados por el LLM que pueden estar o no modificados por el usuario. 102
9.1.5. Poder configurar los valores de los parámetros que va a usar el LLM para la generación y poder visualizar con que parámetros exactos ha generado el nivel pedido en un momento dado . . . 104
9.1.6. Poder pedir al LLM que escale o rote ciertos objectos elegidos por el usuario 105

Introduction **107**

Conclusions and Future Work **113**

Contribuciones Personales **121**

Bibliografía **127**

A. Manual de uso **129**

Índice de figuras

1.1. Diagrama de Gantt con la planificación temporal del proyecto de los meses de septiembre a diciembre 4
1.2. Diagrama de Gantt con la planificación temporal del proyecto de los meses de enero a marzo 5
1.3. Diagrama de Gantt con la planificación temporal del proyecto de los meses de abril a junio 5

2.1. Estructura de un paso t de una RNN tradicional [2] 8
2.2. Estructura de un paso t en el modelo LSTM [2] 9
2.3. Proceso del algoritmo Diamante-Cuadrado. 12
2.4. Ruido de Perlin. 13

3.1. Arquitectura general del sistema 20
3.2. *Widget* para escribir peticiones 20
3.3. Diagrama de flujo de la interconexión de los scripts de Python 23

4.1. Un ejemplo de JSON bien estructurado 26

5.1. Figura que muestra el flow de *event graph* 32
5.2. Figura que muestra las diferencias entre *event graph* y la funcion contenido recursivo 33
5.3. Figura que muestra el flujo para asignar una textura en la funcion *spawn mesh* 34
5.4. Ejemplo de una malla estática con texturas 35
5.5. Ejemplo de una malla estática sin texturas 35
5.6. Un edificio del paquete *LowPolyCity* 36
5.7. Un coche del paquete *LowPolyCity* 37
5.8. Un camión del paquete *LowPolyCity* 37
5.9. Una parada de autobus del paquete *LowPolyCity* 38
5.10. Un árbol del paquete *LowPolyCity* 38
5.11. Un semáforo del paquete *LowPolyCity* 39

6.1. Gráfica obtenida de las 6.1 y 6.2 46
6.2. Gráfica obtenida de las 6.3 y 6.4 49

6.3. Gráfica obtenida de las 6.5 y 6.6 52
6.4. Gráfica obtenida de las 6.7 y 6.8 55
6.5. Gráfica obtenida de las 6.9 y 6.10 58
6.6. Gráfica obtenida de 6.11 y 6.12 61
6.7. Gráfica obtenida de las 6.13 y 6.14 64
6.8. Gráfica obtenida de 6.15 y 6.16 67
6.9. Gráfica obtenida de 6.17 y 6.18 70

7.1. Entorno desarrollado manualmente por el usuario 1 73
7.2. Entorno desarrollado usando la herramienta por el usuario 1 74
7.3. Entorno desarrollado manualmente por el usuario 2 75
7.4. Entorno desarrollado usando la herramienta por el usuario 2 76
7.5. Entorno desarrollado manualmente por el usuario 3 77
7.6. Entorno desarrollado usando la herramienta por el usuario 3 77
7.7. Entorno desarrollado manualmente por el usuario 4 78
7.8. Entorno desarrollado usando la herramienta por el usuario 4 79
7.9. Entorno desarrollado manualmente por el usuario 5 80
7.10. Entorno desarrollado usando la herramienta por el usuario 5 81
7.11. Entorno desarrollado manualmente por el usuario 6 82
7.12. Entorno desarrollado usando la herramienta por el usuario 6 83
7.13. Entorno desarrollado manualmente por el usuario 7 84
7.14. Entorno desarrollado usando la herramienta por el usuario 7 84
7.15. Entorno desarrollado manualmente por el usuario 8 85
7.16. Entorno desarrollado usando la herramienta por el usuario 8 86
7.17. Entorno desarrollado manualmente por el usuario 9 87
7.18. Entorno desarrollado usando la herramienta por el usuario 9 87
7.19. Entorno desarrollado manualmente por el usuario 10 88
7.20. Entorno desarrollado usando la herramienta por el usuario 10 89
7.21. Respuestas sobre el tiempo ahorrado 91
7.22. Respuestas sobre el esfuerzo mental ahorrado 91
7.23. Respuestas sobre la facilidad de uso de la herramienta 92
7.24. Respuestas sobre si la herramienta ha sido útil para centrarse más en el concepto 92
7.25. Respuestas sobre si la herramienta ha sido precisa respecto a la descripción 93
7.26. Respuestas sobre si el usuario usaría la herramienta en un desarrollo real 93
7.27. Respuestas sobre la valoración general de la herramienta 94
7.28. Tiempo de uso por usuario entre generación manual y por LLM . . . 94
7.29. Tiempo medio de uso entre la generación manual y por LLM 95

9.1. Gantt chart with the project schedule for September to December . . 109
9.2. Gantt chart with the project schedule for January to March 110
9.3. Gantt chart with the project schedule for April to June 110

Índice de tablas

1.1. Plan de trabajo 4

6.1. Llama 3.2 corto (Parte 1) 45
6.2. Llama 3.2 corto (Parte 2 - Valores medios) 45
6.3. Llama 3.2 largo - Tabla 1 (Parte 1) 47
6.4. Llama 3.2 largo - Tabla 1 (Parte 2 - Medias) 48
6.5. Llama 3.2 imaginativo (Parte 1) 50
6.6. Llama 3.2 imaginativo (Parte 2 - Valores medios) 51
6.7. Phi 2 corto (Parte 1 - Valores individuales) 53
6.8. Phi 2 corto (Parte 2 - Valores medios) 54
6.9. Phi 2 largo (Parte 1 - Valores individuales) 56
6.10. Phi 2 largo (Parte 2 - Valores medios) 57
6.11. Phi 2 imaginativo (Parte 1 - Valores individuales) 59
6.12. Phi 2 imaginativo (Parte 2 - Valores medios) 60
6.13. Llamalite corto (Parte 1 - Valores individuales) 62
6.14. Llamalite corto (Parte 2 - Valores medios) 63
6.15. Resultados por iteración (Parte 1 - Valores individuales) 65
6.16. Resultados por iteración (Parte 2 - Valores medios) 66
6.17. Llamalite imaginativo (Parte 1 - Valores individuales) 68
6.18. Llamalite imaginativo (Parte 2 - Valores medios) 69

9.1. Work plan 110

Capítulo 1

Introducción

Según ha ido evolucionando y creciendo el mundo de los videojuegos, estos se han hecho más complejos. Los primeros juegos eran juegos arcade, cuyo objetivo principal era que los jugadores completaran más y más pantallas, mientras que a día de hoy los videojuegos pueden centrarse en transmitir un mensaje con la narración de una historia y la profundidad de sus personajes, la inmersión del jugador conseguida a partir del apartado gráfico y musical, la jugabilidad y fluidez con el diseño de mapas, niveles y mecánicas.

Cada parte que compone el videojuego requiere de un equipo de profesionales que se centran en diferentes ámbitos, algunas de esas múltiples disciplinas no se originaron a partir de los videojuegos, como el arte, la narrativa o la música. Pero el diseño y creación de niveles y mapas sí tuvieron como origen estos mismos. Y al igual que en las otras disciplinas, estas dos últimas conllevan un proceso largo y complejo a la par que creativo, que requiere crear y relacionar conceptos de tal modo que tras numerosas revisiones quede todo cohesionado, lo cual puede conllevar una cuantiosa cantidad de tiempo y recursos invertidos.

Esto ha llevado a gran cantidad de juegos, tanto pequeñas como grandes producciones, hoy en día considerados grandes éxitos, a tener que retrasar mucho su esperada salida, como por ejemplo, Horizon Zero Dawn (*Guerrilla Games*, 2017) que desde que se anunció por primera vez en el E3 de 2015, el público tuvo que esperar hasta 2017 para hacerse con él , Kingdom Hearts III (*Square Enix*, 2018) que fue anunciado en 2013 y hasta 2018 no salió a la venta o Hollow Knight (*Team Cherry*, 2017) el cual fue financiado por un kickstarter que finalizó en 2014 y no apareció en el mercado hasta 2017.

Nuestro trabajo pretende conseguir una herramienta que, aprovechando la capacidad de cómputo de los LLM, como por ejemplo el posiblemente más conocido ChatGPT con GPT-4, ayude a los usuarios en la generación de mapas y niveles para así poder aliviar la carga en la creación de los mapas en los videojuegos, aunque también podría ser escalable a otros campos en los cuales la generación o replicación de un mapa sea útil.

1.1. Motivación

El diseño de niveles, también conocido como mapeado, representa uno de los pilares fundamentales en la creación de un videojuego. Este proceso no solo impacta directamente en la jugabilidad y en la experiencia del usuario, sino que también influye en aspectos narrativos, de exploración y de inmersión dentro del universo del juego. Debido a su importancia, el diseño de niveles suele requerir una considerable inversión de tiempo, esfuerzo humano y recursos económicos, lo cual puede ser un obstáculo especialmente significativo para estudios pequeños o desarrolladores independientes.

En este contexto, el avance de modelos de lenguaje de gran escala (LLMs, por sus siglas en inglés) ofrece una oportunidad innovadora para reducir parte de esta carga. Gracias a su capacidad de generación de contenido, aprendizaje autónomo y comprensión del contexto, un LLM puede ser empleado como asistente en la generación de mapas o niveles de forma más eficiente. La propuesta consiste en integrar estas capacidades en un entorno visual intuitivo, que permita al usuario interactuar con el modelo de manera directa y comprensible, sin necesidad de profundos conocimientos técnicos.

Uno de los objetivos principales es acelerar las fases de prueba y error, típicas del diseño de niveles, así como facilitar la creación de espacios coherentes y estéticos, aprovechando la habilidad del modelo para detectar patrones y generar variaciones significativas. De esta manera, el diseñador puede centrarse en la parte creativa y conceptual del desarrollo, mientras que el LLM asiste con tareas repetitivas, estructurales o de inspiración.

Además, este enfoque también busca igualar el acceso al diseño de videojuegos. Al crear una herramienta que sea accesible, manejable y con una curva de aprendizaje moderada, se abre la puerta a nuevos perfiles de usuarios interesados en estos campos: personas con ideas creativas, pero que tradicionalmente se han visto limitadas por la falta de conocimientos técnicos o por barreras económicas. Así, el proyecto no solo persigue una mejora en la eficiencia del desarrollo, sino también fomentar la inclusión y la participación de nuevos talentos en el ámbito del diseño de videojuegos.

En resumen, esta propuesta se sitúa en la intersección entre la inteligencia artificial y la creatividad humana, con el objetivo de hacer del diseño de niveles un proceso más accesible, ágil e inspirador, al servicio tanto de desarrolladores profesionales como de entusiastas con recursos limitados.

1.2. Hipótesis

Con lo expuesto anteriormente y con la potente tecnología actual de la que disponen los LLM, se cree que creando un programa sencillo e intuitivo para el diseñador, con el cual pueda generar un nivel o escenario modificable de forma automática tan solo con una pequeña descripción, de manera que ahorre recursos en apartados básicos de la generación, explotando la potencia de los LLM en estos campos, de

modo que los grandes desarrolladores puedan destinar el resto de recursos a otros apartados y en caso de los pequeños creadores, pueden dar paso a su creatividad sin preocuparse tanto del presupuesto. Con esto nos podemos plantear la siguiente hipótesis:

> *Hipótesis 1: Explotando la potencia de los LLM, usando sus capacidades, se cree que es posible que estos generen niveles jugables, estéticos y creativos, de una manera sencilla y barata.*

1.3. Objetivos

El objetivo general de este trabajo es crear una herramienta que ayude a los desarrolladores y diseñadores de videojuegos a la hora de crear niveles y mapas en Unreal Engine 5. A continuación, se expondrá más detalladamente una lista con los objetivos específicos de este trabajo.

1. Diseño conceptual de herramienta de uso de LLMs para generación de niveles

   Dado un LLM cualquiera, lee una descripción de la idea a representar introducida por el usuario desde un chat en el interior del propio Unreal, este mensaje después es modificado con datos de formato para asegurarnos que la salida de la LLM es la adecuada, la cual se conecta nuevamente con Unreal Engine 5, el encargado de leer el JSON y generar el nivel, accediendo al modo editor desde el cual este nivel puede ser modificado.

2. Determinar el formato de salida correcto que tiene que dar el LLM para el JSON

   Dado un LLM cualquiera, en el script de Python es posible conseguir que la salida proporcionada, se ajuste al formato JSON necesario para parsear la respuesta en Unreal Engine 5.

3. Establecer la comunicación entre el LLM y la aplicación

   Dado un LLM cualquiera, conseguir una comunicación bidireccional entre el LLM y las distintas partes de la aplicación para poder recibir respuestas del LLM y mandar peticiones de una manera fluida, donde el proceso de recibir respuestas y mandar peticiones se puede hacer indefinidamente.

4. Gestionar la salida del LLM y generar el nivel acorde a esta salida

   Dado un JSON bien formado según el objetivo 2, con ese JSON es posible generar un nivel dado un proceso que recibe un archivo como el JSON mencionado anteriormente y genera un nivel lo más fiel posible al archivo, en caso de no poder aun así generará un nivel.

5. Realización de pruebas con usuarios

   Establecer unos experimentos para analizar la efectividad de la aplicación con respecto al diseño a mano, estas pruebas deben incluir las dos versiones de los diseños de los niveles, la manual y la que se hizo con la aplicación, además de incluir un breve contexto del usuario al que se le realizó el experimento.

1.4. Plan de trabajo

En esta se va a describir el plan de trabajo a seguir para la consecución de los objetivos descritos en el apartado anterior.

Tabla 1.1: Plan de trabajo

Mes	Objetivos
Septiembre 2024	Planificar el proyecto, saber que tecnologías se usarán, detallar la idea
Octubre 2024	Aprender a usar LLMs en local, primer prototipo del módulo de Python que sacaba respuestas
Noviembre 2024	Pruebas con el prototipo para ver los tipos de respuesta, Inverstigar técnologías de Unreal Engine 5
Diciembre 2024	Refinar el primer prototipo del módulo de Python, aprendizaje de las tecnologías de Unreal Engine 5
Enero 2025	Terminar las bases del módulo de Python, primer prototipo de Unreal Engine 5
Febrero 2025	Establecer una conexión entre ambas partes, refinar prototipo de Unreal Engine 5
Marzo 2025	Refinar la conexión entre las partes, añadir texturas al proyecto, empezar redacción de la memoria
Abril 2025	Finalización del proyecto de generación de niveles, continuar redacción
Mayo 2025	Obtención de resultados con la evaluación con usuarios y finalización de la redacción de la memoria
Junio 2025	Presentación del proyecto

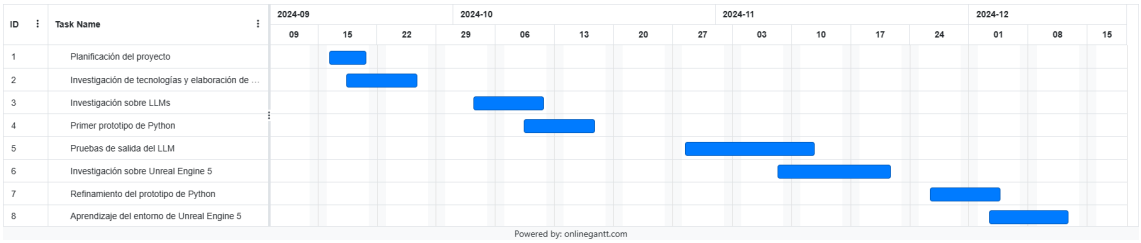

Figura 1.1: Diagrama de Gantt con la planificación temporal del proyecto de los meses de septiembre a diciembre

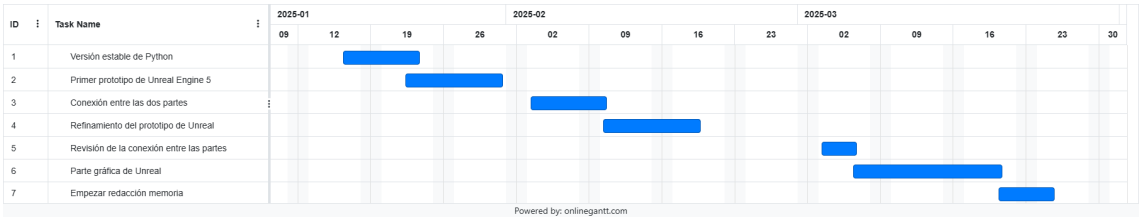

Figura 1.2: Diagrama de Gantt con la planificación temporal del proyecto de los meses de enero a marzo

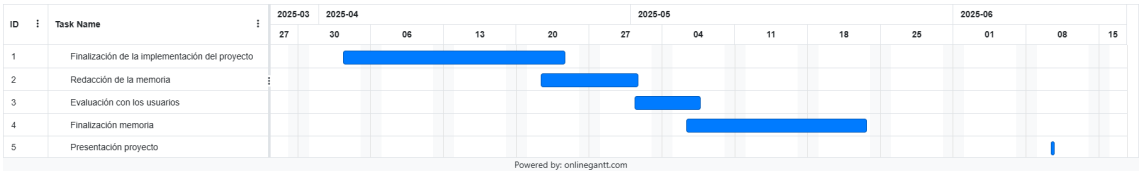

Figura 1.3: Diagrama de Gantt con la planificación temporal del proyecto de los meses de abril a junio

1.5. Estructura del resto del documento

En esta sección, se ofrece un esquema del resto de contenidos que hay en el documento, como sus capítulos y secciones de cada uno. Así como el orden en el que están expuestos los contenidos del documento.

En el capítulo 2 del documento se expone un análisis del Estado del Arte donde se exploran las distintas tecnologías relacionadas con este proyecto, además se exponen características de estas tecnologías que no se abordan de manera explícita en este proyecto. También se realiza un estudio sobre trabajos relacionados con nuestro proyecto para abordar distintos puntos de vista sobre un problema común.

En el capítulo 3 del proyecto se expone la arquitectura interna que tiene este proyecto. Donde se definen los principales módulos del proyecto y como están conectados entre sí.

En el capítulo 4 se habla sobre el módulo de Python y del LLM. Que se describen las clases usadas, las funciones de las clases y las normas usadas por el LLM. Exponiendo el proceso de cómo fue evolucionando desde el comienzo hasta la finalización.

En el capítulo 5 se divide en dos secciones distintas: una sobre el desarrollo en Unreal Engine 5 y la otra sobre las texturas utilizadas en el proyecto. Donde se profundiza sobre qué tipo de proyecto de Unreal Engine 5 se ha usado y cuáles son los contenidos relevantes dentro de él. También se exponen las texturas utilizadas en el proyecto y cuáles han sido las razones para elegir esas texturas.

En el capítulo 6 se realiza un análisis sobre las métricas usadas para el estudio de las respuestas del LLM, dividido en varias secciones, donde se introduce el tema, se expone cómo se han obtenido los datos, cómo se ha realizado el proceso y luego se exponen los resultados y se analizan.

En el capítulo 7 se expone el proceso de evaluación a los usuarios, donde se habla qué acciones van a realizar los usuarios para la evaluación y qué se va a evaluar. Una vez descrito esto. Se exponen los resultados obtenidos con cada usuario. Así como alguna observación y finalmente se exponen los resultados generales.

En el capítulo 8 se expone una discusión sobre las ventajas e inconvenientes de este proyecto.

En el capítulo 9 se habla sobre las conclusiones y resultados generales de todo el proyecto y, después en otra sección se exponen varias líneas de trabajo para realizar en el futuro que no se han podido hacer en este plazo debido a distintos problemas.

Finalmente, en el capítulo 10 se encuentra el manual de uso para este proyecto.

Capítulo 2

Estado de la Cuestión

En este apartado expondremos el estado del arte sobre el diseño y generación de niveles en los videojuegos, abordando sus diferentes tendencias y cómo han ido evolucionando a lo largo de su historia. Además de esto, examinaremos brevemente el estado de la inteligencia artificial y los Modelos Grandes de Lenguaje (LLM), su origen, su uso y su estado actual. Siendo más concretos, hablaremos de los Llamafiles; qué son, en qué se basan y también su estado actual. Respecto a otras tecnologías, se explicará el concepto de motor de videojuego y también de generación procedimental. Finalizando, se hará un análisis de trabajos cuyos temas a tratar tenían como eje principal el LLM y sus

2.1. Inteligencia Artificial

Podemos definir a la inteligencia artificial en el ámbito de la computación, más adelante referida en este documento como IA, como la capacidad que tiene un sistema para aprender de la información dada y con ese aprendizaje pueda tomar decisiones o adaptarse a nuevas situaciones como lo haría un humano aprendiendo un concepto nuevo.

La inteligencia artificial tiene numerosas disciplinas, como son el aprendizaje automático, el aprendizaje por refuerzo, el aprendizaje profundo o el aprendizaje supervisado, el modelo transformador, el modelo de memoria larga a corto plazo, entre otros. El aprendizaje automático parte de la premisa de que, dado un conjunto de datos de prueba, el sistema puede producir resultados basados en lo aprendido con dichos datos.

El aprendizaje profundo o *deep learning* en inglés, se basa en el uso de las redes neuronales profundas. Las redes neuronales profundas se diferencian de las redes neuronales normales por el número de capas que tienen. Entre la capa de entrada y la de salida se sitúan las capas ocultas. Aquí se encuentra la diferencia: las redes neuronales básicas pueden tener una o dos capas ocultas, mientras que una red de aprendizaje profundo puede tener decenas o incluso cientos de capas. En consecuencia, estas redes, responsables de los algoritmos de aprendizaje profundo, son capaces de clasificar datos más complejos. Como se ha explicado anteriormente, el aprendi-

zaje profundo se produce mediante el uso de redes neuronales, que se organizan en capas para reconocer patrones complejos de datos Rouhiainen (2018).

Uno de los ejemplos más claros se encuentra en las traducciones de Facebook, que reveló recientemente que gracias al aprendizaje profundo, había podido realizar unas 4.500 millones de traducciones diarias Rouhiainen (2018) [1]

El modelo de memoria larga a corto plazo, *long short-term memory* (LSTM) en inglés. Es un modelo específico de las redes neuronales recurrentes (RNN). Este modelo fue descrito por Sepp Hochreiter y Jürgen Schmidhuber en el artículo de *long short-term memory* Hochreiter y Schmidhuber (2016). Este modelo busca mejorar las limitaciones de las RNN para recordar patrones en secuencias largas, por la estructura de esta donde dado un dato actual la RNN actualiza el estado oculto y da una predicción, con esta estructura, dada una secuencia con muchos pasos acabaría olvidando el principio de la secuencia.[2]

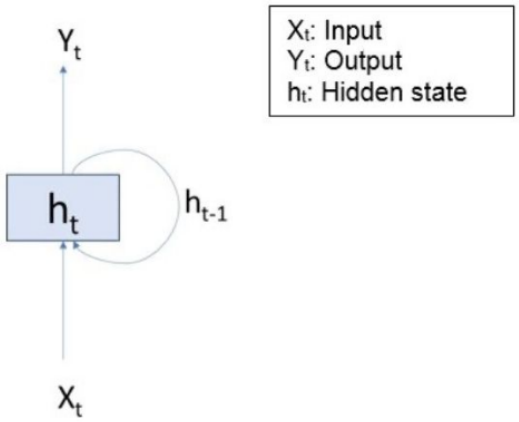

Figura 2.1: Estructura de un paso t de una RNN tradicional [2]

En el modelo LSTM las redes son capaces de "recordar" un dato relevante de la secuencia por varios instantes de tiempo, por tanto puede tener memoria a corto plazo, como las RNN, y también a largo plazo. Para poder conseguir que la red sea capaz de recordar, la celda LSTM contiene una parte llamada "celda de estado", una entrada, una salida, una compuerta para olvidar (*forget gate*), una compuerta para actualizar (*update gate*) y una compuerta de salida (*output gate*), con esta estructura el modelo LSTM es menos sensible a los cambios en el tiempo que las RNN. [2]

Cada compuerta es una red neuronal que funciona como válvulas donde abren el paso a la información o la cierran. Con esto en mente, la red LSTM puede elegir, según los valores que tenga en ese momento, si olvidar o aprender la información que lleva en ese instante, pudiendo así almacenar más datos, gracias a las compuertas para olvidar y actualizar y a la celda de estado.

Existen modelos masivos como GPT-4, Gemini o Mistral que usan estas técnicas a una escala muy grande. en concreto el modelo transformador, con la función de

[1] Huan Muguel Pino, «Transitioning entirely to neural machine translation», Facebook, 3 de agosto de 2017, <https://code.facebook. com/posts/289921871474277/transitioning-entirely-to-neural-machinetranslation>

[2] *Fuente:* What is Long short-term memory? https://www.mathworks.com/discovery/lstm.html

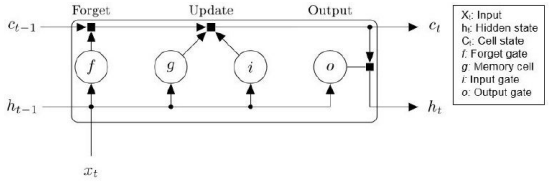

Figura 2.2: Estructura de un paso t en el modelo LSTM [2]

ser usadas en distintos campos tales como generar conversaciones, en el sector de la salud, son unos ejemplos de los muchos campos en los cuales se usan. Se puede intuir que para producir ese tipo de resultado, estos modelos han sido entrenados con una cantidad de datos astronómica.

2.2. Modelos Grandes de Lenguaje (LLM)

Los modelos grandes de lenguajes o más adelante referidos por sus siglas en inglés LLM, existen desde hace varias décadas, pero debido a la falta de potencia en la tecnología de la época, estaban muy limitados y se utilizaban muy poco, sobre todo en usos académicos. En los últimos años han ido madurando hasta alcanzar niveles de funcionalidad sorprendentes en modelos grandes y aplicaciones de IA generativa, llegando a revolucionar la inteligencia artificial y la sociedad a día de hoy.

El punto de inflexión en la evolución de los LLM fue en 2017 con el uso del modelo *Transformer*. Este modelo mejoró significativamente la comprensión de las relaciones de palabras dentro de oraciones, permitiendo generar así frases gramaticalmente correctas y semánticamente coherentes.

En la actualidad, los LLM más populares son: GPT-4, GPT-3.5 de OpenAI, Mistral, que comparte nombre con la empresa, Gemini de Google o LLaMa de Meta, entre otros. En un nivel más pequeño o básico, los LLM se basan en aprendizaje automático, como se ha descrito previamente; el aprendizaje automático es un subconjunto de la IA y consiste en "alimentar" un programa con un gran conjunto de datos para así entrenarlo para buscar relaciones entre los mismos. Por ejemplo, una frase donde las letras con más repeticiones sean la "a" y la "e". A partir de esto, el modelo podría llegar a concluir que esos caracteres son los más probables de aparecer en un texto en español. Esto también se puede aplicar a otros idiomas o aspectos. Aunque con solo una frase no se puede sacar mucha información, si lo extendemos a millones de frases o datos, el modelo será capaz de generar sus propias frases, detectar fallos en las frases introducidas o generar otras respuestas que no tengan que ser en lenguaje natural, pudiendo llegar a crear imágenes o música. (Cloudflare, 2025)

El tipo específico de redes neuronales que usan los LLM se denominan modelos transformadores o *transformers* mencionados previamente. Este modelo fue descrito por primera vez en 2017 en un artículo llamado *Attention is all you need* de Ashish Vaswani, un equipo de Google Brain y un grupo de la Universidad de Toronto Vaswani et al. (2017).

Estos modelos son usados por generadores de texto como ChatGPT para la

predicción, resumen, respuesta a preguntas y un largo etcétera. El elemento principal del modelo transformador es no tener que depender de redes neuronales recurrentes (RNN) o de redes neuronales convolucionales (CNN). Los transformadores pueden procesar secuencias de entrada en paralelo, lo que los hace muy eficientes para los entrenamientos y la inferencia de datos. En el contexto de predicción de texto hay dos innovaciones que estos modelos hacen posibles, la primera es la codificación posicional que consiste en asignar un número único a cada palabra de la secuencia en lugar de mirar cada palabra en el orden que aparece; proporciona información sobre la posición de cada token (palabras o parte de la entrada) en la secuencia. La segunda es la autoatención, donde la atención es el mecanismo que calcula el peso de cada palabra de una oración en relación con todas las demás palabras, gracias a ello, el modelo podrá predecir qué palabras se usarán a continuación en la secuencia, esto ocurre cuando el modelo es entrenado con una gran cantidad de datos.

2.3. Motores de Videojuegos

Los motores de videojuegos nacieron de la necesidad de simplificar el proceso de creación de los mismos, ya que en los inicios, estos se creaban cada uno partiendo de cero, teniendo que implementar todo. Según lo dicho, podemos definir un motor de videojuegos como una biblioteca *software* compuesta por funcionalidades que proporcionan una base que permite realizar las tareas principales y más básicas, como pueden ser:

- Renderizado de gráficos, encargado de hacer todas las operaciones matemáticas como son operaciones con vectores y matrices. Además de la aplicación de texturas y *shaders*.
- Procesamiento de *input*, encargado de recoger y procesar toda la información mandada por el usuario a través de periféricos como el ratón, el teclado o un mando.
- Un motor de físicas, encargado de implementar la simulación de las físicas necesarias para colisiones, saltos, aceleración, velocidad, entre otros datos.
- Procesamiento de audio, encargado de reproducir tanto los sonidos como la música que se haya programado dentro del videojuego.

Uno de los componentes más importantes para un motor de videojuego es la API gráfica que utiliza, algunos motores pueden usar más de una API gráfica. Una API gráfica se puede definir como una colección de comandos que permiten usar a las aplicaciones las opciones hardware de la unidad de procesamiento gráfico, en inglés GPU. Montoya (2017)

Algunos ejemplos de APIs gráficas pueden ser DirectX de Microsoft, OpenGL de Silicon Graphics, WebGL que está basada en OpenGL, entre otros ejemplos.

Otros elementos necesarios para un motor de videojuego son:

- el matemático encargado de hacer todos los cálculos necesarios para el videojuego, por ejemplo en caso de ser 3D operaciones con vectores, matrices o similares.
- el gráfico para los renderizados, las texturas, los *shaders* según la API usada.
- la parte del input, encargada de recoger y procesar toda la información proveniente del jugador a través de los periféricos como pueden ser el ratón, el teclado o el mando.
- el físico encargado de implementar toda la parte de las físicas que puede tener un videojuego.

Otros elementos necesarios para los motores en los videojuegos son los relacionados con el audio, con el manejo de errores y el manejo de escenas. Thorn (2011)

En la última década también hemos presenciado el surgimiento de motores comerciales dedicados al público general, como lo son Unity, Unreal Engine o Godot, que facilitan el desarrollo de videojuegos gracias a su accesibilidad y a la gran cantidad de documentación que hay sobre ellos.

2.4. Generación Procedimental

Podemos definir la generación procedimental, procedural o por procedimientos como un método de creación de contenidos diversos a través de algoritmos, en oposición a la creación manual. Esta generación se puede aplicar a campos como la computación, simulaciones, música, programación y videojuegos, que es donde centraremos nuestra atención en esta sección. Unos ejemplos destacados del uso de la generación procedimental en videojuegos pueden ser los siguientes:

- Empezando por uno de los pioneros que utilizaron esta técnica para generar niveles tenemos a *Rogue*(1980), este juego genera procedimentalmente la mazmorra de la partida además de introducir el concepto de "muerte permanente" ya que una vez muerto el personaje, la partida vuelve a empezar desde cero en una mazmorra nuevamente generada. La repercusión que tuvo este juego fue tal que llegó a crear un género en los videojuegos conocido como *Roguelike*, que se podría traducir como: "Juegos parecidos a *Rogue*".
- *Spore*(2008) es un videojuego en el que no tomas el papel de un personaje en concreto si no que tomas el papel del creador. El objetivo de este juego es crear una especie que sobreviva al entorno o que muera en el intento. El juego destaca por la generación masiva de todo el contenido mediante técnicas procedimentales, no solo los mundos se generan así, también las animaciones de las criaturas mediante muchas partes predefinidas.
- Uno de los juegos que más se juegan en la actualidad es *Minecraft*(2009), un juego del género sandbox creado por "Notch" que más tarde fundó la empresa Mojang. Este juego presenta una infinidad de tareas por hacer desde el principio de una manera sencilla, lo que destaca principalmente es que la generación

de los mundos lo hace de manera procedimental usando el ruido de Perlin y el uso de semillas, cada mundo generado tiene una semilla ligada a él y si tu seleccionas esa semilla ese mundo se va a generar exactamente igual en otra partida. Para crear los mundos al principio usaba mapas de alturas 2D con el ruido de Perlin pero finalmente optó por usar un mapa de alturas en 3D dando más aleatoriedad a la generación de mundos.

- *No man's sky*(2016) es otro juego del género sandbox donde el objetivo es simular un universo generado mediante técnicas procedimentales, desde la distribución de los planetas, la vida que albergan, su geología, etc. Dando lugar a millones y millones de combinaciones posibles.

Existen diversos algoritmos a través de los cuales implementar la generación procedimental:

- **El algoritmo Diamante-Cuadrado**, que partiendo de una matriz cuadrada de tamaño $2^n + 1$ con las cuatro esquinas inicializadas a un valor, sigue los siguientes dos pasos alternativamente hasta que toda la matriz está completa:

  - **El paso del diamante:** Para cada cuadrado en la matriz, se escribe en el centro de dicho cuadrado el valor dado por la media de las cuatro esquinas más un valor aleatorio.
  - **El paso del cuadrado:** Para cada diamante en la matriz, se escribe en el centro de dicho diamante el valor dado por la media de las cuatro esquinas más un valor aleatorio.

  Cada valor aleatorio está multiplicado por una escala constante que disminuye con cada iteración por un factor de 2^{-h}, donde h es un valor entre 0.0 y 1.0 Fournier et al. (1982).

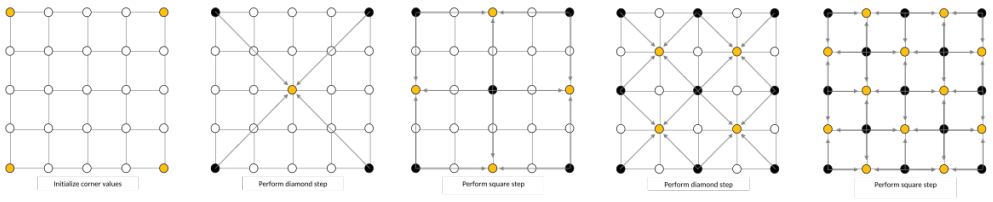

Figura 2.3: Proceso del algoritmo Diamante-Cuadrado.

Fuente: Christopher Ewin. Licencia CC BY-SA 4.0.

- **Algoritmo del ruido de Perlin**, propuesto por Perlin (1985), genera un patrón de ruido continuo y natural, su implementación consta de tres pasos: definir una rejilla de vectores de gradiente aleatorios, calcular el producto escalar para cada punto de la rejilla entre los vectores de gradiente y su desplazamiento y por último se realiza una interpolación entre los valores.

  - **Definición de la rejilla:** En una rejilla n-dimensional se asigna a cada intersección de las celdas un vector de gradiente aleatorio, excepto en el

caso unidimensional que se asigna un escalar aleatorio entre -1 y 1 en vez de un vector.

- **Producto escalar:** Para asignar un valor a cada punto dentro de la rejilla, primero se localiza la celda sobre la que reside dicho punto. Luego, identificando cada una de las 2^n esquinas de la celda y sus vectores gradiente, se calcula un vector de desplazamiento desde dicha esquina hasta el punto. Después se calcula el producto escalar entre ambos vectores; este valor será 0 si el punto está en la esquina ya que el valor del vector de desplazamiento será nulo. Este proceso tiene una complejidad en tiempo de $O(2^n)$ siendo n el número de dimensiones.
- **Interpolación:** El último paso consiste en la interpolación de los 2^n productos escalares. Para ello se usa una función en la que la primera derivada sea 0 en los bordes, en consecuencia los valores próximos a la rejilla valdrán aproximadamente lo mismo que el producto escalar dotando así del aspecto continuo y visualmente natural tan característico del ruido de Perlin.

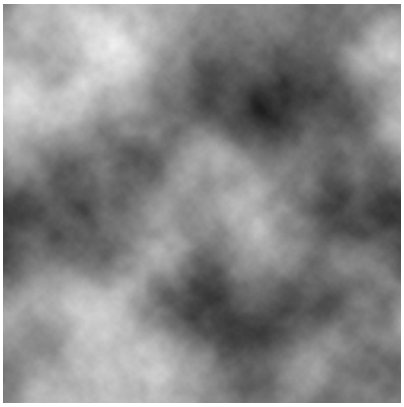

Figura 2.4: Ruido de Perlin.

- **El ruido Simplex,** es una mejora introducida al ruido Perlin, introducida por Perlin (2002), para superar las limitaciones del algoritmo original, especialmente en dimensiones superiores. Consta de tres pasos marcados:

  - **División del espacio en símplices:** En lugar de utilizar un rejilla como en el ruido de Perlin clásico, el ruido Simplex divide el espacio de n-dimensiones en símplices, que son las formas más simples posibles en cada dimensión (por ejemplo, triángulos en 2D, tetraedros en 3D...). Esta división reduce la complejidad computacional de la generación de ruido.
  - **Asignación de gradientes y cálculo de contribuciones:** A cada vértice del simplex se le asigna un vector de gradiente pseudoaleatorio. Para un punto dado dentro de un símplice, se calcula la contribución de cada vértice mediante una función de atenuación que depende de la distancia entre el punto y el vértice. Estas contribuciones se suman para obtener el valor final del ruido en ese punto.

- **Interpolación:** Las contribuciones de los vértices se combinan utilizando una función de interpolación suave, que garantiza la continuidad y suavidad del ruido generado. Esta interpolación se realiza de manera eficiente debido a la simplicidad de la estructura de los símplices.

- **Aprendizaje profundo o *Deep Learning*,** recientemente se ha estado investigando maneras de hacer PGC usando técnicas de *deep learning* como lo describen en el artículo Liu et al. (2021) donde describen los distintos tipos de contenido que se puede generar como: los niveles, los cuadros de texto, los modelos de las entidades, las texturas y el audio y la música. También describen los distintos métodos de entrenar a un modelo para que realice lo que llaman como *deep learning PGC* o DLPGC.

2.5. Sistemas que usan LLMs para generar contenido en videojuegos

Esta sección tiene como objetivo dar contexto a este proyecto mediante la revisión de trabajos similares en el ámbito de los LLMs y su uso en el mundo de los videojuegos, más concretamente en la generación y diseño de niveles. Haciendo esto se puede ver en qué se han centrado otros estudios y también qué los diferencia del nuestro.

En segundo lugar nos permitirá saber qué áreas se han estudiado más en profundidad y cuales son más superficiales. Con esto en mente se puede ver en qué áreas puede merecer la pena aportar otro enfoque distinto y en cuales no se debería profundizar mucho más.

En consecuencia, esta subsección y en general esta sección sienta las bases para comprender la relevancia y originalidad de la investigación que se está llevando a cabo en este proyecto.

Para saber qué trabajos se han seleccionado para analizarse. Vamos a explicar los distintos criterios en los cuales se ha basado esta búsqueda:

- El trabajo cuenta con los LLM como una herramienta principal en su investigación.

- El trabajo está relacionado con el área de los videojuegos.

- En un caso más concreto, el trabajo está relacionado con un área específica de los videojuegos siendo esta el diseño o generación de niveles.

El primer trabajo que revisaremos es: **Level generator thorugh Large Language models** Todd et al. (2023).

En el proyecto se plantea que los LLM son una herramienta muy potente y compleja capaz de realizar un abanico de tareas muy diverso y amplio. Se plantean si dentro de ese abanico existe la posibilidad de poder crear niveles para un juego en concreto, el Sokoban; un popular juego de puzzles creado en Japón.

En el principio del trabajo se plantean las dificultades de usar un LLM como generador de niveles, la primera de ellas es la representación donde en los últimos años se han estado investigando modelos para generar las bases de los juegos y luego trabajar sobre ellas o generar niveles con machine learning este grupo es conocido como *Procedural Content Generation through Machine Learning(PCGMP)* con varios métodos como redes neuronales convolucionales, modelos de Markov, entre otros Todd et al. (2023).

Estos modelos tienen en común que representan el espacio del juego como un arreglo de matrices de dos y tres dimensiones. La investigación dice que esto es un enfoque intuitivo ya que así los modelos pueden aprender rápido sobre las dinámicas espaciales de un juego. Pero el problema es que esta información tiene una extensión variable debido a la naturaleza de los juegos; por ejemplo, si tienes un mapa más simple que otro, el más pequeño estará representado dentro del juego de una manera más pequeña para ahorrar espacio en memoria. Debido a esto, tokenizar los mapas resulta en que los mapas tienen una extensión variable, dificultando el proceso de aprendizaje de un modelo.

Luego plantean que la cantidad de datos sobre un juego en concreto es bastante limitada y pequeña para lo que un LLM pueda llegar a necesitar para aprender correctamente, además de que no se puede saber si lo aprendido por el LLM afectará a la jugabilidad del nivel.

Luego se explica la base de datos usada y cómo está representada, en este caso los niveles son representados como una malla de caracteres ASCII con un carácter distinto para cada elemento distinto. Usan dos colecciones para el entrenamiento del modelo. El modelo usado fue GPT o *Generative Pre-trained Transformer*. Las métricas que usaron fueron: jugabilidad, novedoso, diversidad y exactitud.

Las preguntas que se plantearon responder fueron las siguientes:

- ¿Son capaces los LLM de generar niveles nuevos y jugables?
- ¿Afecta el pre-entrenamiento a la capacidad de generar nuevos niveles?

Se preguntan si hacer el pre-entrenamiento sería bueno para esta tarea en concreto. Porque se sabe que hacer pre-entrenamiento para tareas relacionadas con el lenguaje natural ha demostrado que los modelos dan respuestas más acertadas que no haciendo pre-entrenamiento. Para examinar esto consideran 3 variantes del modelo GPT-2: uno estándar, otro centrado en Java y otro sin entrenar. Cada modelo se entrena de la misma manera y en los resultados se ve que el modelo sin entrenar obtiene una puntuación más alta que los otros dos.

Se saca la conclusión de que el modelo de GPT-2 está restringido por la cantidad de datos que tiene disponible, pero se cree que GPT-3 puede generalizar mejor bases de datos pequeñas y que en un futuro pueda generar mejores resultados.

El segundo trabajo es **Game Generation via Large Languague Models**. Hu et al. (2024)

En este trabajo afirman que con el alzamiento de los LLM también lo ha hecho la generación procedimental de contenidos. Pero cita que los intentos actuales están enfocados en la creación de niveles sobre juegos ya existentes con sus normas y restricciones como podría ser el caso de *Mario Bros*. Pero creen que es posible generar una infraestructura basada en LLM que puede generar las normas y niveles de un juego y creen que pueden aportar algo de claridad en el área de generación de juegos mediante la generación procedimental de contenido.

Luego citan algunos ejemplos en los cuales los LLM han sido capaces de generar niveles para juegos específicos, como puede ser El primer ChaGPT que fue llevado a una competición PGC(procedural content generation) Abdullah et al. (2024) donde se centraba en generar niveles del juego *Angry Birds*. O el uso de múltiples LLM con los métodos conocidos para PGC formaron MarioGPT que puede generar infinitos niveles de *Super Mario Bros*. con el uso de prompts Sudhakaran et al. (2023) entre otros ejemplos de investigación que citan en el trabajo.

Pero ninguno ha intentado la generación de las reglas o normas de un juego y luego sus niveles, así que decidieron investigar ese punto.

A continuación se menciona que los LLMs están pre-entrenados con una gran base de datos para entender el lenguaje humano, como interpretarlo y finalmente cómo generar un texto en lenguaje natural. Luego menciona que la generación procedimental ha hecho un gran avance con el uso de *machine learning* y también *deep learning* pero se han centrado sobretodo en la generación de niveles sobre juegos existentes, como se ha mencionado antes, y opinan que falta generalización en ese área.

Se habla también de hacer *fine-tuning* a los LLMs para adecuarlos a la generación de niveles, pero esto sigue restringido a ciertos juegos además de que las normas de cómo se generan los niveles no son visibles.

Luego se describe un concepto llamado *Video Game Language Description* o VGLD; este es un lenguaje descriptivo que se ha usado en la inteligencia artificial de videojuegos. Después, describe cada elemento de este lenguaje, qué parámetros reciben y de qué se encarga cada uno.

A continuación explican la creación de lo que llaman LLMGG que es un LLM como, por ejemplo, GPT, un prompt de texto que contiene las reglas gramaticales de VGLD y lo que se espera. Dado eso, el LLM genera las reglas del juego y los niveles en forma de texto, que luego ese texto es traducido a un juego interactivo según la sintaxis de VGLD con motores como GVGAI. Concretan que esta infraestructura no depende del LLM que se use, sí que puede elegir cualquiera y el sistema al final generará un juego de mayor o peor calidad y también puede hacer fine-tuning con esos LLMs para esta tarea.

Luego describen las reglas de los prompts y también definen las reglas de cómo tienen que ser las respuestas, una vez hecho esto se explica el proceso que se ha hecho dónde han generado distintos prompts, cada uno más completo del anterior y han hecho 10 pruebas con cada uno de ellos con distintos LLMs, donde los resultados dicen

que aunque los LLM entiendan la sintaxis y las restricciones de VGLD es probable que sin expresarlo explícitamente en cada prompt haya resultados incorrectos y no sea posible parsearlo. Luego se comenta que pueden producir errores en la parte de la lógica del juego debido a las alucinaciones del LLM ya que de vez en cuando el LLM genera niveles que son gramaticalmente correctos según lo descrito pero no es posible jugarlos. También puntualizan que es complicado que el LLM se corrija a sí mismo porque aunque se lo digas y te conteste que lo ha entendido es bastante probable que vuelva a cometer esos mismos errores.

Y el último trabajo presentado se titula **Large Language Models and Games: A Survey and Roadmap** Gallotta et al. (2024).

En el principio citan un poco el origen de los LLMs que al principio solo tenían intereses académicos pero con el tiempo su popularidad aumentó exponencialmente tocando áreas de sitios muy distintos entre sí llegando incluso a los juegos de mesa, videojuegos también se han convertido en un campo para experimentar con los LLMs.

Donde van a recoger las investigaciones académicas y de desarrolladores independientes que usan los LLMs para la creación de juegos y su uso dentro de los juegos. Se van a centrar en investigar los recursos ya existentes que vayan a dejar un impacto en un futuro cercano o medio, porque exponen que los avances modernos son tan rápidos que si se centraran en eso su investigación estaría obsoleta en un año. Su objetivo es hacer un manual comprensivo de revisiones sobre las conferencias de inteligencia artificial y los juegos.

Luego define la terminología usada en la investigación como, por ejemplo, qué es un LLM, su definición, en qué están basados entre otros.

Luego define y explica los roles de los LLMs en los juegos, el primer rol es el de jugador. Los "jugadores LLM" donde los LLM requieren una transformación de sus respuestas para adecuarse al contexto del juego además que necesitan el estado del juego actual para poder dar respuestas en un nivel adecuado. Pero dependiendo del juego estos requisitos puede ser más simples o más complejos, en la investigación los clasifican en tres categorías generales distintas:

- Juegos donde los estados y las acciones pueden ser representados mediante tokens.
- Juegos donde los *inputs* principales y los diferentes *outputs* están en lenguaje natural.
- Juegos donde con aplicaciones externas se pueden controlar los movimientos de los jugadores mediante una API externa.

En la primera categoría en su mayoría son juegos de mesa por turnos como el ajedrez que se puede representar de manera más compacta que un *first person shooter* o FPS por ejemplo, ya que sus movimientos posibles y el estado del juego se puede representar de manera más sencilla, otro ejemplo es que los LLM pueden incluso jugar juegos de cartas como *Slay the spire* (Mega Crit, 2017) donde puede entender las sinergias entre las cartas leyendo las cartas de dentro del juego.

La segunda categoría pertenece a los juegos de aventuras, y citan de ejemplo a *Zork* (Infocom, 1977) donde el estado del juego es representado con lenguaje natural y este juego está ya equipado con un parser para tratar las respuesta de lenguaje natural. Esto sugiere que un LLM puede hacer peticiones al juego para que pueda jugar, después describen la manera que tiene el LLM de jugar este tipo de juegos donde el LLM produce múltiples posibles acciones dentro del juego y mediante el aprendizaje por refuerzo optimiza su forma de jugar.

Finalmente en la última categoría donde indican que esta categoría no son juegos como tal en cuanto a mecánicas pero son fáciles de implementar, puntualizan que la API es un elemento imprescindible porque permite al LLM actuar como jugadores sin generar acciones directamente pero produciendo programas que actúan como pólizas, citan de ejemplo a VOYAGER que hace la generación de código con GPT4 para jugar *Minecraft*.

Otro rol de los LLM en los juegos es el de ser los *Non-player characters* o NPCs que son agentes que existen en los videojuegos pero que sus acciones no son controladas por los jugadores. Los NPCs existen para enriquecer la experiencia del jugador y puede actuar como aliados, enemigos, mascotas, entre otros.

Por tanto puede ser una buena idea usar la IA para controlar estos personajes y los LLMs tienen una ventaja sobre el resto porque pueden entender el mundo del juego y adaptar sus respuesta a cómo se encuentra el estado del mundo dentro del juego o adaptar su comportamiento.

También describen los distintos roles de NPCs que puede tomar un LLM como, uno de narrador para el jugador, NPC que interactúan directamente con jugador o otros que no interactúan directamente pero lo ayudan indirectamente. Pero esto implica un riesgo y es el riesgo del LLM a alucinar y que de repente un NPC responda una cosa que el jugador no esperaba y que no esté relacionado con el estado del juego en ese momento.

Otros roles que para no extendernos demasiado vamos a mencionarlos son el de asistente del jugador o de comentarista dentro de los juegos o incluso pueden llegar a ser GM o game masters en juegos de rol de mesa.

El trabajo de investigación continua hablando de las limitaciones de los LLMs en los juegos, después realiza las conclusiones sobre cada uno de los roles que puede tomar un LLM dentro de un juego y finaliza comentando sobre los problemas éticos de los LLMs en los juegos.

Capítulo 3

Diseño y Arquitectura de un sistema de generación procedimental en base a peticiones del usuario con LLM

Para el sistema que realiza la generación procedimental, fue necesario establecer diferentes tipos de conexión entre los diferentes procesos que participan en la generación. En este capítulo se relata cómo se hizo. En esta sección también se introducirá el concepto del sistema para los usuarios.

3.1. Diseño

La aplicación o herramienta consistirá en un *plugin* dentro del entorno de Unreal Engine 5, donde al darle al botón de play aparecerá la interfaz gráfica de la aplicación, en la cual puedes introducir la descripción que quieres que el sistema genere. Una vez enviado, tienes que esperar a que el sistema lo cree y ponga las texturas correspondientes en los lugares adecuados y la aplicación habrá terminado de generar el nivel pedido por el usuario.

3.2. Arquitectura

La arquitectura del proyecto se organiza en tres áreas distintas interconectadas entre sí: el LLM, la "capa de procesamiento" y Unreal Engine 5. Se llama capa de procesamiento a todos los scripts necesarios para manejar el envío y recepción de las peticiones que solicitan las otras capas a través de sockets. En la siguiente figura, se muestra un diagrama con la arquitectura general del proyecto.

En la primera aproximación realizada, fue implementado un servidor en Python y un cliente para unirse y realizar las peticiones en Unreal. Para implementar dicho cliente fue usado un plugin gratuito llamado *Socketer* creado por un usuario de la comunidad, que facilitaba la implementación proporcionándonos una serie de *blueprints* con los que crear el cliente de manera sencilla.

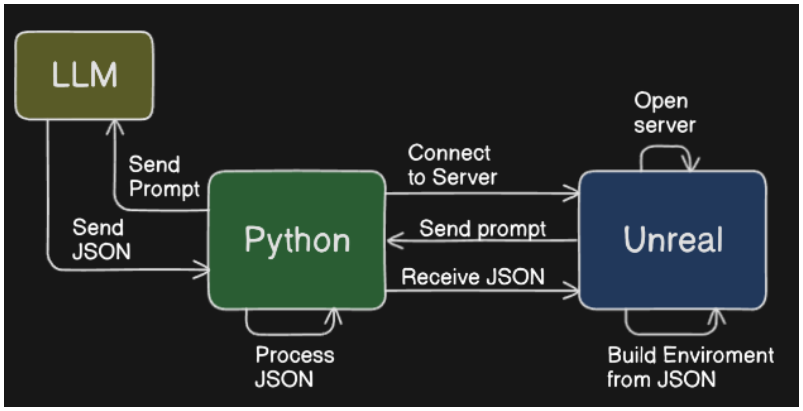

Figura 3.1: Arquitectura general del sistema

Los *blueprints* son un sistema de programación visual que permite a los desarrolladores crear la lógica del juego sin necesidad de escribir código tradicional. Este sistema se basa en la conexión de nodos visuales que representan eventos, funciones, variables y condiciones, facilitando así la creación de comportamientos complejos de forma más intuitiva.

Con un cliente y un servidor en funcionamiento, era necesario tener alguna manera de redactar las peticiones que se tenían que mandar a la LLM en Unreal, para ello fue creada un *widget* con un cuadro de texto para albergar las peticiones del usuario y un botón de "Enviar" que al ser presionado enviaba la cadena de texto a Python para que fuera redirigida a la LLM. Añadimos también de las librerías gratuitas de Unreal Engine 5 un controlador de personaje en primera persona para poder movernos alrededor del mapa y explorar el escenario con libertad.

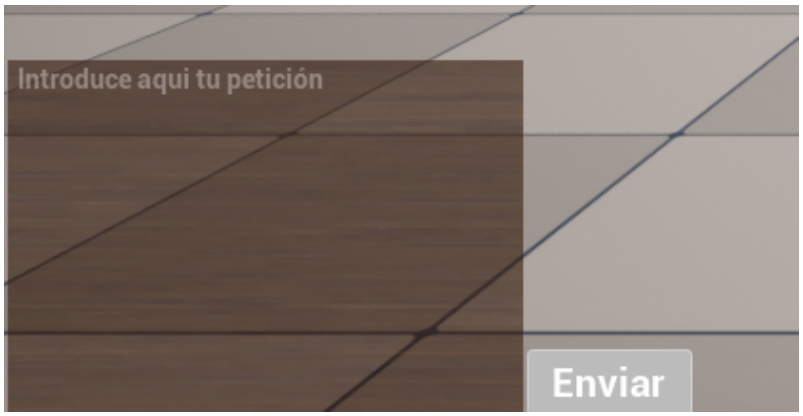

Figura 3.2: *Widget* para escribir peticiones

Con todas las funciones necesarias para la transmisión de mensajes implementadas, se realizaron unas pruebas preliminares que funcionaron correctamente, pero se descubrió que debido a la naturaleza del plugin cada vez que se quería realizar una prueba, era necesario reconectar el cliente en Unreal Engine 5 lo cual implicaba reiniciar el nivel y perder cualquier elemento ya generado. Así que se decidió cambiar el servidor a Unreal y el cliente a Python.

En la segunda y definitiva iteración, como ya se ha mencionado previamente, se implementó un servidor en Unreal y un cliente en Python. Debido a que el plugin *Socketer* usado previamente para la creación del cliente no tenía funcionalidades de servidor, fue necesario descartarlo y programar nuestro propio servidor. Para ello, en vez de usar el sistema de *blueprints*, se usaron las clases de C++, menos intuitivas pero mucho más flexibles. En inspiración del plugin anteriormente usado, se nombró a la clase *SocketerC*. Dicha clase contiene los siguientes métodos:

- *CreateTCPServer*: A este método se le proporciona un puerto y, usando el subsistema de sockets de Unreal, configura el tamaño del buffer del socket a 2MB, asigna la IP (en nuestro caso **127.0.0.1** o **localhost**) e intenta abrir un socket en dicha IP. Si lo consigue, comienza a escuchar hasta un máximo de ocho conexiones. Esta función devuelve verdadero si se ha podido crear un servidor y falso en caso de que no haya sido posible.

- *AcceptConections*: Este método, es ejecutado permanentemente usando un bucle infinito y pregunta constantemente por nuevas conexiones. En caso de que las reciba, comprueba si ya ha sido conectado otro cliente previamente y en caso afirmativo, rechaza la conexión pendiente, ya que en nuestra implementación solo es necesaria la conexión con un único cliente, en caso de estar libre de conexiones, la petición es aceptada y se asigna el socket a la variable *Client*. Una vez aceptada una conexión, el cliente es preguntado en repetidas ocasiones para asegurarse de que sigue conectado.

- *SendData*: Esta función recibe como parámetro el mensaje o petición que quiere ser enviado a la LLM y lo envía, para ello, primero comprueba que hay un cliente conectado y, en caso afirmativo, convierte la cadena de texto a UTF-8, después calcula el tamaño en bytes de la cadena y lo envía al cliente, en caso de recepción exitosa, se envía el mensaje.

- *ReceiveData*: El método contrario, el servidor espera perpetuamente la entrada de datos, y en caso de ser recibidos, primero se reciben cuatro bytes con el tamaño del mensaje posterior, luego se recibe el mensaje en formato JSON, que es impreso por consola y se almacena en la variable global *ReceivedDataCache* siempre y cuando el valor previo a su modificación y los datos recibidos sean diferentes.

- *GetLatestReceivedData*: Esta función devuelve *ReceivedDataCache*.

- *CloseSocket*: Esta función al ser ejecutada, desconecta a todos los clientes, para más tarde destruir sus sockets y cerrar el servidor.

Esta clase nos proporcionaba todo lo necesario para crear y cerrar un servidor, aceptar conexiones, enviar y recibir datos, además de almacenarlos.
Aún así, la clase *SocketerC* no era posible instanciarla dentro del nivel, así que fue necesaria la creación de una segunda clase esta vez híbrida (con métodos de C++ y en *blueprints*) *SocketerSpawner*, para añadirla al nivel en ejecución, esta función consiste de las siguientes funciones:

- *Constructor*: Establece el valor de la variable *bDataReceived* a falso y a la variable *lastData* a un símbolo especial como es el símbolo de "$" que nunca va a coincidir con el JSON recibido. Esta variable es usada para comparar los datos recibidos con los últimos guardados para ver si son diferentes.

- *BeginPlay*: Este método es llamado al ejecutar el programa, cuando se ejecuta, se crea una instancia de *SocketerC* y en caso de crearla correctamente llama a su método *CreateTCPServer* en el puerto **7777**.

- *HasReceivedData*: Devuelve cierto si se han recibido nuevos datos, y falso en caso contrario. Esta función es invocable desde *blueprint*.

- *HandleReceivedData*: Este método compara los nuevos datos recibidos con los ya almacenados en la variable *lastData* y en caso de ser diferentes, los almacena en dicha variable, establece *bDataReceived* a verdadero y devuelve verdadero, devuelve falso en caso de que los datos sean iguales. Esta función es invocable desde *blueprint*.

- *GetReceivedData*: Este método público devuelve los últimos datos recibidos y estable *bDataReceived* a falso. Esta función es invocable desde *blueprint*.

- *Tick*: Esta función es llamada cada frame, aumentando una variable *TimeAccumulator* con el valor de *DeltaTime*, hasta que llega a tres segundos, momento en el cuál el contador vuelve a cero y se llama a *HandleReceivedData*.

- *SendData*: Esta función recibe el *prompt* mandado por el usuario y llama a *Server->SendData*. Esta función es invocable desde *blueprint*.

- *StopServer*: Si existe una instancia de servidor, esta función llama al método interno para terminar el servidor y, borra su referencia además, de notificar el cierre con un mensaje via consola.

- *EndPlay*: Un método de Unreal que se llama cuando se termina la ejecución del programa, llama a *StopServer*.

Para resumir, esta clase permite instanciar *SocketerC* y proporciona una interfaz para llamar a sus métodos internos desde *blueprints*.

En el lado receptor de las peticiones en Python, la interconexión consta de dos capas, una que comunica Python con Unreal, la clase *SocketerClient.py* y otra que comunica dicha clase con el script de LLM, *ManagerServer*. En la figura 3.3 se puede observar el recorrido que realizan los datos.

La clase *ManagerServer* es una clase auxiliar que se encarga de la comunicación entre las clases *LLM.py* y *SocketerClient.py*, para ello abre un servidor local en el puerto **50000** usa una cola a la que se suscriben las dos clases al principio de la ejecución y se transfieren mensajes usando operaciones bloqueantes de *put* y *get*.

La clase *SocketerClient* se encarga de la comunicación con el servidor, contiene los siguientes métodos:

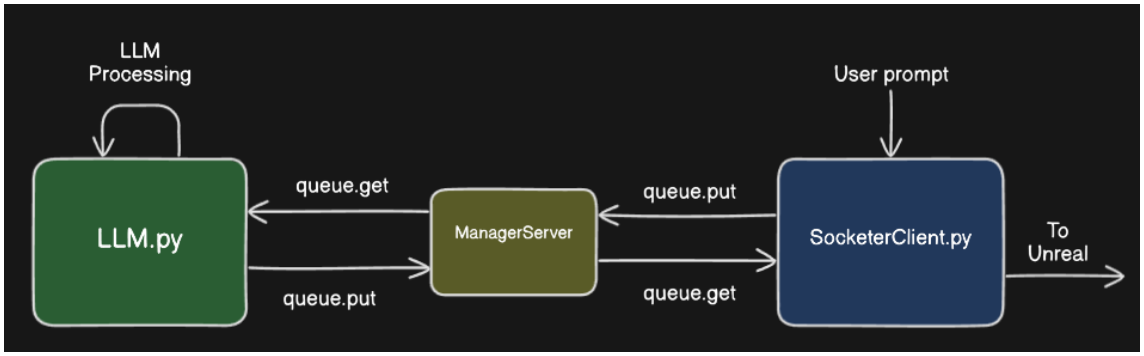

Figura 3.3: Diagrama de flujo de la interconexión de los scripts de Python

- *connect*: Este método establece la conexión con el servidor a través de un socket, conectandose a **localhost:7777**.
- *receive string*: Esta función espera y recibe un mensaje de cadena de texto proveniente del servidor. Primero se reciben 4 bytes con la longitud del mensaje y, luego el mensaje en sí, una vez recibido lo escribe dentro de la cola del *ManagerServer* para que la LLM lo reciba.
- *send json*: Envía el JSON procesado por la LLM al servidor de Unreal para ello, primero envía 4 bytes con el tamaño del JSON y luego el JSON codificado en UTF-8.
- *listen and respond*: Este método espera permanentemente cierto tiempo hasta devolver timeout, mientras se espera, comprueba si ha recibido nuevos datos usando el método *receive string*, más tarde se queda esperando con la operación bloqueante *queue.get* a recibir el JSON procesado por la LLM, una vez recibido llama a *send json* para enviarlos.

Capítulo 4

Diseño del motor de generación procedimental de niveles basado en LLM

Esta parte del proyecto se encarga de describir los mensajes que va a recibir y enviar un LLM dado, donde para conseguir nuestro objetivo de poder procesar una respuesta de este LLM es necesario que ese mensaje esté estructurado según unas normas dadas, que explicaremos más adelante.

4.1. Prototipo del sistema con LLM en local

En los primeros prototipos del proyecto, se estableció la idea de abrir el LLM en local para no tener que depender de una conexión a internet y que el usuario pueda elegir el LLM que quiera usar según las capacidades técnicas que tenga, sin que llegue a afectar al resultado obtenido o al resultado esperado por el usuario.

Una vez decidido eso, se estableció la clase que se encargaría de "hablar" con el LLM que se estuviese ejecutando en ese momento, llamada *LLM.py*. Antes de explicar la clase vamos a detallar las normas del JSON.
La normas para la formación del JSON de respuesta que se establecieron en este momento son las siguientes:

- El archivo JSON será solo un *JSON object*
- Cada *JSON object* tendrá cinco campos: *type*, *name*, *description*, *localization* y *content*
- Los campos de *type*, *name* y *description* son strings.

  Donde los valores de *type* pueden variar entre dos valores aceptados *object* y *environment*. Los otros dos campos pueden valer cualquier palabra natural.
- El campo de *localization* varía según si el campo *type* tiene como valor *object* o *environment*.

  Si es el primer valor de *type* el campo solo se compone de un **JSON object** llamado "A" que dentro tiene dos campos, "x" e "y" que son números flotantes.

Si es el segundo valor de *type* el campos se compone de dos *JSON objects* llamados "A" y "B" que los dos tienen tienen los mismos campos que el *JSON object* "A" cuando el campo *localization* vale *object*.

- Finalmente el campo *content* es un *JSON array* de otros *JSON object* como el descrito anteriormente, donde este campo solo tendrá valores cuando el campo *type* de ese *JSON object* tiene como valor *environment*.

Todas estas normas se pueden ver mejor en la figura 4.1

```
{
    "tipo": "entorno",
    "nombre": "EntornoPrincipal",
    "descripcion": "descripcion",
    "localizacion": {
        "P1": {
            "x": -500.0,
            "y": -500.0
        },
        "P2": {
            "x": 500.0,
            "y": 500.0
        }
    },
    "contenido": [
        {
        "tipo": "objeto",
        "nombre": "objeto1",
        "descripcion": "descripcion",
        "localizacion": {
            "P1": {
                "x": -100.0,
                "y": -100.0
            }
        },
        "contenido": []
        }
    ]
}
```

Figura 4.1: Un ejemplo de JSON bien estructurado

En un primer acercamiento se intentó usar la plataforma *Ollama* para ejecutar los modelos que ofrece la plataforma como LLama 3.2 o Mistral entre otros.
En la clase *LLM.py* estaban las siguientes funciones:

- *Generar-input-usuario:* Esta función era la encargada principalmente de dos tareas, obtener el input de la consola introducido por el usuario, y añadirle un "extra input" el cual se trataba de un conjunto de líneas de texto, el cual fuimos determinando a base de prueba y error, el cual buscaba que la generación fuese la más óptima posible.

  El extra input era una de las partes fundamentales en la generación del JSON, en él explicamos la estructura que debía tener cada campo y subcampo, que franja de valores debían tomar los campos que estuviesen acotados, el más claro es objeto u entorno como clave del JSON, o las implicaciones de tomar ciertos valores modificando la estructura de ese JSON fuera de los parámetros originales.

  En las últimas actualizaciones de esta versión añadimos un archivo JSON de prueba el cual mandaría junto al input modificado lo cual era sin que lo supiéramos una estrategia de entrenamiento ya existente conocida como reflexión, aplicada de manera pobre debido a nuestro desconocimiento en las primeras instancias.

- *Interactuar-llm:* Esta era la función encargada de enviar el input modificado en "Generar-input-usuario" al LLM y recibir su respuesta, esta sigue el modelo clásico de la interacción con LLMs.

  Genera una respuesta a partir de un mensaje compuesto por el modelo a usar y los mensajes enviados, en los cuales se debe poner el rol de quien lo envía y el contenido, el cual es el input. Y generaba una respuesta o output a partir de la cual podremos trabajar para conseguir el JSON.

- *Procesar-guardar-json:* Esta era la segunda parte fundamental en la generación del JSON a partir de el LLM, ya que en una primera instancia se encarga de limpiar el output dado por el LLM. Se hizo una función, la cual no ha dado problemas, encargada de encontrar el primer y el último corchete del archivo JSON de manera que todo el texto "basura" generado por el LLM, es decir, cadenas de texto que no nos interesan tener según las normas que se explicarán después, son eliminadas.

  Pudiendo así tener en una cadena de texto el JSON completo y sin imperfecciones para poder guardarlo correctamente mediante las funciones básicas de la librería "json".

- *Main:* Como todo programa ejecutable este tenía un main encargado de cargar el JSON de ejemplo, y posteriormente lanzar ordenadamente las funciones de modo que obtengas el input, lo modifiques, lo envíes a el LLM y tras obtener la respuesta sea procesada y guardada correctamente.

Se observó que funcionaba correctamente para un primer acercamiento pero se buscaba más libertad a la hora de elegir el LLM a usar por lo que decidimos descartar

la idea de usar *Ollama* y usar directamente los LLM con la API de OpenAI para usarlos en la clase.

4.2. Prototipo del sistema usando llamafiles

Como se comentó al final de la primera iteración, sección 4.1, se decidió abandonar la idea de usar *Ollama* para usar otros LLMs usando los LLamafiles con la API de OpenAI, para ello se modificaron las siguientes funciones de la clase *LLM.py*:

- *Interactuar-llm:* La modificación principal en esta función es el cambio de modelo de ollama a la api de open AI y un cambio poco significativo, mover el procesamiento de la salida del LLM a la misma función, esto se debe a la implementación del método reflexion mediante el cual reenviamos ejemplos de JSONs bien generados a el LLM de nuevo y le pedimos posteriormente que mejore la salida evitando errores de ejecución a cambio de más tiempo de procesamiento.

- *Procesar-guardar-json:* Se trasladó la parte de gestión del output a Interactuar-llm.

- *Main:* Se añadió una llamada al principio de relexion aprovechando el tiempo de ejecución de espera Unreal Engine 5.

  Se movió el extra input al main ya que se eliminó la función de generar input de python, probamos a sustituir este método mediante una librería reciente llamada *strict json* pero el resultado fue muy decepcionante, ya que empezaban a dar errores y generaba JSONs notablemente peores que los generados por nuestra función, por lo que decidimos volver a ella e incluso seguir experimentando y mejorandola para el LLM que usamos actualmente.

  Se añadió todo el sistema de recepción y envío de datos de Unreal Engine 5 mediante sockets, explicado en profundidad posteriormente en su apartado.

  Por último se añadió la estructura y funciones necesarias para el uso de los embeddings para la modificación del JSON.

Y también se añadieron las siguientes funciones a la clase *LLM.py*:

- *Extract-names:* Es la función encargada de modificar el json de manera que los objetos y entornos generados por el LLM pasen un pequeño filtro de modo que las palabras sean sustituidas por la palabra cuya relación semántica sea superior a un margen seleccionable respecto a las palabras para las cuales existe un *asset* gráfico, cabe recalcar que si dicha palabra tiene un *asset* la comparación con sí misma dará una puntuación perfecta por lo cual siempre se sustituirá por sí misma de modo que no se modificará realmente, de modo que buscamos generar el menor número de cubos (objetos para los cuales no tenemos textura) posibles, esto se explicará más en profundidad en el capitulo 5.

Esta función fue probada primero en español, idioma en el cual hasta los embeddings más grandes que encontramos no daban valores que encontrásemos lógicos, por lo cual tuvimos que cambiar el proyecto completamente al inglés, el cual funciona de una manera excelente en comparación.

- *Read-names-from-file:* Función utilizada para leer los nombres de los assets guardados en un archivo de texto para su uso en los embeddings.
- *Send-to-llm:* Función encargada de interactuar con el LLM sin recibir ninguna respuesta, utilizada para el método de reflexion.
- *Reflexion-llm:* Función encargada de enviar los JSONs de ejemplo guardados en una carpeta y enviados por *Send-to-llm* con un texto explicativo para que los tome de ejemplo

También se eliminó la función de *Generar-input-usuario* ya que se prescindió de esta función al cambiar la manera de obtener el input del usuario, de la consola de Python pasó a ser un chatbox interno en el modo play de Unreal Engine 5, donde también se explicará más en detalle esto en el capítulo 5.

Por último, se añadieron también campos como NUMEX (number of examples) y un booleano para poder activar o desactivar la reflexion, además del código referente al script que conecta el módulo de Python al socket.

Capítulo 5

Desarrollo del sistema de generación procedimental en base a peticiones del usuario con LLM

En este capítulo se explicará en dos secciones. En la primera se explican los detalles del desarrollo del módulo de Unreal Engine 5, así como qué tipo de proyecto fue usado y luego en la siguiente sección se exponen las texturas usadas en este proyecto y cuales fueron las razones por las que se usaron.

5.1. Implementación del entorno en Unreal Engine 5

Primero se explicarán las funcionalidades de la parte relacionada con *Unreal Engine 5*, que llamaremos UE5 a partir de ahora. Donde esta parte fue llamada antes "capa de renderizado". Por último, se explicará cómo funcionan las texturas en UE5 y dentro de la aplicación.

Esta capa se encuentra en nuestro proyecto hecho sobre *Blueprints*, en la primera aproximación, como se comentó en la sección de arquitectura y luego se pasó a un proyecto hecho sobre C++ de UE5 donde se combinaron clases de C++ y *Blueprints* de UE5.

El *Blueprint* llamado *SpawnerJSON.uasset*. Vamos a dividir este Blueprint en distintas partes y a explicar cada una de ellas. Estas partes son: *event graph*, función de contenido recursivo, función *spawn mesh* y función *spawn mesh recursivo*.

Antes de explicar cada parte citada antes, vamos a hacer una explicación general del flujo que sigue este *Blueprint*. El *Blueprint* recibe un JSON bien formado según las reglas comentadas en la sección del LLM y lo parsea según los campos que tiene definidos, como dentro de los *JSON object* con el campo *type* con valor *environment* pueden tener o no otro *JSON object* dentro, por tanto ese JSON necesita ser parseado recursivamente con la función contenido recursivo, hasta encontrar un *JSON obejct* que no tenga otro dentro, mientras se parsea se van generando las texturas de los

objetos con las funciones *spawn mesh* y *spawn mesh recursivo*.

Event Graph de *SpawnerJSON*

Empezamos explicando más detalladamente *event graph*. Vamos a ir explicando por partes, describiéndolo, para explicar el flujo del *Blueprint* ya que este es grande.

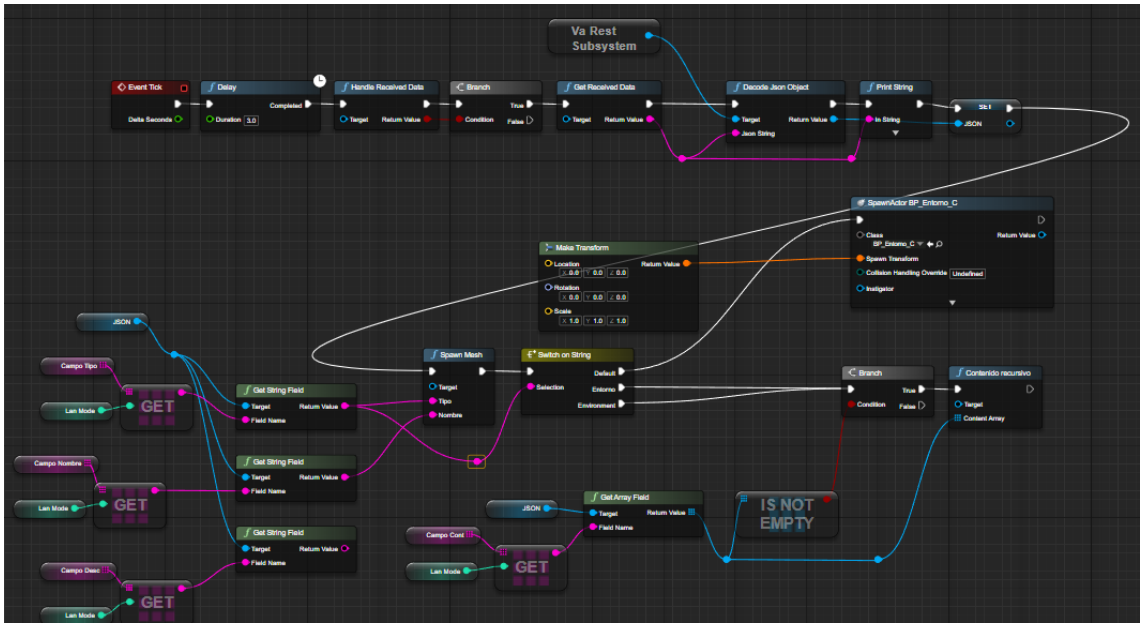

Figura 5.1: Figura que muestra el flow de *event graph*

Como se puede ver en la figura 5.1 el flujo empieza con el nodo *Event tick*, que es rojo, encargado de accionar este *Blueprint* con cada tick dentro del modo juego, luego pasa a un nodo llamado *delay* para que el nodo *Handle Received Data* le pueda dar tiempo a empezar a procesar la información, en caso de haberla recibido. Si ha recibido un mensaje contenido en *Handle Received data* se invoca a un *VaRestSubsystem* del plugin de *VaRest* que invoca al nodo *Decode Json Object* encargado de codificar el string recibido a un archivo.

Como se ve en la parte de abajo a la izquierda de la figura 5.1 una vez codificado el JSON como se ha descrito anteriormente, se leen los campos de *type*, *name* y *description* y luego se obtiene el valor del campo con el nodo *Get String Field*. Una vez obtenidos esos valores, los valores de *type* y de *name* se pasan a la función *spawn mesh* para determinar qué textura tendrá. Más adelante se explicará este proceso. Una vez procesada la textura se determina si esa textura era un *environment* o un *object*,en caso de ser un *environment* se revisa su campo *content* por ver si quedan *JSON objects* por procesar; este proceso se explica más adelante. En caso de que el valor del campo, una vez procesado qué textura tiene asignado este objeto, si el valor del campo *type* es distinto de los valores *object* o *environment* se generará en la posición de la textura, una textura por defecto para indicar que ese valor es incorrecto; esto se hace así porque en la función *spawn mesh* suponemos que el valor del campo *type* es un valor correcto.

Como se ve en la parte de abajo a la derecha de la figura 5.1 se mira lo mencionado antes, que su campo *content* no sea vacío obteniendo su valor llamando al nodo *Get Array Field*; y si es así, se llama al nodo "contenido recursivo".

Función de "Contenido recursivo" de *SpawnerJSON*

Ahora se explicará la función de "Contenido Recursivo".

Contenido recursivo es una función definida dentro del *blueprint SpawnerJSON* la cual recibe un array, llamados matrices en UE5, de *VaRest JSON value* referencia a objeto. El flujo de esta función empieza mandando el array recibido a un nodo *For Each Loop* donde se recorre cada uno de los *JSON object* del array pasado. Crea una variable llamada *Recursive JSON* para parsear los campos del *JSON object* correspondiente de la misma forma que en *Event Graph*, visto anteriormente. Así que solo se explicarán los cambios con respecto a la clase ya mencionada, para evitar repetir.

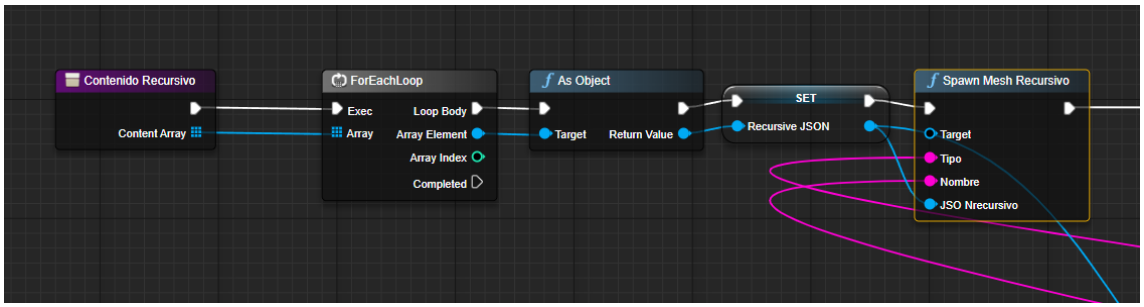

Figura 5.2: Figura que muestra las diferencias entre *event graph* y la funcion contenido recursivo

La primera parte y la única que cambia con respecto a *Event Graph* es la que se muestra en la figura 5.2. El array pasado como parámetro se manda al nodo *For Each Loop* donde cada elemento de este, que es un *JSON Value* referencia a objeto, sale y se castea a *JSON object* con el nodo *As Object* y luego se configura como una variable local para la función. Otra diferencia que hay es que la función encargada de generar las texturas aquí es *spawn mesh recursivo* y no *spawn mesh*, el resto de la función es igual a *Event graph*.

Función *spawn mesh* y *spawn mesh* recursivo de *SpawnerJSON*

Por último, vamos a explicar la función *Spawn Mesh* que recibe dos strings, uno llamado *Type* y otro llamado *Name* y no devuelve nada.

La función *spawn mesh* es muy grande y, además, por la naturaleza de la función, hay muchas partes que se repiten. Por tanto, vamos a explicar el flujo de la función con un ejemplo.

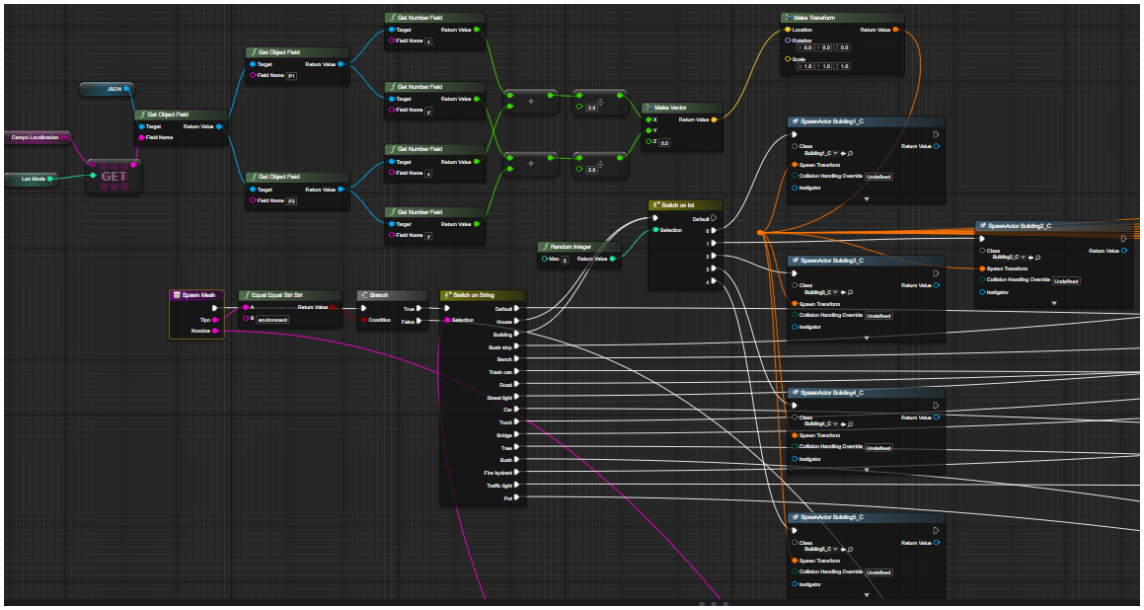

Figura 5.3: Figura que muestra el flujo para asignar una textura en la funcion *spawn mesh*

Veamos como sería el flujo de la función en caso de renderizar una textura en el caso de que a la función le llegase el string *building* en el parámetro *type* y el string *environment* en el parámetro *name*. Lo primero que se mira al entrar a la función es que el parámetro de entrada *type* sea la cadena *environment* o la cadena *object* para ir a la parte de arriba de la función o la parte de abajo como se puede ver en la parte de la izquierda de la figura 5.3, como en este ejemplo es la cadena *environment* irá a la parte de arriba. Luego lleva el parámetro *name* a un switch donde en caso de coincidir con alguna de las cadenas de dentro de este, el flujo sigue al siguiente switch donde se elige una textura al azar de entre las disponibles para esa palabra y en caso de no coincidir se cargará una textura por defecto o BPentorno o BPobjeto como se puede ver en la parte derecha de la figura 5.3. En el caso del ejemplo se irá al switch para elegir una textura de edificio. Cuando se ha escogido una textura concreta, se calcula su posición de maneras distintas que se mostrarán después según sea un *environment* o un *object* y se llama al nodo *spawn actor from class* que spawnea la malla estática correspondiente en las coordenadas que hemos calculado previamente.

Quedaría por explicar en profundidad la función *Spawn mesh recursivo* pero es una copia de la función anteriormente explicada, la única diferencia es que recibe un parámetro más que es el *JSON object* que se está procesando en ese momento.

5.2. Sección de texturas

En esta sección se explicará cómo se han configurado las texturas para este trabajo y también se mostrarán qué texturas o mallas se han escogido y la razón por la cual se han escogido.

Para la creación e implementación de las texturas del proyecto se decidió usar mallas

estáticas o *Static meshes*, estas se pueden definir como objetos en 3D o mallas de polígonos que no se pueden modificar en tiempo de ejecución, de ahí el nombre de estáticas. Además al ser un objeto en 3D, la malla en sí solo tiene la forma. En UE5 los objetos en 3D son solo la forma, donde tú, como usuario, le puedes cambiar la textura que tenga a ese objeto para hacer dos mallas distintas, por ejemplo; tienes un objeto en 3D que es un cubo y dos texturas una que recuerda a una pared de ladrillos y otra que recuerda a una pared de piedra. Si pones la textura de pared de ladrillos tendrás una malla estática distinta que si pones la textura de pared de piedra. Un ejemplo con las figuras 5.4 donde la malla tiene textura y en la figura 5.5 donde la misma malla no tiene textura.

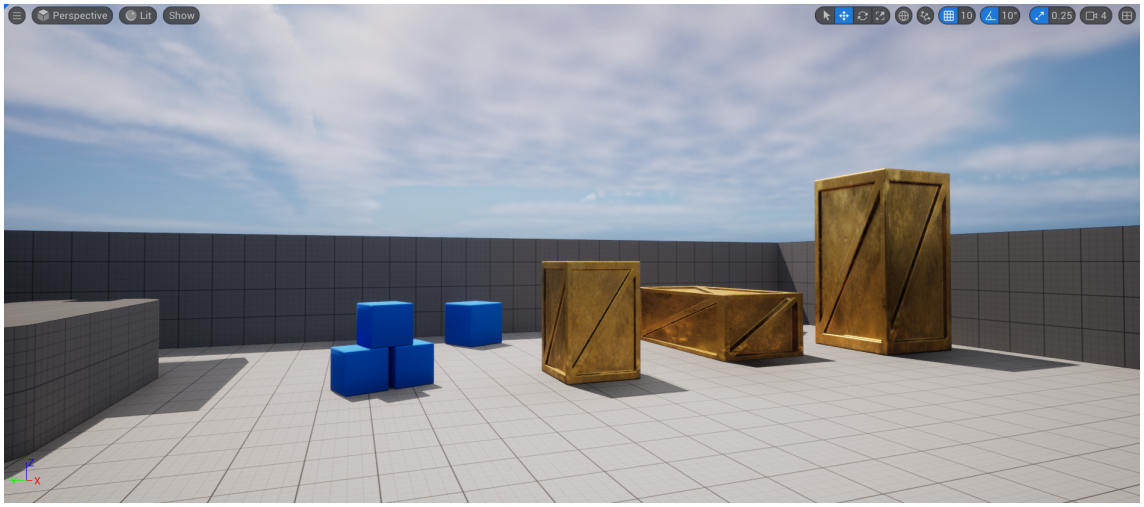

Figura 5.4: Ejemplo de una malla estática con texturas

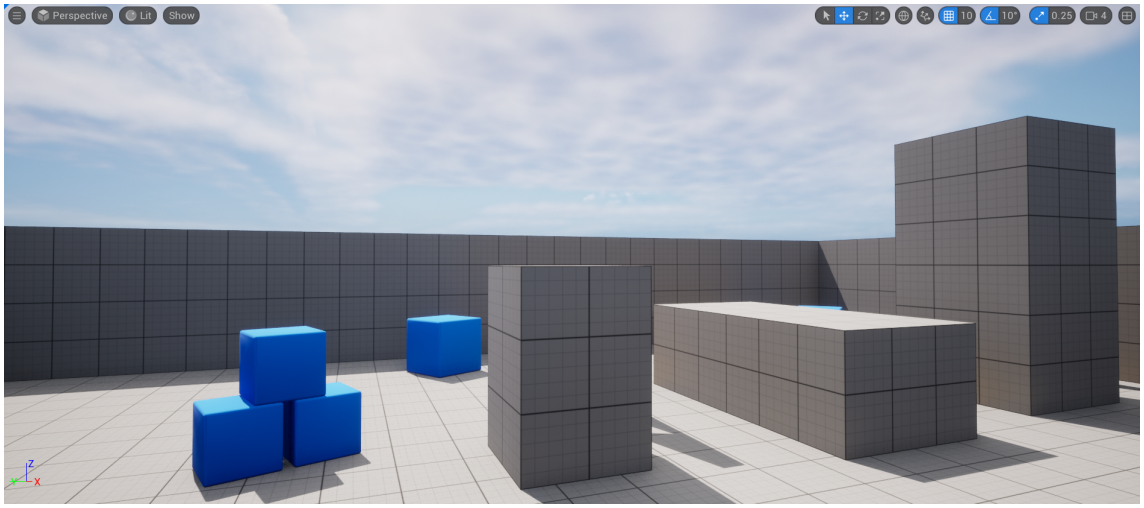

Figura 5.5: Ejemplo de una malla estática sin texturas

Pero las mallas estáticas no se pueden manejar dentro de los *blueprints* de UE5. Los *blueprints* solo pueden modificar o crear actores dentro del nivel. Para conseguir que el *blueprint* de *SpawnerJSON* pudiese crear en el nivel las mallas estáticas elegidas

en el proyecto, que se expondrán más adelante en esta sección, se tuvo que hacer lo siguiente:

- Creación de una *Bluerpint class* que hereda de *StaticMeshActor* para cada una de las texturas usadas en el proyecto.
- Dentro de cada clase de *StaticMeshActor* se le asigna su correspondiente malla estática y también se le asignan las texturas de esa malla estática.

La razón por la cual se ha usado este sistema es porque con un *blueprint* que es un *StaticMeshActor* es porque la aplicación puede colocar directamente en el mapa estos actores, como lo haría un usuario. También se valoró usar la instanciación de mallas estáticas, usado cuando se quiere repetir la misma malla muchas veces siguiendo un patrón y reduciría el tiempo de ejecución. Pero valorando la idea, se vio que no aportaría un valor real al proyecto y se descartó. Volviendo a la idea de que cada actor puesto en el nivel fuese distinto uno de otro.

Para el proyecto se eligió un paquete de *assets* llamado *LowPolyCity*[1] que es gratuito donde las texturas del paquete son simples, pero eso no es sinónimo de que sea algo malo, además como no tienen muchos detalles, las texturas del paquete cargan rápido.
Como son muchas texturas, se van a enseñar algunos ejemplos de estas, se pueden ver en las figuras: 5.6, 5.7, 5.8, 5.9, 5.10 y 5.11

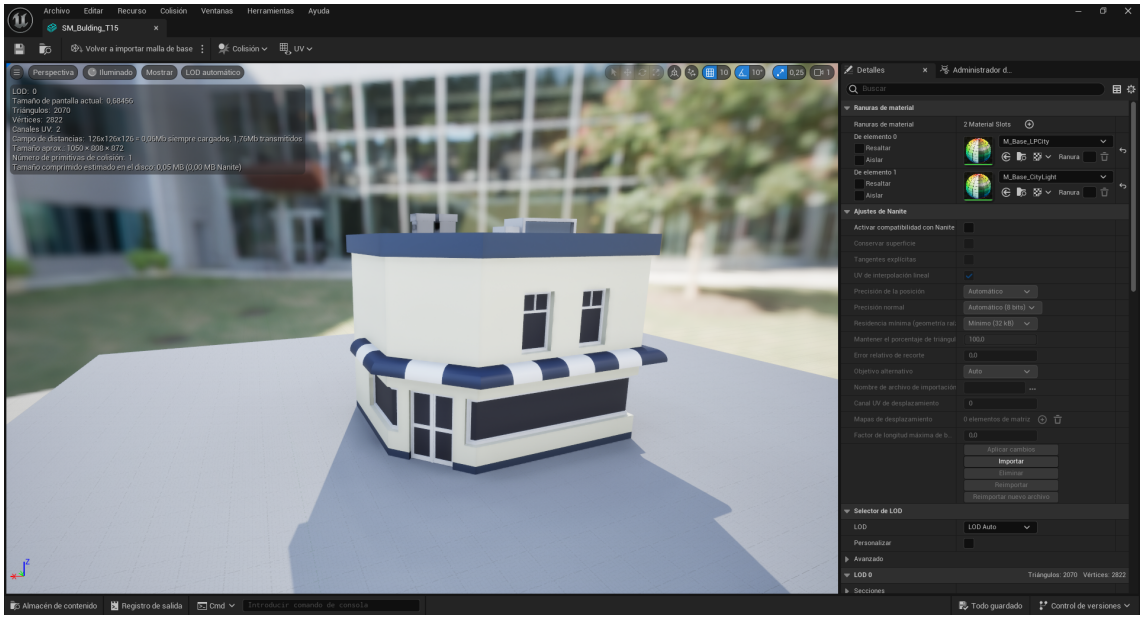

Figura 5.6: Un edificio del paquete *LowPolyCity*

[1]Creado por DrCG, https://www.artstation.com/marketplace/p/VOD3A/city-asset-pack-for-unreal-engine-toon-stylized-modular-low-poly-environment

Figura 5.7: Un coche del paquete *LowPolyCity*

Figura 5.8: Un camión del paquete *LowPolyCity*

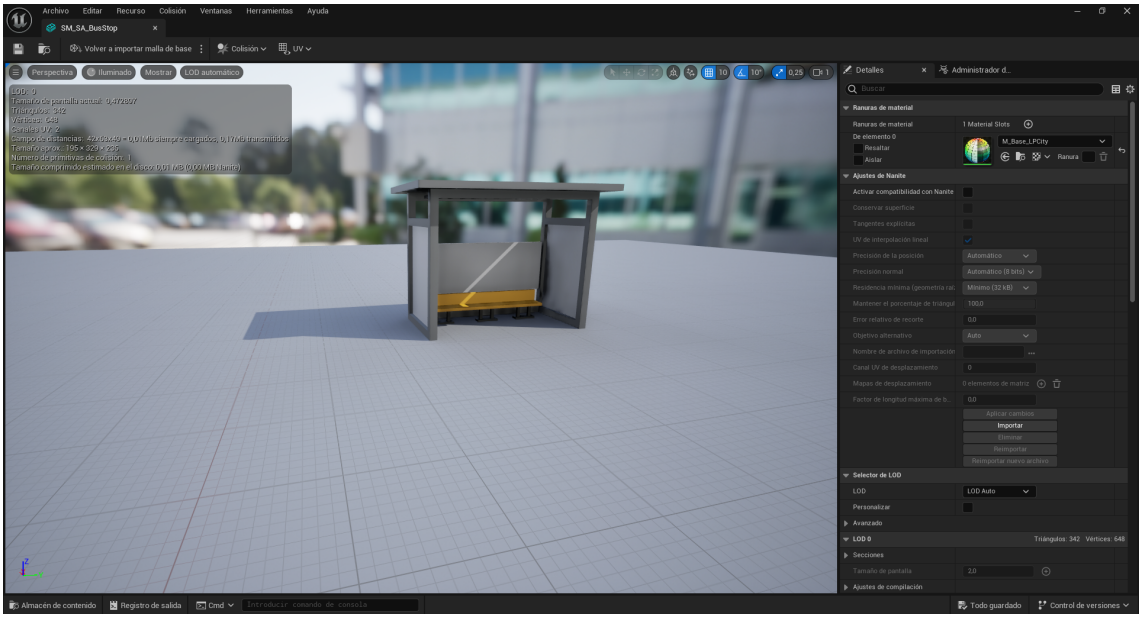

Figura 5.9: Una parada de autobus del paquete *LowPolyCity*

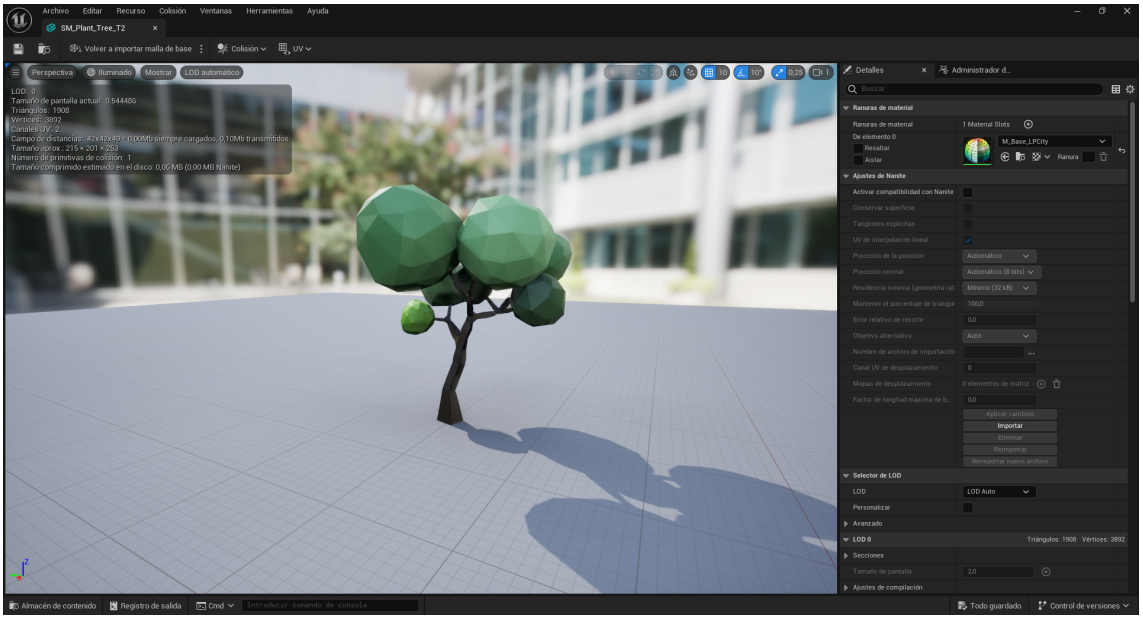

Figura 5.10: Un árbol del paquete *LowPolyCity*

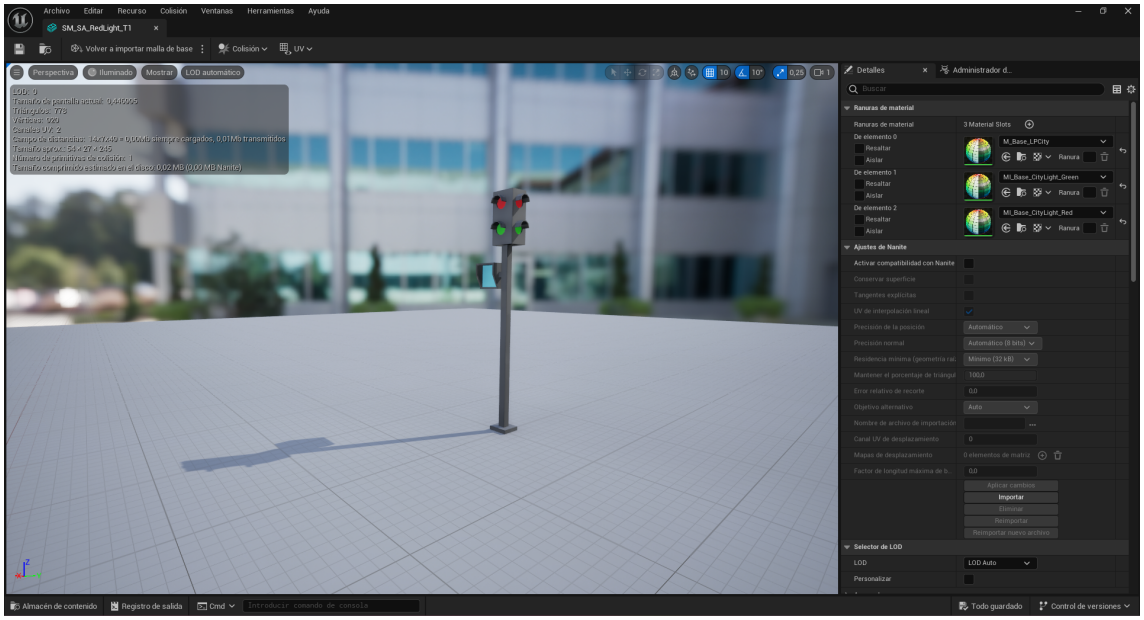

Figura 5.11: Un semáforo del paquete *LowPolyCity*

Capítulo 6

Conclusiones del análisis de métricas aplicado a la salida del LLM

En esta sección se muestra de una manera estadística, gráfica y lo más objetiva posible, los valores más importantes de entorno que rodea a la salida de la información del LLM, a pesar de ser unos valores impuestos por unas reglas predefinidas y modificadas por nosotros. Se probarán 3 LLMs con diferente capacidad de cómputo, para tener puntos de vista diferentes respecto a los valores medidos. Se compararán Llama 3.2 (8gb) la versión que elegimos en un primer momento tras probar varias durante el trabajo, Phi 2 (Phi-2-orange-Q4_K_M) un modelo pequeño de *llamafile* destacado en su tamaño y una versión reducida de Llama 3.2 de 1gb que nos fue de gran utilidad en los primeros compases del trabajo.

Para ello hemos determinado varios campos de estudio y metodologías de prueba, los cuales son los más importantes a la hora de valorar el trabajo de estas máquinas. Y estos son:

1. Precisión: En un primer momento iba a ser una medida levemente subjetiva mediante la cual se iba a valorar en partes iguales el posicionamiento de los objetos y el parecido entre lo pedido por el usuario y la salida proporcionada por nuestro programa. Lo cual se redujo a solamente la segunda puesto que la primera la arreglamos a nivel de usuario, y por ello lo descartamos. Esta medida se verá representada por Número de objetos pedidos representados/Número de objetos pedidos totales.

2. Tiempo: Es la medida del tiempo de ejecución del LLM, es decir el tiempo que tarda en darte una salida desde el segundo que le entra la información, por lo que mide unicamente el tiempo de cómputo de la parte referente al programa de Python, el cual es un aproximado de máximo 5 segundos de error. Más adelante explicaré como lo conseguimos medir.

3. Porcentaje de cubos: Hemos llamado de esta manera al porcentaje de objetos que se generarían sin textura, puesto que no disponemos de texturas para ellos, sobre el total de elementos generados.

4. Número de errores: Esta es la variable que contabiliza las veces que ha fallado el LLM al proporcionar un *output*, dando un error de ejecución,po ello su valor será 0 o 1, dependiendo si ha dado error o no. Por como está diseñado nuestro programa y la cantidad de pruebas anteriores nunca debería fallar, pero lo hemos puesto ya que se realizarán multitud de pruebas y en algún punto se cambiará de LLM para realizar una comparativa, la cual si podría empezar a fallar.

Hemos elegido estos campos porque reflejan bien las métricas del diseño y por otra parte se han descartado los siguientes campos que aparecen seguidos del porqué:

1. Número de objetos generados: fue uno de los primeros campos que tuvimos en cuenta, pero lo vimos poco relevante puesto que hacer muchos objetos no representa que esté mejor representado y genera valores tan dispares y aleatorios que no siguen un patrón del cual podamos obtener algún valor o dato relevante.

2. Profundidad: Esta métrica si que sería necesaria en un estudio con mayor capacidad económica que permitiese utilizar a los grandes LLMs actuales, que se centra en la cantidad de detalles generados por el LLM, pero realmente en nuestras pruebas hay dos principales problemas que hacen que sea imposible obtener resultados relevantes con ella, la escasa cantidad de *assets* que provocan un cambio masivo de las palabras sinónimas por lo que detalles se perderían y se convertirían en incongruencias (por ejemplo dentro de una casa se podrían generar sillas, sofás, butacas... nuestro programa sustituiría la mayoría por bancos al ser el único *asset* relacionado) y segundo por el uso de LLMs pequeños como ya nombré, que son nuestra única posibilidad.

3. Tiempo (general) : Decidimos sustituir esta métrica de tiempo por una de tiempo del LLM puesto que el retardo de las conexiones con Unreal vienen atadas severamente a retardos en los servidores y aplicaciones y con nuestra capacidad de cómputo actual, la cual está completamente ocupada por la ejecución del LLM, dan datos muy dispares entre nuestro propio grupo, mientras que si el LLM no excede nuestra capacidad de cómputo el valor es más realista.

6.1. Objetivos

Los objetivos principales son obtener cuáles son los mejores LLMs para distintos contextos en nuestro proyecto, ver si todos son válidos y compararlos para obtener el LLM más versátil.

6.2. Casos de prueba

Hemos definido tres casos de prueba los cuales abarcan todo el espectro de generación de escenarios que se le puede pedir a un LLM.

1. Texto corto: En este caso queremos una respuesta rápida y que contenga todo lo pedido, puesto que es poco es de gran importancia que no olvide ningún detalle de la entrada del usuario, "Generate a city with buildings surrounding a street with cars".

2. Texto largo: En esta métrica buscamos llevar al límite la capacidad de no olvidar ningún caso de los planteados al LLM, le pedimos "Genera una ciudad, que tenga edificios rodeando una calle con bancos, repleta de coches parados en semáforos al aldo de una parada de autobuses/ "Generate a city, which has buildings surrounding a street with benches, full of cars stopped at traffic lights next to a bus stop".

3. Texto imaginativo : En este caso le pedimos al LLM que genere algo desde una descripción nada detallada, al estar centrado en el tema ciudad nuestras pruebas, le pedimos "Genera una ciudad/ "Generate a city".

6.3. Método de obtención de datos

Para la obtención de datos hemos aprovechado una de las iteraciones anteriores del trabajo, la cual utilizaba el propio terminal de Python para mandar al LLM la descripción a generar, para medir el tiempo comprobamos el momento en el cual un *print* lo reproduce por el terminal y el porcentaje de cubos y la precisión se valoran posteriormente en la generación de Unreal. De modo que tras una ejecución hemos podido recolectar todos los datos para posteriormente poderlos volcar y reflejar en tablas y gráficos como los que mostraremos después.

6.4. Método de volcado de datos

Para el volcado de datos creamos un pequeño programa el cual nos permite añadir a un fichero *Excel* las filas con un identificador, los datos a medir y una descripción aclarativa para saber la razón de la nota, lo que sumado a un proceso de de filtrado y ajuste de la tabla nos genera directamente una gráfica . El filtrado se basa en eliminar los primeros casos si se ha decidido cambiar algo para los siguientes, obteniendo así resultados sobre datos provenientes de una ejecución en las mismas condiciones exactas. El ajuste se refiere al cambio en la tabla para que su representación gráfica sea lo más visual posible, como eliminar el nombre de la ejecución y sustituirlo por un número de iteración.

6.5. Resumen del proceso

Se realizaron un número arbitrario de pruebas divididas por LLM y categoría, 25 pruebas por categoría, de las cuales se han eliminado las primeras en las cuales buscábamos solucionar pequeños errores y en la primera se han tenido que eliminar 4 nuevos casos al encontrar un error importante detectado posteriormente en la transcripción, por eso consta de 16 y el resto de 20.

Hay 3 categorías definidas que juegan con distintos aspectos de la generación de código centradas en el input que reciben, un texto corto y concreto, un texto largo y concreto y un texto corto referente a que la LLM genere de una forma más creativa dejando le una completa libertad a la hora de interpretar el mensaje. Y posteriormente generando las tablas y los gráficos pertinentes.

En la gráfica aparecerán los valores de cada ejecución junto a una linea recta la cual representa la media del conjunto para poder observar las desviaciones de una forma mucho más visual.

6.6. Llama 3.2 (8gb)

6.6.1. Resultados y valores obtenidos al procesar un texto corto y concreto

Tabla 6.1: Llama 3.2 corto (Parte 1)

Iteración	%Precisión	Tiempo(s)	%Cubos	Errores
1	100	20	0	0
2	100	10	0	0
3	100	30	0	0
4	80	60	20	0
5	80	10	0	0
6	100	20	0	0
7	100	60	0	0
8	100	30	0	0
9	25	10	0	0
10	100	90	40	0
11	100	30	0	0
12	100	10	0	0
13	100	20	0	0
14	100	30	10	0
15	100	10	0	0
16	100	15	0	0

Tabla 6.2: Llama 3.2 corto (Parte 2 - Valores medios)

Iteración	MediaP	MediaTp	Media %C	MediaErr
1	92.8125	28.4375	4.375	0
2	92.8125	28.4375	4.375	0
3	92.8125	28.4375	4.375	0
4	92.8125	28.4375	4.375	0
5	92.8125	28.4375	4.375	0
6	92.8125	28.4375	4.375	0
7	92.8125	28.4375	4.375	0
8	92.8125	28.4375	4.375	0
9	92.8125	28.4375	4.375	0
10	92.8125	28.4375	4.375	0
11	92.8125	28.4375	4.375	0
12	92.8125	28.4375	4.375	0
13	92.8125	28.4375	4.375	0
14	92.8125	28.4375	4.375	0
15	92.8125	28.4375	4.375	0
16	92.8125	28.4375	4.375	0

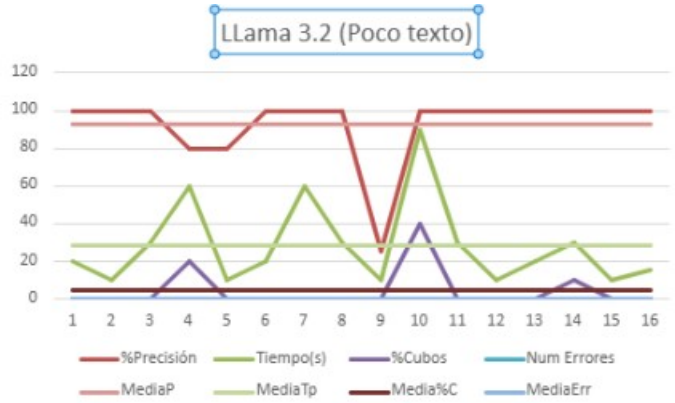

Figura 6.1: Gráfica obtenida de las 6.1 y 6.2

En este caso de prueba se buscaba que el LLM rinda al 100 % en todos campos puesto que la entrada era muy concreta y corta (generar una ciudad con un edificio al lado de una carretera repleta de coches). La métrica en la que nos queremos centrar con estas pruebas es el porcentaje de cubos, puesto que queremos ver que generando cosas pequeñas lo hace correctamente.

Como se puede observar en los datos de la tabla y en la representación gráfica la mayoría de los datos son muy estables, el número de errores permanece en 0 en las 16 pruebas, es muy complicado que genere un error de ejecución por todo el filtrado y apoyo que se le da a esa parte, y menos con esta LLM, ya que ha sido una de las más utilizadas en el desarrollo pre-pruebas, el porcentaje de cubos es prácticamente proporcional al número de objetos que se generaron y a su vez suele coincidir con picos de tiempo, pero se mantiene muy estable en 0, lo cual es normal por la clase de entrada que añadimos en esta prueba, al igual pasa con la precisión al estar concretado la entrada, casi nunca se sale de los objetos que debe generar muy pocas veces se olvida de alguno, y se destaca un punto muy bajo en el cual se generó unicamente un edificio y objetos en su interior no requeridos por el usuario (Dato sacado de las descripciones cortas del mini programa que añade información a las tablas).

Ya por último cabe destacar la irregularidad en el tiempo que ya no es solo por la salida aunque si que suele estar relacionado, a mayor cantidad de objetos mayor tiempo, sino que también generando salidas parecidos variaba el tiempo unos 10 segundos.

En conclusión, los datos son muy positivos y estables a excepción del tiempo, el cual igualmente es relativamente corto al no haber superado en ningún momento el minuto de ejecución el LLM, para este tipo de generación Llama 3.2 es más que suficiente.

6.6.2. Resultados y valores obtenidos al procesar un texto largo y concreto

Tabla 6.3: Llama 3.2 largo - Tabla 1 (Parte 1)

Iteración	Precisión	Tiempo	%Cubos	Errores
1	90	40	20	0
2	80	60	20	0
3	60	90	10	0
4	50	50	10	0
5	90	60	0	0
6	50	60	20	0
7	80	90	40	0
8	60	40	10	0
9	90	40	0	0
10	80	50	10	0
11	80	90	40	0
12	60	60	20	0
13	90	40	10	0
14	90	50	40	0
15	60	60	10	0
16	80	60	20	0
17	50	40	40	0
18	80	40	40	0
19	90	90	10	0
20	80	50	20	0

Tabla 6.4: Llama 3.2 largo - Tabla 1 (Parte 2 - Medias)

Iteración	**MediaP**	**MediaTp**	**Media %C**	**MediaErr**
1	74,5	58	19,5	0
2	74,5	58	19,5	0
3	74,5	58	19,5	0
4	74,5	58	19,5	0
5	74,5	58	19,5	0
6	74,5	58	19,5	0
7	74,5	58	19,5	0
8	74,5	58	19,5	0
9	74,5	58	19,5	0
10	74,5	58	19,5	0
11	74,5	58	19,5	0
12	74,5	58	19,5	0
13	74,5	58	19,5	0
14	74,5	58	19,5	0
15	74,5	58	19,5	0
16	74,5	58	19,5	0
17	74,5	58	19,5	0
18	74,5	58	19,5	0
19	74,5	58	19,5	0
20	74,5	58	19,5	0

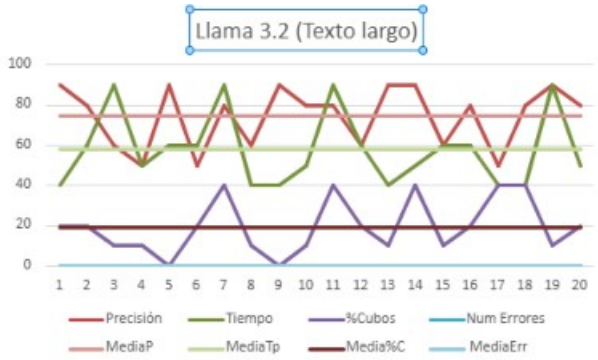

Figura 6.2: Gráfica obtenida de las 6.3 y 6.4

En este caso de prueba se busca poner al límite la capacidad de retención de los datos a representar del LLM con una descripción medianamente larga y detallada. El apartado más importante de esta sección es la precisión, puesto que buscamos que genere todo lo que hemos pedido.

Viendo la representación en las tablas y la gráfica no ha ocurrido ningún error en las ejecuciones y el resto de valores son muy dispares en sus propias categorías aunque sin desviaciones altas siempre alrededor de la media, el porcentajes de cubos es más alto de lo esperado puesto que no debería haberse salido mucho del guión, pero a su vez al darle una descripción tan larga es posible que le de más espacio a la imaginación o rellenar para dejar una salida más completa, la cual realmente como se puede observar nunca llega al 100 % de precisión, el tiempo medio ronda el minuto como cabía esperar es bastante elevado pero teniendo en cuenta la gran cantidad de objetos generados es el precio que cabía esperar pagar.

En conclusión, más allá de la pequeña decepción respecto a la precisión, la cual sigue siendo alta pero al estar completamente definido lo que se quería generar, era el punto más clave, y eso requería un resultado lo más óptimo posible, por el resto de parámetros muy correctos, tiempo elevado pero era lo esperado, bastantes cubos que no es sorprendente puesto que la descripción lleva a crear algo mucho más elaborado y eso lleva a detalles sin assets y ningún error lo cual confirma que la generación y su proceso de filtrado funcionan excelentemente.

6.6.3. Resultados y valores obtenidos al procesar un texto imaginativo

Tabla 6.5: Llama 3.2 imaginativo (Parte 1)

Iteración	**%Precisión**	**Tiempo(s)**	**%Cubos**	**Errores**
1	100	40	0	0
2	70	60	0	0
3	90	40	0	0
4	100	120	20	0
5	90	90	0	0
6	100	40	0	0
7	90	60	0	0
8	60	90	0	0
9	80	60	0	0
10	100	50	40	0
11	100	30	0	0
12	90	60	0	0
13	100	45	0	0
14	100	30	10	0
15	100	40	0	0
16	90	60	0	0
17	100	100	0	0
18	100	30	0	0
19	80	30	0	0
20	100	60	0	0

Tabla 6.6: Llama 3.2 imaginativo (Parte 2 - Valores medios)

Iteración	**MediaP**	**MediaTp**	**Media %C**	**MediaErr**
1	92	56.75	3.5	0
2	92	56.75	3.5	0
3	92	56.75	3.5	0
4	92	56.75	3.5	0
5	92	56.75	3.5	0
6	92	56.75	3.5	0
7	92	56.75	3.5	0
8	92	56.75	3.5	0
9	92	56.75	3.5	0
10	92	56.75	3.5	0
11	92	56.75	3.5	0
12	92	56.75	3.5	0
13	92	56.75	3.5	0
14	92	56.75	3.5	0
15	92	56.75	3.5	0
16	92	56.75	3.5	0
17	92	56.75	3.5	0
18	92	56.75	3.5	0
19	92	56.75	3.5	0
20	92	56.75	3.5	0

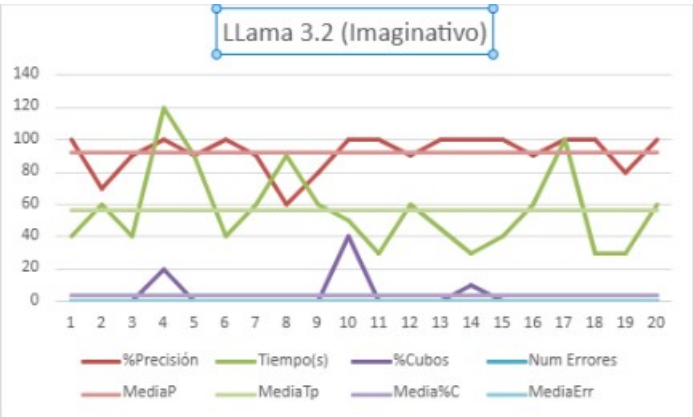

Figura 6.3: Gráfica obtenida de las 6.5 y 6.6

En este caso de prueba se busca ver el potencial creativo del LLM, desde una entrada que pedía una representación de una ciudad con completa libertad, lo que a su vez provoca que el parámetro de precisión sea más subjetivo. La métrica más relevante aquí es el tiempo de ejecución.

Como se representa en los datos de la tabla y en la representación gráfica en este caso se observan valores muy volátiles y dispares en la precisión y el tiempo, lo cual era de esperar, puesto que la precisión es más subjetiva y a su vez la generación es completamente libre, el tiempo medio ronda el minuto que ya es un tiempo considerablemente alto llegando a tardar 2 minutos en una ocasión y superar el minuto en múltiples ocasiones más. Los errores siguen siendo nulos y la cantidad de cubos también es baja lo cual es bastante inesperado, puede que el valor de sustitución de valores por sus sinónimos sea alto o que la semántica relacionada a las ciudades simplemente está agrupada en pocos campos semánticos y para los cuales tenemos al menos un *asset*.

En conclusión, la sorpresa mayor es la generación de cubos puesto que era de esperar un valor medio más elevado, por el resto valores esperables, un tiempo medio elevado, una precisión cercana al máximo ya que es difícil generar cosas que descuadren mayormente y la nula aparición de errores. Llama 3.2 es bastante útil para la generación creativa, el mayor problema es que ese tiempo únicamente para el cómputo de una salida sencilla es muy elevado lo cual puede causar un rechazo a su uso cotidiano.

6.7. Phi 2 orange (3-4 gb)

En este LLM hay que tener en cuenta una serie de objeciones que hacen que la comparativa no sea equivalente al 100 %, hemos tenido que modificar el programa para correrlo desde un *launcher* de consola externo, koboldcpp, el cual tiene parámetros distintos, uno de ellos es el número máximo de caracteres que determina prácticamente el tiempo de ejecución, hemos probado para poner el número de parámetros que se acerque a la media de tiempo de las otras métricas, lo cual también provoca errores como veremos a continuación.

6.7.1. Resultados y valores obtenidos al procesar un texto corto y concreto

Tabla 6.7: Phi 2 corto (Parte 1 - Valores individuales)

Iteración	**Precisión**	**Tiempo**	**%Cubos**	**Num Errores**
1	0	40	100	1
2	100	40	20	0
3	0	30	100	1
4	0	35	100	1
5	0	40	100	1
6	80	35	33	0
7	0	35	100	1
8	0	35	100	1
9	80	30	25	0
10	0	45	100	1
11	0	40	20	0
12	0	30	100	1
13	0	30	100	1
14	0	45	100	1
15	100	35	20	0
16	0	30	100	1
17	80	45	33	0
18	0	30	100	0
19	0	35	100	1
20	50	40	50	0

Tabla 6.8: Phi 2 corto (Parte 2 - Valores medios)

Iteración	**MediaP**	**MediaTp**	**Media %C**	**MediaErr**
1	24.5	36.25	75.05	0.6
2	24.5	36.25	75.05	0.6
3	24.5	36.25	75.05	0.6
4	24.5	36.25	75.05	0.6
5	24.5	36.25	75.05	0.6
6	24.5	36.25	75.05	0.6
7	24.5	36.25	75.05	0.6
8	24.5	36.25	75.05	0.6
9	24.5	36.25	75.05	0.6
10	24.5	36.25	75.05	0.6
11	24.5	36.25	75.05	0.6
12	24.5	36.25	75.05	0.6
13	24.5	36.25	75.05	0.6
14	24.5	36.25	75.05	0.6
15	24.5	36.25	75.05	0.6
16	24.5	36.25	75.05	0.6
17	24.5	36.25	75.05	0.6
18	24.5	36.25	75.05	0.6
19	24.5	36.25	75.05	0.6
20	24.5	36.25	75.05	0.6

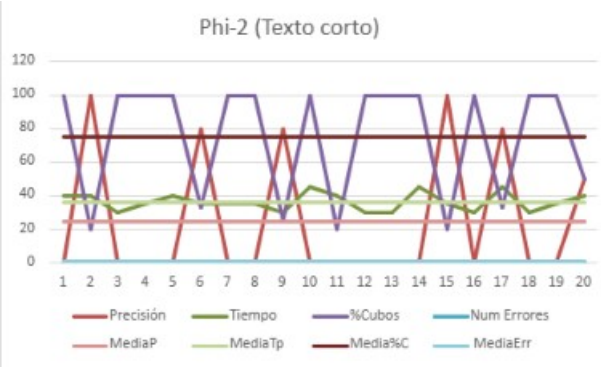

Figura 6.4: Gráfica obtenida de las 6.7 y 6.8

La prueba ha sido bastante desastrosa, es la más sencilla y solo ha conseguido ejecutarse correctamente 3 de cada 10 veces, para las cuales si que obtenía resultados decentes, el tiempo de ejecución como comentamos previamente es muy estable, pero el resto de los campos no son realmente valorables debido al gran número de errores, los cuales se suelen dividir en dos grupos, creación de cosas que no son un JSON, a veces creaba código en java para los objetos, otras veces una especie de reglas que se deberían aplicar al JSON con frases como (two adjacent buildings should not face each other, dos edificios adyacentes no deben mirarse), y el otro tipo de error, ya mencionado, al tener un tamaño fijo de caracteres algunas salidas quedan a medias por lo que no pueden ser procesadas, se podría probar un mecanismo que aumente la cantidad de caracteres permitidos si falla, pero viendo la cantidad de fallos no es viable, si solo fuesen errores de este tipo se hubiese probado.

Concluyendo esta prueba solo nos deja claro que es un LLM con un uso del tiempo muy consistente pero al menos a la hora de generar JSON no es el más adecuado a pesar de no ser muy pequeño.

6.7.2. Resultados y valores obtenidos al procesar un texto largo y concreto

Tabla 6.9: Phi 2 largo (Parte 1 - Valores individuales)

Iteración	Precisión	Tiempo	%Cubos	Num Errores
1	0	70	100	1
2	0	55	100	1
3	0	70	100	1
4	0	70	100	1
5	0	60	100	1
6	0	60	100	1
7	0	70	100	1
8	80	55	0	0
9	0	70	100	1
10	0	60	100	1
11	0	70	100	1
12	0	70	100	1
13	0	60	100	1
14	0	55	100	1
15	60	60	20	0
16	0	55	100	1
17	0	70	100	1
18	0	60	100	1
19	0	55	100	1
20	0	60	100	1

Tabla 6.10: Phi 2 largo (Parte 2 - Valores medios)

Iteración	**MediaP**	**MediaTp**	**Media %C**	**MediaErr**
1	7	62.75	91	0.9
2	7	62.75	91	0.9
3	7	62.75	91	0.9
4	7	62.75	91	0.9
5	7	62.75	91	0.9
6	7	62.75	91	0.9
7	7	62.75	91	0.9
8	7	62.75	91	0.9
9	7	62.75	91	0.9
10	7	62.75	91	0.9
11	7	62.75	91	0.9
12	7	62.75	91	0.9
13	7	62.75	91	0.9
14	7	62.75	91	0.9
15	7	62.75	91	0.9
16	7	62.75	91	0.9
17	7	62.75	91	0.9
18	7	62.75	91	0.9
19	7	62.75	91	0.9
20	7	62.75	91	0.9

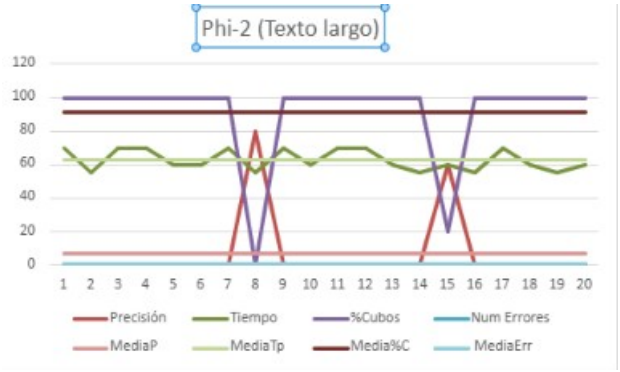

Figura 6.5: Gráfica obtenida de las 6.9 y 6.10

Continuando con la dinámica de pruebas terribles tenemos los datos de esta, que con un 10

Con esta prueba nos confirma ya un poco el LLM que no es por el input o la cantidad de caracteres, sino que él mismo es el problema puesto que no estará entrenado para trabajar correctamente con JSON.

6.7.3. Resultados y valores obtenidos al procesar un texto imaginativo

Tabla 6.11: Phi 2 imaginativo (Parte 1 - Valores individuales)

Iteración	Precisión	Tiempo	%Cubos	Num Errores
1	0	90	100	1
2	0	80	100	1
3	0	90	100	1
4	0	100	100	1
5	0	90	100	1
6	0	80	100	1
7	0	80	100	1
8	0	80	100	1
9	0	90	100	1
10	0	80	100	1
11	0	80	100	1
12	0	90	100	1
13	0	90	100	1
14	0	80	100	1
15	0	90	100	1
16	0	90	100	1
17	0	80	100	1
18	0	80	100	1
19	0	90	100	1
20	0	100	100	1

Tabla 6.12: Phi 2 imaginativo (Parte 2 - Valores medios)

Iteración	**MediaP**	**MediaTp**	**Media %C**	**MediaErr**
1	0	86.5	100	1
2	0	86.5	100	1
3	0	86.5	100	1
4	0	86.5	100	1
5	0	86.5	100	1
6	0	86.5	100	1
7	0	86.5	100	1
8	0	86.5	100	1
9	0	86.5	100	1
10	0	86.5	100	1
11	0	86.5	100	1
12	0	86.5	100	1
13	0	86.5	100	1
14	0	86.5	100	1
15	0	86.5	100	1
16	0	86.5	100	1
17	0	86.5	100	1
18	0	86.5	100	1
19	0	86.5	100	1
20	0	86.5	100	1

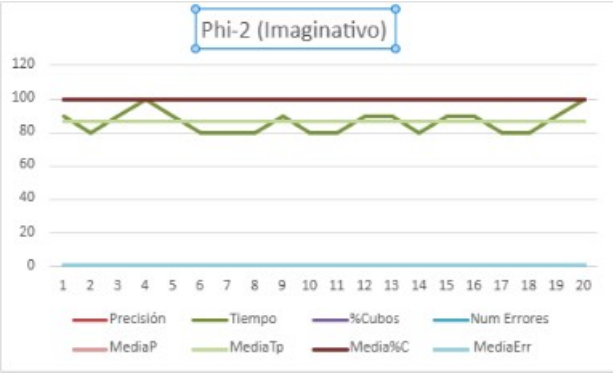

Figura 6.6: Gráfica obtenida de 6.11 y 6.12

No hay absolutamente nada de lo que se pueda comentar, hemos probado a ir aumentando el espacio hasta que aproximadamente cada salida tardase minuto y medio, lo que es el límite máximo que le dabamos para que fuese comparable, y no ha conseguido sacar nada potable, principalmente porque intenta hacer el JSON absurdamente grande y detallado y poniendo 10k caracteres que tardaba cada ejecución casi 10 minutos ha fallado 3 veces más por lo que simplemente este LLM no sería un buen candidato para trabajar en esto.

Tras todas las pruebas de analisis concluimos demostrando que realmente no es un buen LLM para crear JSON.

6.8. Llama 3.2 (1gb)

Visto el problema que tuvimos en la métrica anterior nos decantamos por utilizar a un viejo conocido nuestro, un LLM que es la versión reducida del primero que hemos visto, el cual ya probamos por los problemas que suponía tener ocupada toda la capacidad de computo en el ordenador para varios integrantes de nuestro grupo. A partir de ahora nos referiremos a él como LlamaLite para distinguirlo de la versión de 8gb.

6.8.1. Resultados y valores obtenidos al procesar un texto corto y concreto

Tabla 6.13: Llamalite corto (Parte 1 - Valores individuales)

Iteración	**Precisión**	**Tiempo**	**%Cubos**	**Num Errores**
1	100	20	0	0
2	80	15	20	0
3	100	20	0	0
4	100	30	0	0
5	100	20	0	0
6	100	15	0	0
7	100	20	0	0
8	100	20	20	0
9	80	15	0	0
10	100	20	0	0
11	100	25	0	0
12	100	20	0	0
13	100	15	0	0
14	100	15	20	0
15	100	20	0	0
16	100	15	0	0
17	80	20	40	0
18	100	15	20	0
19	100	15	0	0
20	100	25	0	0

Tabla 6.14: Llamalite corto (Parte 2 - Valores medios)

Iteración	**MediaP**	**MediaTp**	**Media %C**	**MediaErr**
1	97	19	6	0
2	97	19	6	0
3	97	19	6	0
4	97	19	6	0
5	97	19	6	0
6	97	19	6	0
7	97	19	6	0
8	97	19	6	0
9	97	19	6	0
10	97	19	6	0
11	97	19	6	0
12	97	19	6	0
13	97	19	6	0
14	97	19	6	0
15	97	19	6	0
16	97	19	6	0
17	97	19	6	0
18	97	19	6	0
19	97	19	6	0
20	97	19	6	0

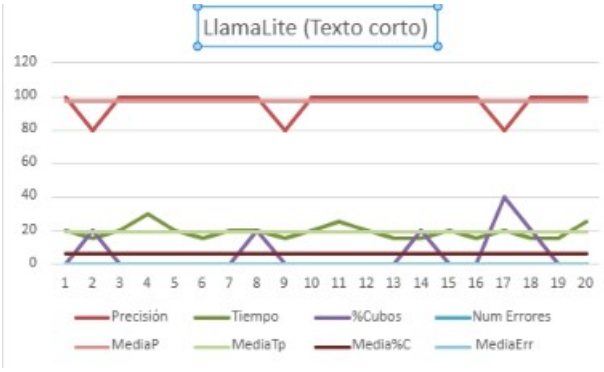

Figura 6.7: Gráfica obtenida de las 6.13 y 6.14

Como se puede observar en los datos obtenidos ha sido una ejecución prácticamente perfecta, sin ningún error, con un tiempo de ejecución medio muy bajo, 19 segundos es realmente bajo, un porcentaje de cubos que no supera el 50 % en ninguno de los pocos casos en los que tan siquiera aparece un cubo,que aparecen uno de cada cuatro casos, y una precisión prácticamente infalible, un dato que apuntamos es que a diferencia de su versión de mayor capacidad los resultados de este LLM eran muy similares entre sí, y puede que eso haya beneficiado en gran medida al LLM en este apartado.

Con los resultados obtenidos desde este LLM podríamos concluir que para una generación de una zona pequeña y concreta,(al menos con la temática ciudades como es la nuestra), es una opción muy acertada.

6.8.2. Resultados y valores obtenidos al procesar un texto largo y concreto

Tabla 6.15: Resultados por iteración (Parte 1 - Valores individuales)

Iteración	Precisión	Tiempo	%Cubos	Num Errores
1	80	45	20	0
2	80	40	25	0
3	100	60	20	0
4	60	50	50	0
5	80	50	20	0
6	60	45	33	0
7	80	70	20	0
8	100	50	20	0
9	60	35	20	0
10	80	50	20	0
11	60	50	0	0
12	60	60	33	0
13	80	50	20	0
14	100	60	0	0
15	80	40	40	0
16	60	45	20	0
17	100	50	40	0
18	100	50	20	0
19	60	50	20	0
20	80	70	0	0

Tabla 6.16: Resultados por iteración (Parte 2 - Valores medios)

Iteración	**MediaP**	**MediaTp**	**Media %C**	**MediaErr**
1	78	51	22.05	0
2	78	51	22.05	0
3	78	51	22.05	0
4	78	51	22.05	0
5	78	51	22.05	0
6	78	51	22.05	0
7	78	51	22.05	0
8	78	51	22.05	0
9	78	51	22.05	0
10	78	51	22.05	0
11	78	51	22.05	0
12	78	51	22.05	0
13	78	51	22.05	0
14	78	51	22.05	0
15	78	51	22.05	0
16	78	51	22.05	0
17	78	51	22.05	0
18	78	51	22.05	0
19	78	51	22.05	0
20	78	51	22.05	0

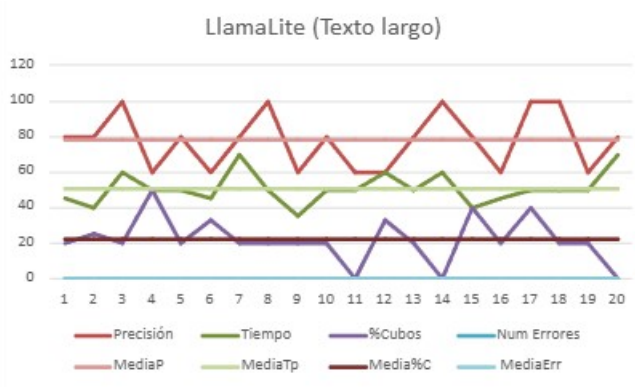

Figura 6.8: Gráfica obtenida de 6.15 y 6.16

Los datos que más nos sorprenden en esta representación referente al texto largo es la bajada de nivel en la precisión, en muy pocos ejemplos mostraba una escena tal cual la pedíamos, se olvidaba principalmente de detalles pero en algunas ejecuciones como la 6 se olvidaba de gran parte del texto, la mayoría de las veces son detalles pequeños que es lo que implica el 80 % de precisión (por ejemplo no ponía múltiples cosas de una palabra en plural, que es el error de precisión más frecuentado), pero en otras ocasiones se quedaba con una parte del texto y trabajaba en ella como si el resto de la entrada de datos ni le hubiera llegado, de ahí multiples bajones. El tiempo destaca también un poco para mal, ya que debería ser el punto fuerte del LLM y no destaca notoriamente para este tipo de entradas, generalmente aparecen cubos en prácticamente todas las ejecuciones, y sigue sin generar errores lo que es realmente importante.

Con los resultados obtenidos desde este LLM podríamos concluir que para una generación de una zona pequeña y concreta,(al menos con la temática ciudades como es la nuestra), es una opción muy acertada.

6.8.3. Resultados y valores obtenidos al procesar un texto imaginativo

Tabla 6.17: Llamalite imaginativo (Parte 1 - Valores individuales)

Iteración	Precisión	Tiempo	%Cubos	Num Errores
1	80	50	40	0
2	80	35	0	0
3	50	50	25	0
4	100	35	0	0
5	80	40	0	0
6	80	35	33	0
7	80	40	0	0
8	80	35	0	0
9	80	40	40	0
10	80	50	0	0
11	80	40	20	0
12	60	40	40	0
13	80	35	20	0
14	80	35	0	0
15	100	40	0	0
16	80	35	20	0
17	100	40	20	0
18	60	55	20	0
19	60	35	25	0
20	80	40	20	0

Tabla 6.18: Llamalite imaginativo (Parte 2 - Valores medios)

Iteración	**MediaP**	**MediaTp**	**Media %C**	**MediaErr**
1	78.5	40.25	16.15	0
2	78.5	40.25	16.15	0
3	78.5	40.25	16.15	0
4	78.5	40.25	16.15	0
5	78.5	40.25	16.15	0
6	78.5	40.25	16.15	0
7	78.5	40.25	16.15	0
8	78.5	40.25	16.15	0
9	78.5	40.25	16.15	0
10	78.5	40.25	16.15	0
11	78.5	40.25	16.15	0
12	78.5	40.25	16.15	0
13	78.5	40.25	16.15	0
14	78.5	40.25	16.15	0
15	78.5	40.25	16.15	0
16	78.5	40.25	16.15	0
17	78.5	40.25	16.15	0
18	78.5	40.25	16.15	0
19	78.5	40.25	16.15	0
20	78.5	40.25	16.15	0

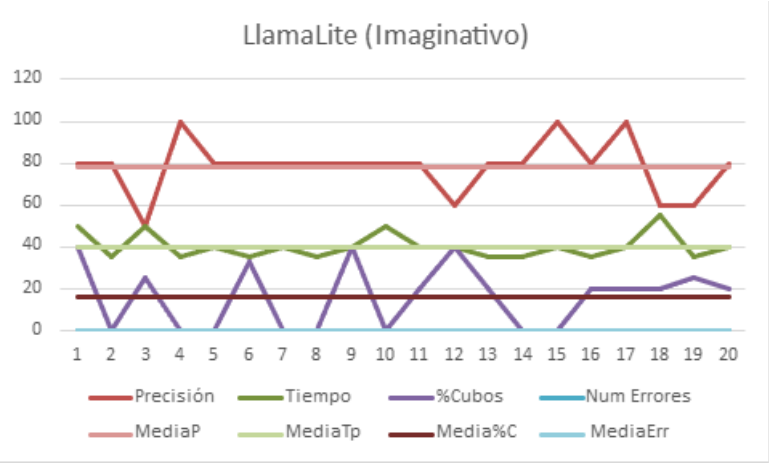

Figura 6.9: Gráfica obtenida de 6.17 y 6.18

En los resultados para la generación imaginativa volvemos a ver destacar su punto fuerte en esta LLM el tiempo es realmente bajo de media para un proceso que suele ser más tardado, genera normalmente escenarios muy acertados aunque a veces se va un poco por las ramas, generando a partir de una idea que se crea el LLM a sí mismo, lo cual en un proceso creativo es hasta beneficioso, pero aquí debemos "penalizar" puesto que la idea principal es generar una ciudad, o a veces se queda muy corto como pasa en la ejecución 3, en la cual solo generó unas casas y ya. Cabe destacar que sigue la tónica en los otros 2 campos restantes, porcentaje de cubos bastante bajo y ningún error.

Visto lo visto son muy buenos números para una categoría que a una LLM pequeña le debería costar destacar sobre modelos más grandes.

6.9. Conclusión final

Una vez todos los datos han sido analizados punto por punto es momento de dejar una puntuación y resumen sobre este apartado en general. Más allá de los problemas que no son directos de las LLMs que ya hemos mencionado, los cuales afectan a todas por igual y se han intentado no tener en cuenta para una valoración lo más justa posible, hay dos claros ganadores y un claro fracaso, Llama en sus dos versiones ha arrasado a Phi 2 en todos los campos, y entre estas dos no hay una que indiscutiblemente sea mejor. Si nos basamos únicamente en los resultados es posible que la versión de 8gb haya dado un mejor rendimiento y no se haya quedado muy descolgada en el tiempo de ejecución, por lo que si tienes un equipo capaz de utilizar esta sin ningún problema sería nuestro caballo ganador, pero aquí es donde nosotros nos decidimos por la versión de 1gb (Llamalite como lo hemos estado llamando) porque previamente a tener los servidores que nos ayudaron con la carga de cómputo, este modelo nos permitió avanzar sin una perdida de fidelidad importante lo que es un punto muy positivo, que en nuestra opinión es fundamental.

Capítulo 7

Evaluación con usuarios del sistema de generación

Para poder comprobar la utilidad real que pueda tener nuestra aplicación, va a ser sometida a evaluación por personas ajenas a su desarrollo. Este procedimiento tiene como objetivo obtener una visión objetiva sobre la experiencia de uso, la funcionalidad y usabilidad del sistema desde la perspectiva de los usuarios finales.

La retroalimentación obtenida proporcionada por los participantes permitirá identificar fortalezas, debilidades y posibles aspectos a mejorar, tanto a nivel técnico como a nivel de experiencia de usuario. Para ello, se diseñará un conjunto de tareas representativas y se aplicará una metodología de evaluación consistente en incluir cuestionarios, observaciones directas o análisis de métricas de uso.

En este capítulo se describen los detalles del proceso de valoración: el perfil de los usuarios seleccionados, el diseño de la evaluación y un análisis de los resultados obtenidos. Todo ello con el fin de garantizar que el producto final responda adecuadamente a las necesidades de sus potenciales usuarios.

7.1. Descripción de la evaluación

Seleccionada un grupo de 10 personas ajenas al proyecto, se les proporcionará una breve explicación sobre cuál es la función de la herramienta y su uso. Tras esta pequeña explicación se les asignará la siguiente tarea de diseñar un entorno de manera manual y luego diseñarlo de manera automática con el uso de la LLM, en este caso una ciudad. Se les realizará una demostración de como funcionan ambos métodos y luego se le pedirá que comiencen con la prueba. El objetivo de este experimento es ver la experiencia general de cada usuario usando la aplicación, además de su duración. Durante la realización del experimento se tomarán notas de lo que cada usuario a encontrado más sencillo y más complicado, además, para poder evaluar el tiempo tardado por cada tarea, los encuestados serán cronometrados durante la realización de las pruebas. También se tomaran métricas cualitativas en un cuestionario que los usuarios tendrán que responder al final de la prueba.

Una vez acabadas las pruebas se le pedirá al encuestado que responda a un cuestionario con las siguientes preguntas:

- ¿Crees que has ahorrado tiempo usando la herramienta en vez de haciéndolo a mano?
- ¿Crees que la herramienta te ha ahorrado trabajo o esfuerzo mental?
- ¿Consideras la herramienta sencilla de usar?
- ¿Te ha ayudado a concentrarte más en el concepto y menos en el detalle?
- ¿Cómo de precisa consideras que ha sido la herramienta respecto a tu descripción?
- ¿Usarías la herramienta en un desarrollo real?
- En general, valora la herramienta.
- Y por último una pregunta opcional, ¿Se te ocurre alguna mejora para la herramienta?

7.2. Evaluación cualitativa del proceso de diseño de usuarios

En esta sección se relata el desarrollo de las pruebas realizadas con usuarios. Para la evaluación general del proceso de diseño las métricas cuantitativas son limitadas debido a que no hay una manera concreta de medir un "buen diseño" o un "mal diseño", en consecuencia, la evaluación cualitativa resulta muy útil, pudiendo con ellas obtener las opiniones y valoraciones de los encuestados para después valorar las fortalezas y debilidades del sistema.

7.2.1. Usuario 1

7.2.1.1. Contexto personal

El usuario 1 es un estudiante de tercer curso de Ingeniería de Minas en la Escuela Técnica Superior de Ingenieros de Minas y Energía de la Universidad Politécnica, entre sus aficiones e intereses, están la geología, los juegos de cartas y los videojuegos. A pesar de tener experiencia jugando estos últimos, el concepto de desarrollo de niveles es nuevo para él, además, a pesar de haber usado LLMs alguna vez no tiene tampoco mucha experiencia con ellas.

7.2.1.2. Desarrollo de un entorno de forma manual

El usuario 1 no encontró mucha dificultad en el manejo de Unreal y fue capaz de realizar su escenario de manera eficaz.

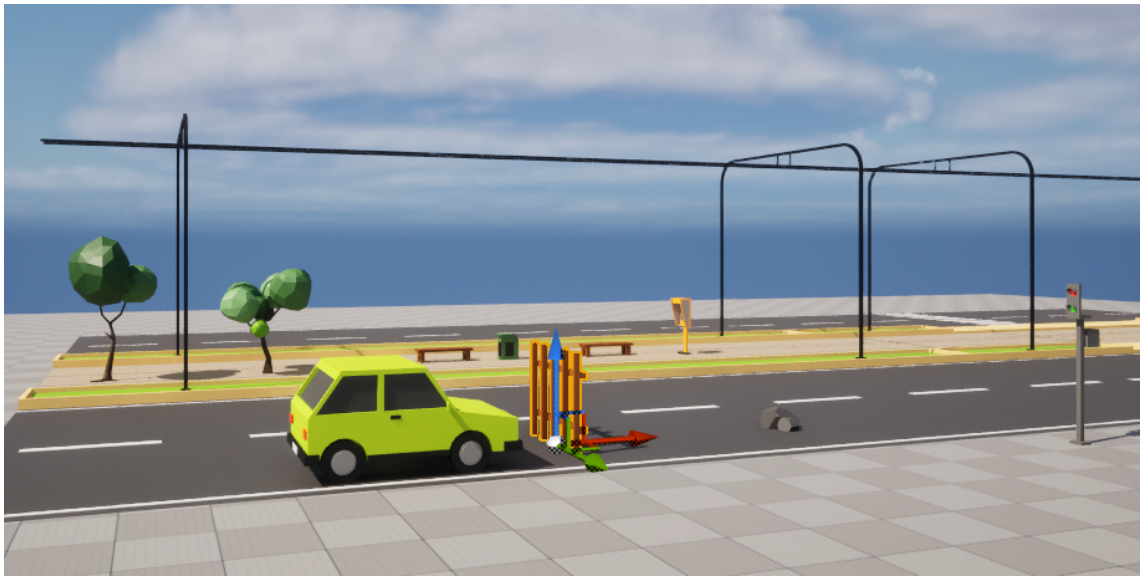

Figura 7.1: Entorno desarrollado manualmente por el usuario 1

7.2.1.3. Desarrollo de un entorno con LLM

Con el uso de la LLM, el usuario 1 fue capaz de recrear su entorno con un grado de precisión aceptable, siendo la LLM capaz de colocar el coche en la carretera, el semáforo y los árboles desplazados del espacio entre los carriles.

Figura 7.2: Entorno desarrollado usando la herramienta por el usuario 1

7.2.1.4. Obtención de métricas y comentarios

El usuario 1 consideró la aplicación cómo funcional y cree que con ciertas mejoras sería muy útil para el diseño de niveles. En cuanto a las métricas, podemos observar que tardó 3 minutos y 25 segundos en realizar el entorno a mano y 3 minutos 47 segundos en realizarlo con la LLM, restando el tiempo de procesamiento de peticiones.

El usuario 1 optó por una estrategia de realizar múltiples peticiones pequeñas para construir su nivel poco a poco en vez de una gran petición para según sus palabras: “No saturar a la máquina”.

7.2.2. Usuario 2

7.2.2.1. Contexto personal

El usuario 2 es un estudiante de segundo curso de Ingeniería Aeroespacial por la Universidad Politécnica, entre sus aficiones e intereses, están la música, las artes y la ingeniería. No es un usuario frecuente de LLMs ya que las considera imprecisas aunque reconoce su utilidad, además el concepto de diseño de niveles es totalmente ajeno a él ya que no es un jugador ávido.

7.2.2.2. Desarrollo de un entorno de forma manual

El usuario 2 encontró el manejo de Unreal algo complicado de entender al principio, aún así una vez consiguió dominarlo fue capaz de crear la siguiente ciudad con una plaza y múltiples edificios con rapidez.

Figura 7.3: Entorno desarrollado manualmente por el usuario 2

7.2.2.3. Desarrollo de un entorno con LLM

Usando la LLM, el usuario 2 consiguió crear una pequeña plaza con varios edificios en tan solo unas pocas peticiones.

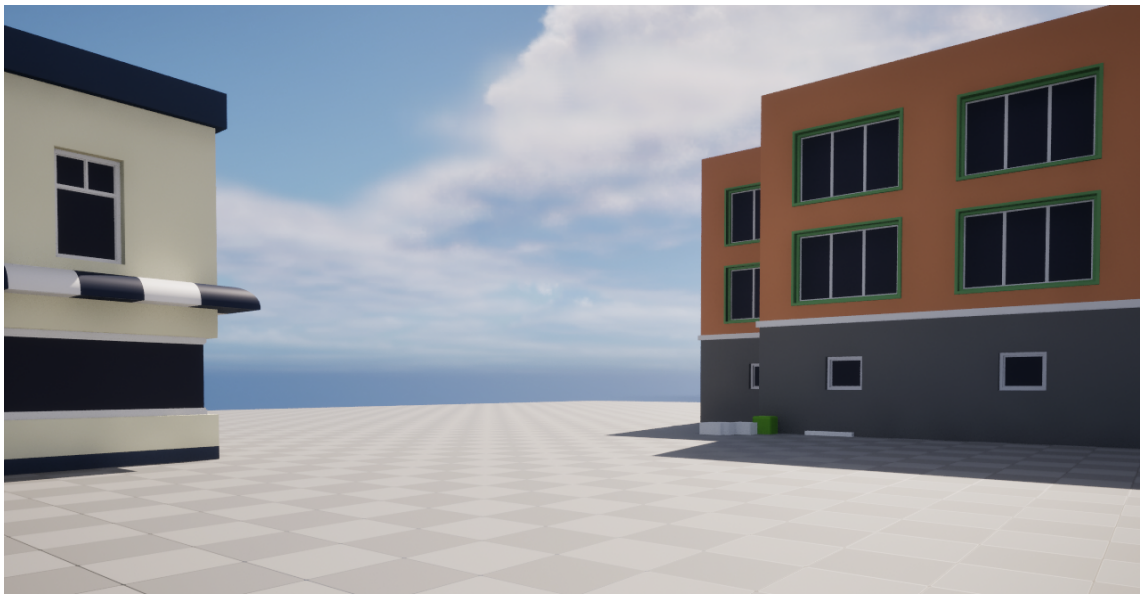

Figura 7.4: Entorno desarrollado usando la herramienta por el usuario 2

7.2.2.4. Obtención de métricas y comentarios

El usuario 2 consideró la aplicación cómo relativamente funcional y cree que aún tiene espacio para la mejora. En cuanto a las métricas, podemos observar que tardó 4 minutos y 38 segundos en realizar el entorno a mano y 3 minutos 51 segundos en realizarlo con la LLM, restando el tiempo de procesamiento de peticiones. Siendo más rápida la creación de niveles con la aplicación en este caso.

El usuario 2 optó por una estrategia de realizar múltiples peticiones pequeñas también, sin embargo echó en falta que se pudiera especificar por ejemplo, el color de los edificios ya que su intención era recrear con total fidelidad su diseño manual.

7.2.3. Usuario 3

7.2.3.1. Contexto personal

El usuario 3 es un estudiante de primer curso de Ingeniería Civil por la Universidad Politécnica, entre sus aficiones e intereses, están la música, el fútbol y los juegos de cartas. Es un usuario frecuente de LLMs ya que las considera muy útiles para realizar tareas fáciles, ha jugado bastantes videojuegos, así que entiende un poco de diseño de niveles.

7.2.3.2. Desarrollo de un entorno de forma manual

Aprendiendo rápido sobre el manejo de Unreal, creó una ciudad con múltiples edificios además de recrear una persecución policíaca por sus calles.

Figura 7.5: Entorno desarrollado manualmente por el usuario 3

7.2.3.3. Desarrollo de un entorno con LLM

En este caso la LLM tan solo fue capaz de recrear un edificio y un coche a su lado, usando una petición considerablemente grande.

Figura 7.6: Entorno desarrollado usando la herramienta por el usuario 3

7.2.3.4. Obtención de métricas y comentarios

El usuario 3 consideró que la aplicación aún tiene espacio para la mejora. En cuanto a las métricas, podemos observar que tardó 3 minutos y 9 segundos en realizar el entorno a mano y 4 minutos 7 segundos en realizarlo con la LLM, restando el tiempo de procesamiento de peticiones.

El usuario 3 optó por una estrategia de realizar una petición grande y detallada, y aunque no funcionó como se esperaba, la LLM si fue capaz de extraer múltiples datos sobre dicha petición.

7.2.4. Usuario 4

7.2.4.1. Contexto personal

El usuario 4 es un estudiante de segundo curso de Ingeniería del Automóvil por la Universidad de Nebrija, entre sus aficiones e intereses, están la música, el fútbol y la formula uno. Usa frecuentemente LLMs ya que las considera muy útiles para realizar tareas, ha jugado algunos videojuegos, pero no entiende sobre diseño de niveles.

7.2.4.2. Desarrollo de un entorno de forma manual

Aprendió con gran facilidad a usar el entorno de Unreal y consiguió crear su entorno en un tiempo récord.

Figura 7.7: Entorno desarrollado manualmente por el usuario 4

7.2.4.3. Desarrollo de un entorno con LLM

En este caso la LLM no fue capaz de generar lo que se le fue pedido, dejando tan solo un edificio sobre un suelo.

Figura 7.8: Entorno desarrollado usando la herramienta por el usuario 4

7.2.4.4. Obtención de métricas y comentarios

El usuario 4 fue crítico hacia la aplicación, encontró cierta frustración en el uso de la LLM, ya que no se mostraba lo que esperaba. En cuanto al tiempo, podemos observar que tardó 1 minuto y 40 segundos en diseñar el entorno de forma manual y 4 minutos 7 segundos realizándolo con la aplicación, restando el tiempo de procesamiento de peticiones.

El usuario 4 probó diferentes tipos de peticiones, pero no consiguió los resultados que esperaba. También sugirió que se pudieran realizar las peticiones en español, ya que considera que en inglés es más díficil al no ser su lengua materna.

7.2.5. Usuario 5

7.2.5.1. Contexto personal

El usuario 5 es un estudiante de tercer curso en el doble grado de Ingeniería Aeroespacial e Ingeniería de Telecomunicaciones en la Universidad Rey Juan Carlos, entre sus aficiones e intereses, están el deporte, la botánica y los videojuegos. Usa con cierta frecuencia LLMs para realizar tareas sencillas, además, ha jugado algunos videojuegos y entiende un poco sobre el concepto de diseño de niveles.

7.2.5.2. Desarrollo de un entorno de forma manual

Para variar un poco las tareas, al usuario 5 se le fue asignado realizar un parque, estuvo peleando con la interfaz de Unreal Engine al principio, pero al final consiguió realizar el diseño que se muestra en la figura 7.9.

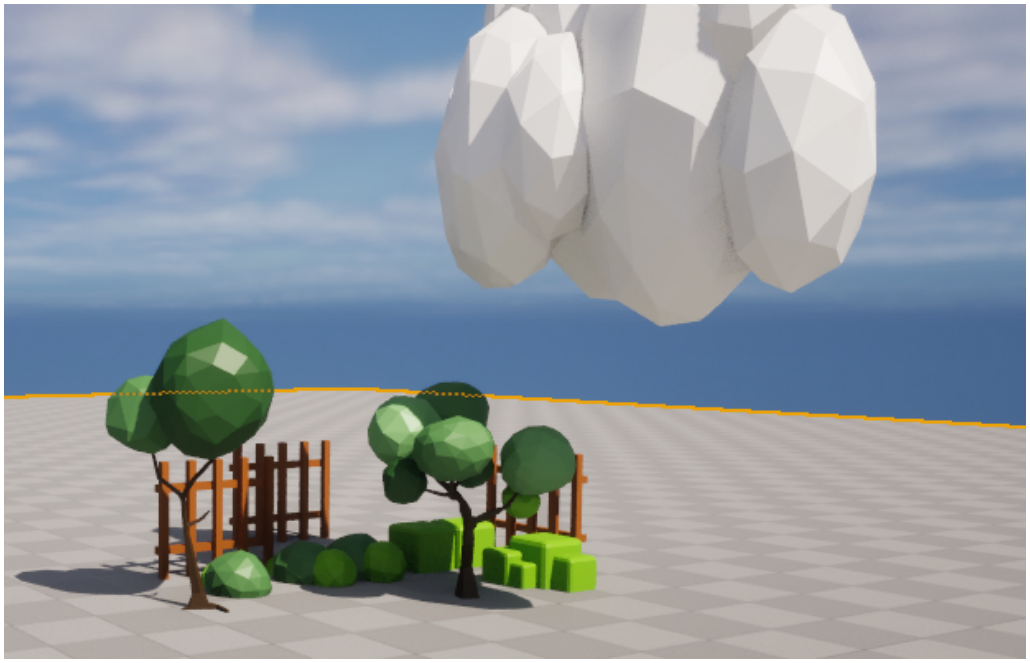

Figura 7.9: Entorno desarrollado manualmente por el usuario 5

7.2.5.3. Desarrollo de un entorno con LLM

Para el entorno desarrollado via LLM el usuario 5 no tuvo muchos problemas para usarla, aunque como se puede ver en la figura 7.10 hay diferentes bloques blancos, ya que la LLM al pedirsele un parque generó flores y animales que no tienen textura.

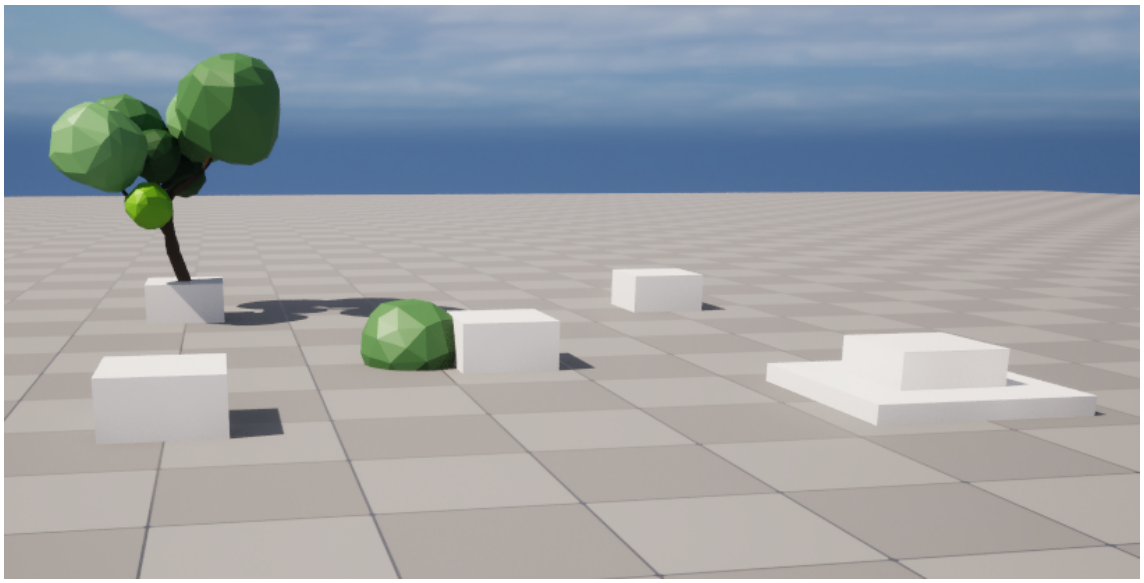

Figura 7.10: Entorno desarrollado usando la herramienta por el usuario 5

7.2.5.4. Obtención de métricas y comentarios

El usuario 5 consideró la aplicación como funcional, aunque le hubiese gustado que mostrara más texturas. En cuanto al tiempo tardado, podemos observar que tardó 2 minutos y 47 segundos en diseñar el entorno de forma manual y 2 minutos 50 segundos realizándolo con la aplicación, restando el tiempo de procesamiento de peticiones, tiempos bastante similares.

El usuario 5 probó con peticiones pequeñas y específicas sobre donde y cómo generar los objetos.

7.2.6. Usuario 6

7.2.6.1. Contexto personal

El usuario 6 es un estudiante de tercer curso de Ingeniería Mecánica en la Universidad Carlos III de Madrid, entre sus aficiones e intereses, están los deportes de contacto y los videojuegos. Nunca ha usado LLMs ya que no le llaman la atención, aunque sí sabe lo que son. Ha jugado videojuegos e incluso diseñado algún nivel propio en sus ratos libres.

7.2.6.2. Desarrollo de un entorno de forma manual

El usuario 6 fue también asignado diseñar una ciudad, en este caso el optó por diseñar una ciudad con tráfico entre las calles. No tuvo problemas para usar el entorno de Unreal.

Figura 7.11: Entorno desarrollado manualmente por el usuario 6

7.2.6.3. Desarrollo de un entorno con LLM

En cuanto a la generación por LLM, decidió crear una ciudad completamente nueva sin nada que ver con la anterior, consiguiendo un resultado aceptable en la que hay un coche “aparcado”.

Figura 7.12: Entorno desarrollado usando la herramienta por el usuario 6

7.2.6.4. Obtención de métricas y comentarios

El usuario 6 consideró la aplicación como muy funcional, aunque le hubiese gustado que la precisión en la que se colocan los objetos fuera mayor. En cuanto al tiempo tardado, podemos observar que tardó 3 minutos y 23 segundos en diseñar el entorno de forma manual y 2 minutos 2 segundos realizándolo con la aplicación, restando el tiempo de procesamiento de peticiones, adelantando nuestra herramienta al uso manual en este caso.

El usuario 6 probó con peticiones con bastante contenido, siendo muy específico asignando las coordenadas de cada objeto que quería generar.

7.2.7. Usuario 7

7.2.7.1. Contexto personal

El usuario 7 es un estudiante de tercer curso de Comunicación Audiovisual en la Universidad Complutense de Madrid, entre sus aficiones e intereses, están principalmente los videojuegos. Ha usado LLMs para realizar grandes tareas. Ha jugado muchos videojuegos y tiene nociones teóricas sobre diseño de niveles.

7.2.7.2. Desarrollo de un entorno de forma manual

El usuario 7 fue asignado diseñar una ciudad, en este caso el optó por diseñar una ciudad con una plaza con un árbol en el centro.

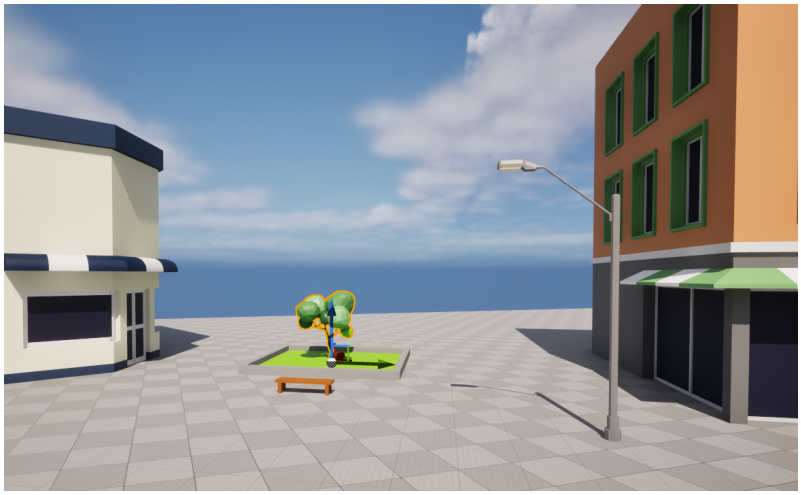

Figura 7.13: Entorno desarrollado manualmente por el usuario 7

7.2.7.3. Desarrollo de un entorno con LLM

Para el diseño de entornos con LLM, a pesar de que el usuario 7 fue bastante específico la LLM no fue capaz de responder correctamente a sus peticiones dejando una casa dentro de otra como se puede ver en la figura 7.14.

Figura 7.14: Entorno desarrollado usando la herramienta por el usuario 7

7.2.7.4. Obtención de métricas y comentarios

El usuario 7 consideró la aplicación como funcional con espacio a mejoras, y achacó la falta de precisión a nuestro uso de una LLM poco potente. En cuanto al tiempo de uso, en el diseño manual tardó 4 minutos y 33 segundos y en el automático, 1 minuto y 57 segundos, siendo bastante rápido a pesar de su imprecisión.

El usuario 7 probó con peticiones grandes con bastante contenido.

7.2.8. Usuario 8

7.2.8.1. Contexto personal

El usuario 8 es un estudiante de cuarto curso de Filosofía en la Universidad Complutense de Madrid, entre sus aficiones e intereses, están los deportes de contacto, los videojuegos y la música. Ha usado LLMs para realizar grandes tareas como redacciones. Ha jugado muchos videojuegos pero no tiene ninguna idea sobre diseño de niveles.

7.2.8.2. Desarrollo de un entorno de forma manual

El usuario 8 al diseñar una ciudad, optó por diseñar una ciudad con una carretera con tráfico atravesándola con varios edificios a los lados.

Figura 7.15: Entorno desarrollado manualmente por el usuario 8

7.2.8.3. Desarrollo de un entorno con LLM

Al diseñar entornos con LLM, a pesar de usar una gran descripción sin muchos detalles la LLM fue capaz de realizar un entorno de calidad aceptable.

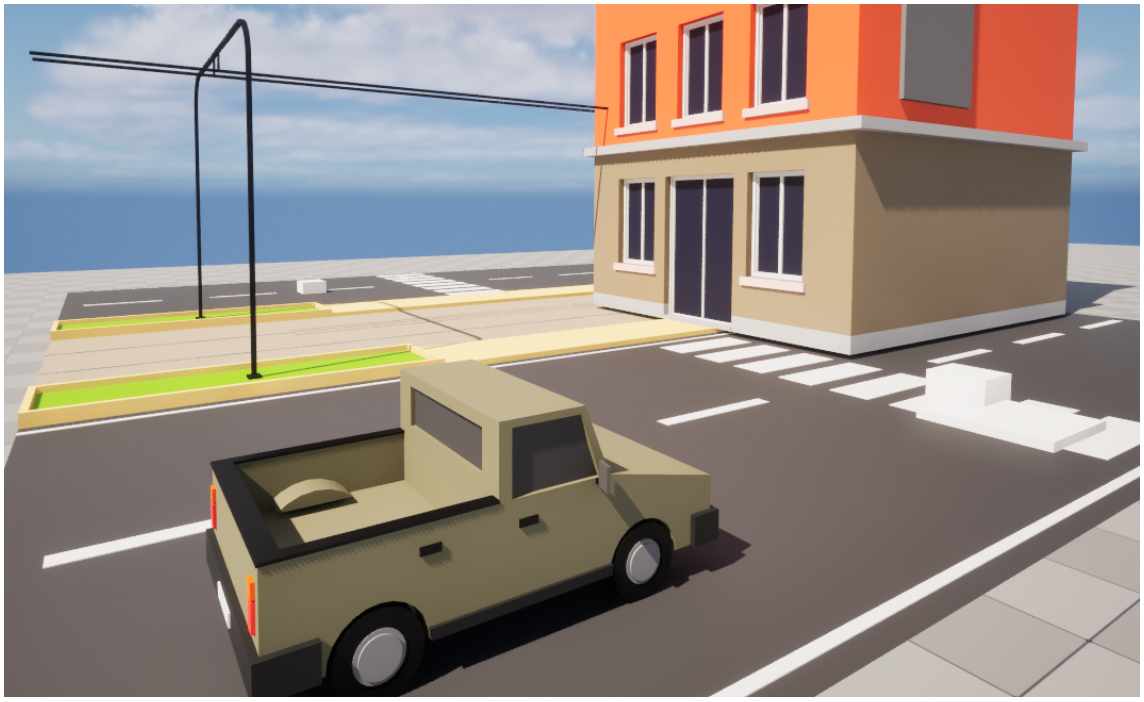

Figura 7.16: Entorno desarrollado usando la herramienta por el usuario 8

7.2.8.4. Obtención de métricas y comentarios

El usuario 8 consideró la aplicación como funcional. En cuanto al tiempo de uso, en el diseño manual tardó 3 minutos y 25 segundos y en el automático, 4 minuto y 57 segundos.

El usuario 8 probó con peticiones grandes bastante generales sin incidir mucho en los detalles.

7.2.9. Usuario 9

7.2.9.1. Contexto personal

El usuario 9 es un estudiante de tercer curso del Doble Grado en Derecho y Ciencias Políticas en la Universidad Carlos III de Madrid, entre sus aficiones e intereses, están la cocina, el arte y la música. No es un usuario frecuente de LLMs, tampoco ha jugado muchos videojuegos ni tiene ninguna idea formal sobre diseño de niveles.

7.2.9.2. Desarrollo de un entorno de forma manual

El usuario 9 necesito cierto tiempo para hacerse con los controles de Unreal y diseñó un bloque de edificios con cuatro vehículos aparcados.

Figura 7.17: Entorno desarrollado manualmente por el usuario 9

7.2.9.3. Desarrollo de un entorno con LLM

En cuanto a la generación por LLM, se generó dos edificios contiguos con un coche en la pared.

Figura 7.18: Entorno desarrollado usando la herramienta por el usuario 9

7.2.9.4. Obtención de métricas y comentarios

El usuario 9 consideró la aplicación como aceptable pero con espacio para mejorar. En cuanto al tiempo de uso, en el diseño manual tardó 4 minutos y 11 segundos y en el automático, 3 minuto y 57 segundos siendo algo más rápido la manera automática.

El usuario 9 probó con peticiones pequeñas y específicas incidiendo en los detalles.

7.2.10. Usuario 10

7.2.10.1. Contexto personal

El usuario 10 es un estudiante de doctorado de Ingeniería Biomédica en la Universidad Carlos III de Madrid, entre sus aficiones e intereses, están la cocina, la natación y la ingeniería. Nunca usa LLMs, y aunque ha jugado bastantes videojuegos no tiene una gran noción sobre diseño de niveles

7.2.10.2. Desarrollo de un entorno de forma manual

El usuario 10 se acostumbró rápidamente a Unreal y diseñó un pequeño parque con bancos y arboles en macetas.

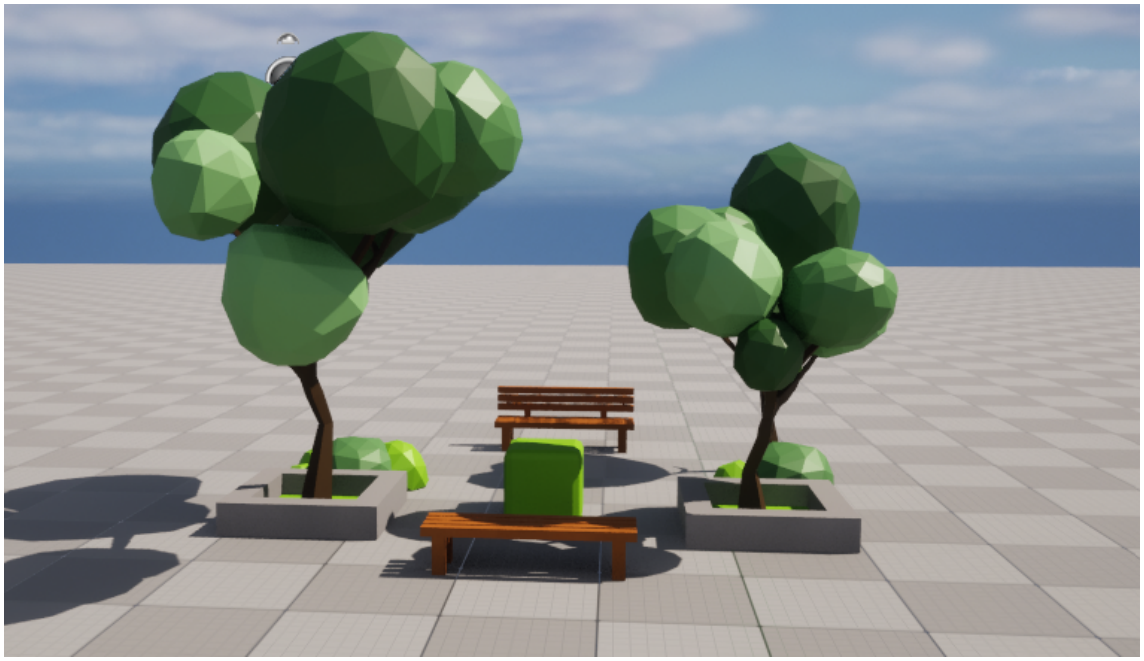

Figura 7.19: Entorno desarrollado manualmente por el usuario 10

7.2.10.3. Desarrollo de un entorno con LLM

Al realizarlo con la LLM, lo encontró una herramienta cómoda pero no muy precisa, se frustró especialmente con que aparecieran cubos sin texturas.

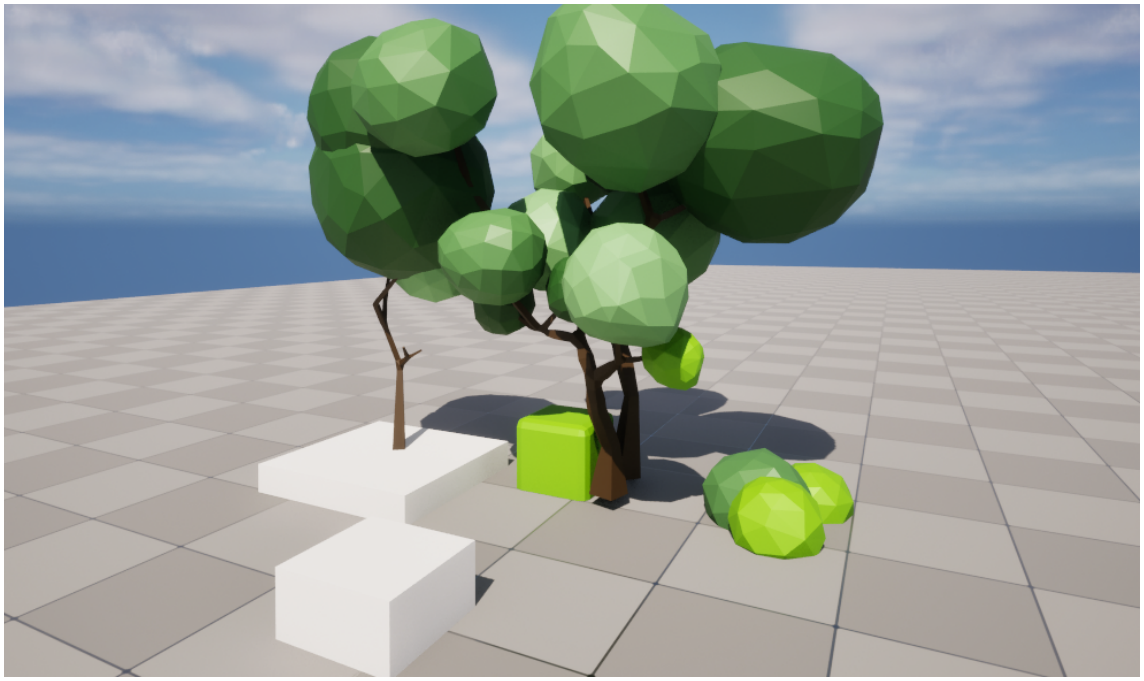

Figura 7.20: Entorno desarrollado usando la herramienta por el usuario 10

7.2.10.4. Obtención de métricas y comentarios

El usuario 10 consideró la aplicación como aceptable pero con espacio para bastantes mejoras. En cuanto al tiempo de uso, en el diseño manual tardó 3 minutos y 56 segundos y en el automático, 2 minuto y 43 segundos siendo algo más rápido la manera automática.

El usuario 10 probó con peticiones pequeñas y específicas incidiendo en los detalles y también con peticiones grandes para probar ambos métodos.

7.2.11. Sugerencias hechas por los encuestados en el cuestionario

En el cuestionario realizado posterior al experimento, la última pregunta trata de sugerencias que los usuarios consideran útiles para el desarrollo y funcionalidades de la aplicación. La pregunta dice así:

¿Se te ocurre alguna mejora para la herramienta? Esta pregunta es de carácter cualitativo, y podemos ver que las respuestas que más se repiten son:

- Que se pueda elegir el modelo y no sea aleatorio.

- Que se pueda indicar a la LLM el eje Z para colocar objetos en altura, como nubes.
- Que se puedan redactar las peticiones en español.
- Que se pueda indicar a la LLM la rotación y escala de los objetos pedidos.
- Que la LLM tenga contexto de lo generado previamente.

Estas sugerencias serán incluidas en las líneas de trabajo a futuro.

7.3. Evaluación cuantitativa del proceso de diseño de usuarios

En esta sección se expondrán los datos obtenidos de las métricas cuantitativas del cuestionario, analizándolos para poder deducir la línea de opinión general de los encuestados, además de la variedad de respuestas, entre otras métricas, este estudio es fundamental y complementa a las métricas cualitativas expuestas previamente.

7.3.1. Datos del cuestionario

Tras realizar el experimento sobre los encuestados y pedirles que respondiesen al cuestionario de satisfacción con preguntas valoradas entre 1 y 5, obtenemos los siguientes datos:

- **¿Crees que has ahorrado tiempo usando la herramienta en vez de haciéndolo a mano?** Siendo 1, "No he ahorrado nada de tiempo" y 5, "He ahorrado una cantidad de tiempo considerable".

  Para esta pregunta, la **media** de los valores es 3,5, luego el resultado general es que sí se ha conseguido ahorrar tiempo usando la herramienta. En cuanto a la **desviación típica**, su valor es de 0.972, por lo que los valores no se alejan mucho de la media, pudiendo asumir entonces que la opinión general está bastante de acuerdo con que se ahorra tiempo usando el sistema. Como podemos observar, un 70 % de los encuestados considera que se ahorra una cantidad considerable de tiempo al usar la herramienta, mientras que un solo usuario considera que no se ahorra nada de tiempo. En la figura 7.21 se pueden ver estos datos gráficamente.

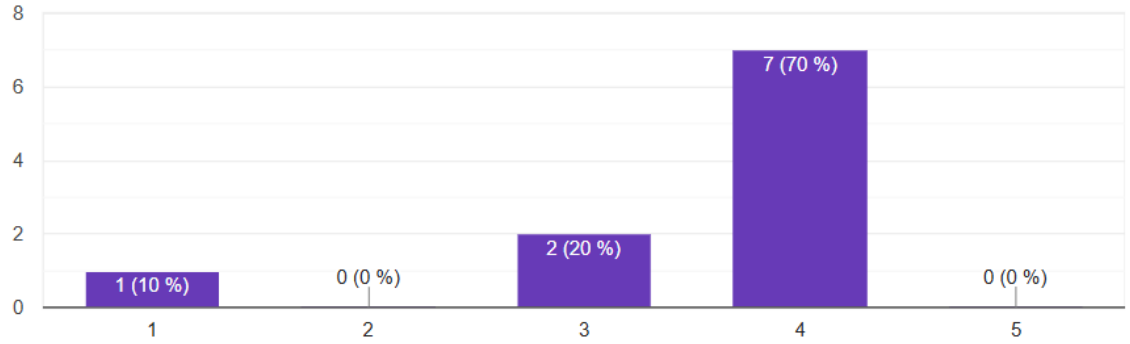

Figura 7.21: Respuestas sobre el tiempo ahorrado

- **¿Crees que la herramienta te ha ahorrado trabajo o esfuerzo mental?** Siendo 1, "Nada" y 5, "Mucho".

  En esta pregunta encontramos que la **media** es de 3,8, luego la opinión general opina que la herramienta si ha ayudado para ahorrar trabajo o esfuerzo mental, en cuanto a la **desviación típica**, que tiene un valor de 1.033, podemos observar que las opiniones son algo más variadas que en la pregunta anterior, pero el consenso general está bastante aproximado a la media. Las respuestas a esta pregunta son muy variadas, aunque podemos deducir de los datos obtenidos, respuestas prácticamente positivas, ya que el 90 % de ellas están dentro de los que piensan que la herramienta sí ahorra esfuerzo y trabajo mental. En la figura 7.22 se pueden ver estos datos gráficamente.

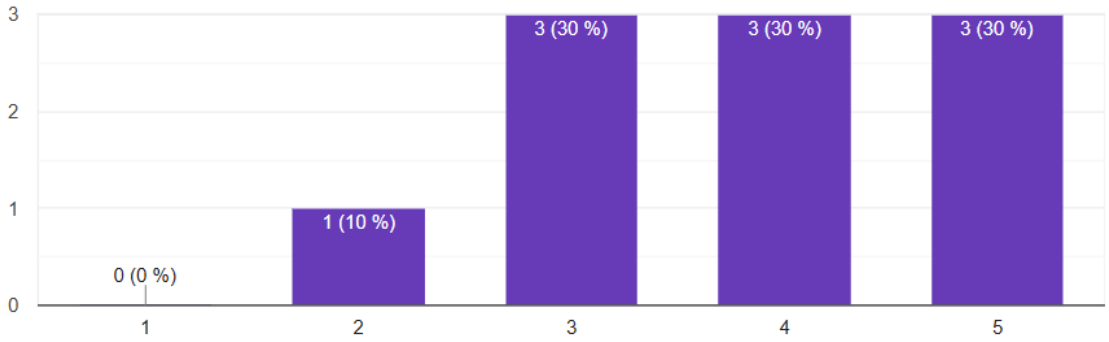

Figura 7.22: Respuestas sobre el esfuerzo mental ahorrado

- **¿Consideras la herramienta sencilla de usar?** Siendo 1, "Es una herramienta difícil de usar" y 5, "Es una herramienta fácil de usar".

  De esta pregunta obtenemos que la **media** es de 4,4, luego la opinión general considera que la herramienta es sencilla de utilizar. En cuanto a la **desviación típica**, tiene un valor de 0,699 considerablemente menor que en las preguntas anteriores, luego podemos deducir que los encuestados están todos bastante de acuerdo con la facilidad de uso de la herramienta. En la figura 7.23 se pueden apreciar estos datos gráficamente.

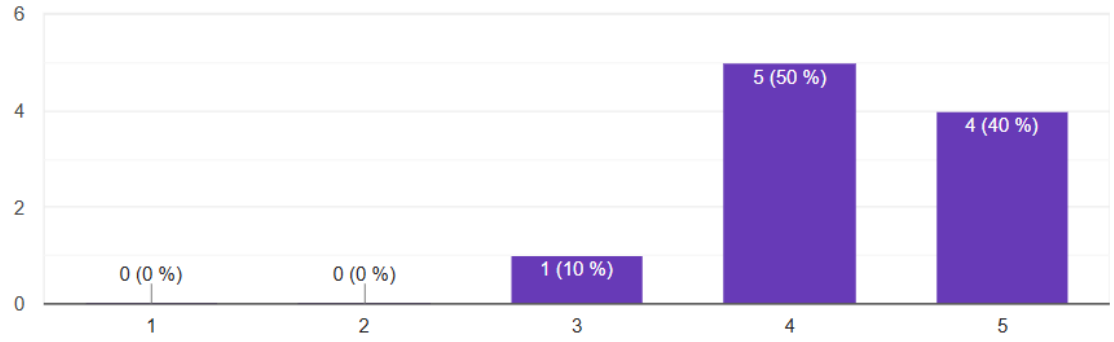

Figura 7.23: Respuestas sobre la facilidad de uso de la herramienta

- **¿Te ha ayudado a concentrarte más en el concepto y menos en el detalle?** Siendo 1, "No me ha ayudado nada" y 5, "Me ha ayudado mucho".

  Para esta pregunta obtenemos una **media** de 3,4 y una **desviación típica** de 1,265, podemos observar que aunque la opinión es positiva, las diferentes opiniones de los encuestados son muy variadas entre ellas. Pero aunque no haya un consenso general podemos observar que el 60 % opina que no les ha ayudado a centrarse más en el concepto que en el detalle. En la figura 7.24 se pueden ver estos datos gráficamente.

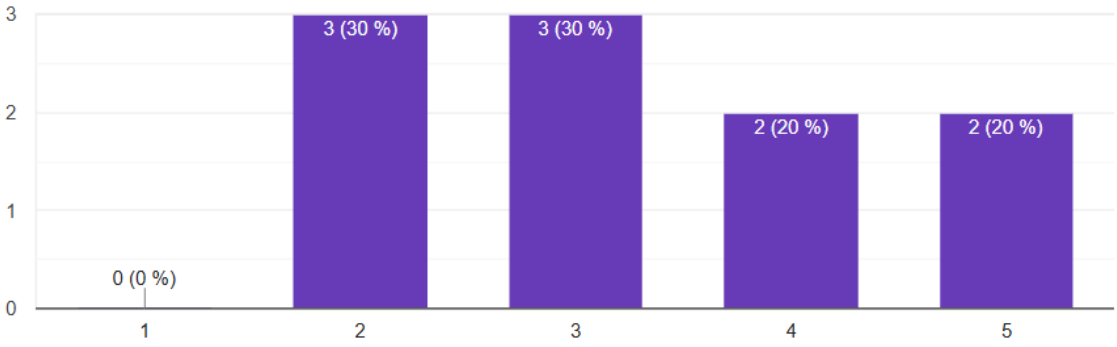

Figura 7.24: Respuestas sobre si la herramienta ha sido útil para centrarse más en el concepto

- **¿Cómo de precisa consideras que ha sido la herramienta respecto a tu descripción?** Siendo 1, "Nada precisa" y 5, "Muy precisa".

  En esta pregunta observamos una **media** de 2,9 significando que la precisión de la herramienta es media, en cuanto a la **desviación típica** su valor de 0,994 nos indica que el consenso general concuerda en cuanto al nivel de precisión de la herramienta. Podemos observar que un 40 % opina que la herramienta tiene una buena precisión. Como podemos ver en la figura 7.28, las respuestas son variadas pero podríamos considerar, que en general la herramienta tiene una precisión media.

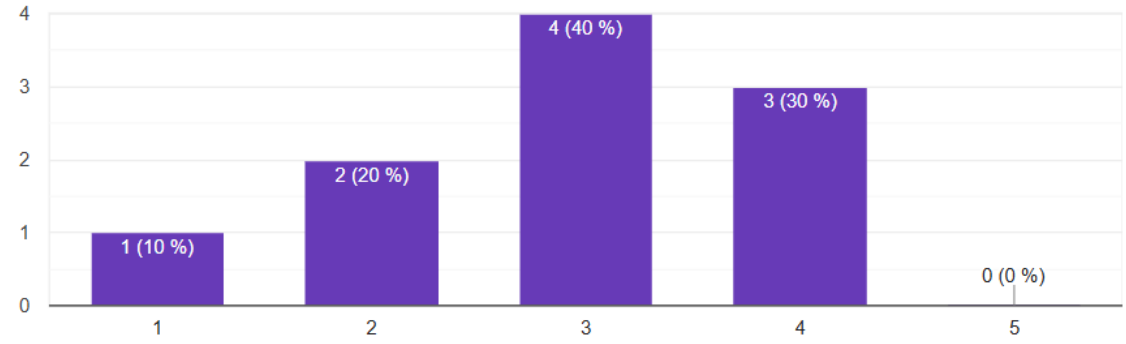

Figura 7.25: Respuestas sobre si la herramienta ha sido precisa respecto a la descripción

- **¿Usarías la herramienta en un desarrollo real?** Siendo 1, "No, para nada" y 5, "Si, siempre que tenga la oportunidad".

  Para esta pregunta obtenemos una **media** de 2,8 luego podemos deducir que los encuestados en general no usarían la herramienta además obtenemos una **desviación típica** de 0,919 luego el consenso general está bastante de acuerdo con la media. Vemos que un 20 % opina que si la usaría con un extremo de un encuestado diciendo que no la usaría nunca. En la figura 7.26 se pueden ver estos datos gráficamente.

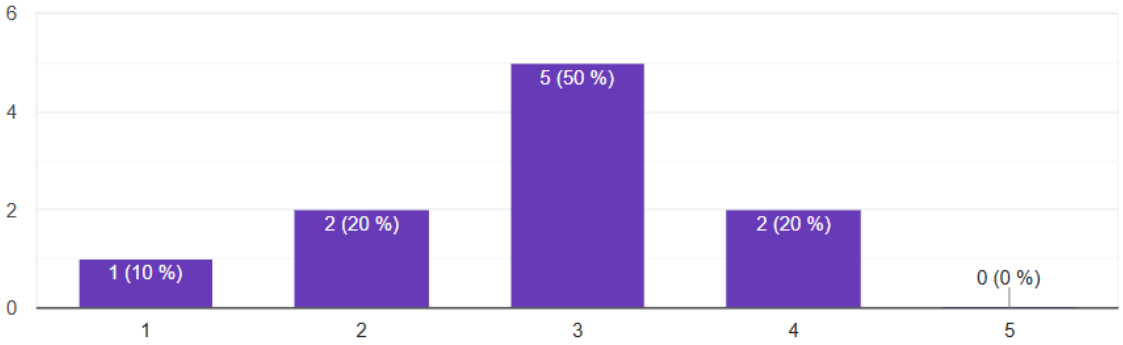

Figura 7.26: Respuestas sobre si el usuario usaría la herramienta en un desarrollo real

- **En general, valora la herramienta** Siendo 1, "Nada útil" y 5, "Muy útil".

  Para esta pregunta obtenemos una **media** de 3,1 luego la valoración de la herramienta en general es positiva, obtenemos además, una **desviación típica** de 0,994 luego el consenso general está bastante de acuerdo con la media. Vemos que un 80 % opina neutral sobre el sistema. En la figura 7.27 se pueden ver estos datos gráficamente.

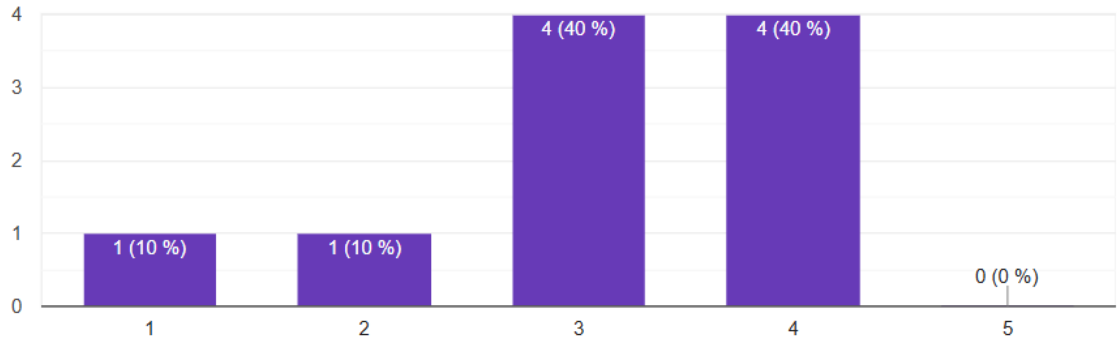

Figura 7.27: Respuestas sobre la valoración general de la herramienta

7.3.2. Otras métricas

En esta sección se relatan otras medias obtenidas durante el proceso, cómo pueden ser el tiempo tardado entre ambos métodos y su relación.

7.3.2.1. Relación de tiempo entre generación manual y automática

En la siguiente figura podemos observar el tiempo tardado por usuario usando la generación manual y usando la generación procedimental con LLM.

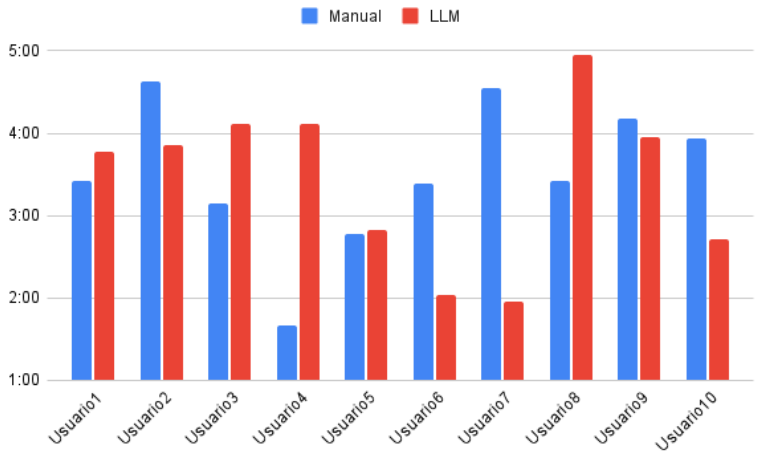

Figura 7.28: Tiempo de uso por usuario entre generación manual y por LLM

Podemos destacar el caso del usuario 4, el cuál fue muy veloz en su uso del diseño manual, y los casos de los usuarios 6 y 7 en los cuáles la generación por LLM fue considerablemente más rápida que el método manual restando el tiempo de procesamiento.

En la siguiente figura, podemos observar el tiempo medio de cada uno de los métodos, siendo el método automático algo más rápido tardando un promedio de 3 minutos 43 segundos y el manual 3 minutos 51 segundos teniendo en cuenta que al tiempo por LLM se le ha substraído el tiempo de procesamiento.

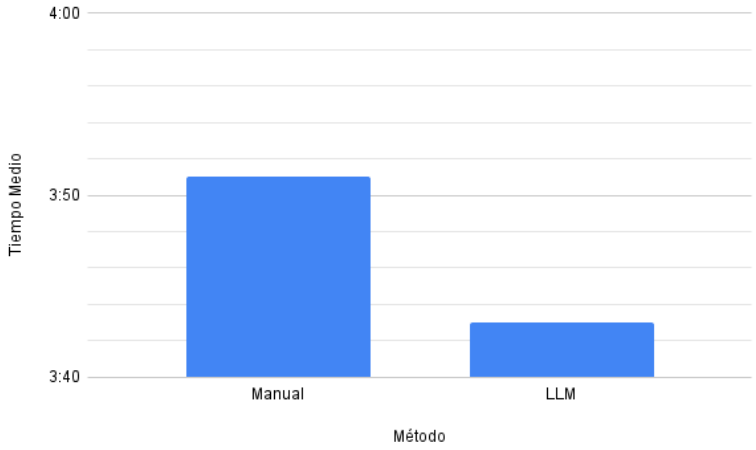

Figura 7.29: Tiempo medio de uso entre la generación manual y por LLM

7.3.3. Conclusiones finales del experimento

Tras observar las métricas cualitativas y cuantitativas anotadas, podemos observar que el método de diseño de niveles mediante uso de niveles aún no cumple con los requisitos necesarios para rivalizar con la precisión y el tiempo invertido usando el método manual.

Capítulo 8

Discusión

El presente trabajo de investigación ha abordado el desarrollo de generación procedimental de entornos, para el que se han usado diferentes LLMs en un entorno implementado en *Unreal Engine 5*. Este capítulo de discusión tiene como objetivo analizar los aspectos más relevantes del proyecto, evaluando las hipótesis planteadas y destacando tanto las ventajas como las limitaciones de este.

La hipótesis principal del proyecto afirmaba que, explotando las capacidades generativas de las LLMs es posible generar niveles jugables, estéticos y creativos de una manera sencilla y barata. Los resultados obtenidos no confirman esta hipótesis, deja sin demostrar que los modelos de lenguaje de gran escala pueden ser utilizados eficazmente para asistir en el diseño de niveles de videojuegos. Las pruebas realizadas muestran que los niveles generados no cumplen con criterios básicos de jugabilidad en general, sino que tampoco presentan variedad, coherencia visual o elementos de originalidad, en un punto positivo, todo ello sin requerir conocimientos técnicos avanzados por parte del usuario. Aún así confíamos que en un futuro, esto podría abrir la puerta a nuevas posibilidades en el desarrollo de videojuegos, especialmente para equipos pequeños o desarrolladores independientes que buscan acelerar el proceso creativo y reducir costos.

La primera ventaja que en la cual se puede pensar, es en el ahorro de tiempo que conlleva el uso de la aplicación para conseguir las bases de un nivel o para conseguir inspiración de cómo debería ser la composición o el tema del nivel que le gustaría hacer. Ya que al usar la aplicación solo se tiene que esperar el tiempo que se tarde en generar el nivel, mientras que al hacerlo manualmente, el tiempo invertido en buscar las texturas o crearlas desde cero, ver la composición del nivel desde el principio y otros aspectos recaen sobre el usuario. Por tanto, si el usuario busca eficiencia pensará en usar nuestra aplicación para la generación del nivel o, en su defecto, la generación de un boceto que posteriormente se pueda modificar. Pero, aunque útil para realizar un primer boceto preliminar la herramienta aún deja que desear en cuanto a tiempo de procesamiento ya que dependiendo del sistema podría tardar más que una colocación de los elementos.

Pensando en la primera ventaja, puede surgir el primer inconveniente o limitación. Como el sistema no apoya que el usuario modifique el nivel dado por la aplicación y este pueda ser devuelto al sistema para mejorarlo, pueden surgir dos escenarios:

que el usuario intente conseguir una buena base desde el comienzo para que pueda seguir la creación del nivel manualmente sin muchas dificultades consumiendo mucho tiempo e intentos o, que el usuario se conforme con una base poco fiel a lo que pidió y la ayuda que brindaba la aplicación sea más bien un inconveniente para el usuario ya que tendrá que modificar todo lo que ha generado el sistema además de tener que añadir más contenido a dicho nivel. En cualquiera de esos dos escenarios, el uso de la aplicación puede resultar incómodo para el usuario y podría elegir no usarla.

La segunda ventaja que tiene nuestro sistema es su modularidad y capacidad de expandirse indefinidamente, ya que es posible añadir nuevos modelos para nuevas temáticas que el usuario necesite, haciendo así de nuestro sistema una aplicación polivalente para cualquier tipo de temática o uso que se le pueda dar.

Relacionado con esta ventaja, para añadir nuevos modelos, habría que modificar una sección del código haciendo este proceso difícil y poco intuitivo para el usuario final.

Otra ventaja que tiene el sistema, es su adaptabilidad a diferentes equipos y sus recursos, ya que al existir todo rango de LLMs con diferente coste de computo, se podrá usar una LLM diferente personalizada a las capacidades del ordenador del usuario, como una de bajos recursos para equipos de gama baja o grandes LLMs para equipos más potentes.

Otra limitación del proyecto es que si quieres conseguir mejores resultados o resultados más precisos, el usuario necesitaría disponer de un modelo más grande. Esto es una gran limitación ya que los modelos más grandes tienen un consumo muy elevado de cómputo que la mayoría de usuarios no puede permitirse, debido al equipo necesario para ejecutar estos modelos.

Una pequeña limitación del sistema es, que hasta la fecha solo interpreta correctamente peticiones inglés, y aunque esto no sea una limitación del sistema en sí, puede ser una barrera de entrada que dificulte su uso a usuarios que no dominen el idioma.

Capítulo 9

Conclusiones y Trabajo Futuro

Tras las métricas y las pruebas con usuarios podemos asegurar que nuestra herramienta tiene mucho futuro, aún sin lograr el resultado esperado, puesto que en un primer momento nuestro objetivo era la generación de un espacio completo detallado, tras probar las primeras versiones se verificó que no era posible. Así que lo redujimos a la generación de un espacio más pequeño. Puesto que no se podía llegar a un nivel de calidad mayor a actual con nuestros medios. Nos centramos en que fuese una máquina fácil de usar, visual y útil para el usuario, y lo hemos conseguido. Aunque a veces el resultado puede diferir en gran medida de lo que el usuario esperaba de su petición a la máquina, hemos logrado crear una herramienta de ayuda a los diseñadores, que era nuestro objetivo principal.

Hemos conseguido crear una herramienta sencilla, intuitiva y rápida de usar, muy escalable y visualmente bonita, ejecutándose desde el propio Unreal donde observas los mismos cambios, creemos nosotros y más importante aún, lo piensan nuestros usuarios de prueba, y por ello creemos que la gran mayoría de nuestros problemas se reducen al uso de LLMs pequeños, comparados con las que todos conocemos o hemos usado, hasta para hacer tareas mucho más sencillas que esta. Si se usase un LLM de mucha potencia como pueden ser GPT-3.5 o 4 podríamos obtener resultados altamente específicos, y en mucho menos tiempo, por lo que realizar pruebas, cambios o actualizaciones sería muy eficiente. Aún con todo esto creemos conveniente afirmar que nuestro proyecto no cumple con la hipótesis previamente mencionada, ya que aunque conseguimos realizar los niveles de manera sencilla y barata, debido a sus características no creemos que sean ni jugables ni estéticos ni creativos.

Además tiene potencial y espacio de mejora, hay que tener en cuenta que hemos tenido que aprender sobre la marcha durante todo el proyecto, lo que ha ido dando resultados mejores de forma exponencial en el tiempo, con un grupo mayor y más profesionalizado en la materia, una mayor cantidad de tiempo de desarrollo o ambas, estamos seguros que puede llegar a ser una herramienta realmente útil en un entorno profesional, y más si se empezase a indagar en el tema, por ejemplo entrenando LLMs para ello o re-entrenando esta a medida que avanzas en la creación de tu entorno generando cohesión con lo anterior, o generando plantillas de creación que aplicase el LLM, o simplemente generando una idea rápida o una cascada de ellas hasta que el creador le convenza una o le de una idea parecida y pueda modificarla o retocarla.

Para el resto de limitaciones que sí que nos competen directamente podremos ver ideas y mejoras en el trabajo a futuro que de realizarse mejoraría aún más la experiencia del usuario y así mismo los resultados obtenidos.

9.1. Líneas de trabajo futuro

En el proyecto se tenía como objetivos crear una aplicación que generase niveles, jugables, para ayudar a diseñadores en su trabajo de creación de niveles donde se buscaba que nuestra aplicación hiciese un boceto inicial del nivel y que mediante los retoques del usuario y varias consultas más al LLM se crease un nivel que el usuario esperaría, se ha conseguido crear una aplicación que genere bocetos iniciales de niveles pero no se ha conseguido que el usuario pueda modificar el nivel y una vez modificado se vuelva a mandar para que se genere algo más preciso. Una vez expuesto esto, se van a presentar varias líneas de trabajo que, en caso de implementarse, el rendimiento de la aplicación mejoraría con respecto a los objetivos iniciales de este proyecto pudiendo incluso superarlos.
Ahora se explicarán las distintas líneas que se han pensado para el futuro del proyecto.

9.1.1. Renderizado en modo editor

Una consideración que se tuvo en algún punto del desarrollo de esta herramienta es que funcionase en el modo editor de UE5 porque es natural en una herramienta que ayude a diseñar, en este caso niveles, funcione en modo editor. Para que sea posible modificar y retocar los diferentes modelos que han sido generados por LLM.

Para que la aplicación funcionase en el modo editor de UE5 y no en el modo juego, hubiera requerido de la creación de nuestro propio *parser* de texto, ya que encontrar un parser que convirtiese un *String* a un archivo JSON resulta complicado, dado que los que hay en el mercado que se ajusten al presupuesto de este proyecto son todos en modo juego. Por lo que no se pudo implementar enteramente esta funcionalidad. Sin embargo vamos a abordar qué cambios son necesarios, teóricamente para que esta funcionalidad se pueda realizar.

Lo primero que habría que cambiar el modo en el que se inicia el servidor en UE5 y cuando se establece la conexión con el cliente, en este caso debería ser al abrir el nivel en sí. Para ello sería necesario modificar la clase *SocketerSpawner*, para que funcionase en el método nativo de UE5 *OnConstruction*, de esta manera el servidor se ejecutaría directamente al abrir el nivel. Además es necesario asegurarse que no se abren múltiples servidores en caso de ejecutar varias veces el nivel o si se abre el *blueprint* de *JSONSpawner*, ya que esto podría dar lugar a errores, para ello se asegura dentro del código que solo sea ejecutable una única vez, y por un nivel.

Luego habría que cambiar el *blueprint* de *JSONSpawner* para que se ejecutase en modo editor y no en modo juego. Para ello, hay que cambiar cuando el *blueprint* se activa, como está hecho actualmente, es con el nodo *Event tick* el cual funciona solo cuando está el modo juego activo. Para cambiar esto, hay que crear un nodo que podemos llamar *Editor Tick*. Para crear este *Editor tick* se tendría que implementar

una *Blueprint Interface* que contiene una función que se pueda llamar en tiempo de edición, en modo Editor. Llamada *Editor Tick*.

Luego se crearía un *blueprint* que hereda de *EditorUtilityActor* que tiene las características de "allow tick before play", "tick even paused" y "is Editor only Actor". Para habilitar el funcionamiento en modo editor.

En la función "Construction Script" de este *blueprint* que hereda de *EditorUtilityActor* se asegura que el actor padre implementa la *Blueprint Interface* anteriormente explicada y también asigna un timer para la función "Editor Tick" de la *Blueprint Interface*. Aún así, para usar retardos dentro de las funciones, es necesario crear un temporizador propio, ya que al funcionar en tiempo de edición, los temporizadores nativos de Unreal no se ejecutan, siendo necesario llevar la cuenta del tiempo pasado calculándolo por cada fotograma que pasa en tiempo de edición, con un método propio.

Una vez creado esto, se tendría que modificar el *blueprint SpawnerJSON* en su "Construction Script" añade un nodo *Add child actor component* donde la clase hija es el *blueprint* que hereda de *EditorUtilityActor* y después implementa dentro de este *blueprint* la interfaz creada anteriormente.

9.1.2. Base de texturas o *sprites* escalables

Otro cambio que, debido a la falta de recursos y conocimientos en el área de modelado en 3D, no se pudo llevar a cabo, es el de tener una base de texturas más grande.

En el caso de que se tenga una base de texturas más grande, el modelo que se ha propuesto para el renderizado de texturas no soportaría esa escalabilidad, por lo que sería necesario cambiarlo. Vamos a explorar cómo se podría cambiar para mejorar la escalabilidad del renderizado.

Una posibilidad sería tener una base de datos donde se guarde cada textura con qué palabras puede estar relacionada, de esta forma una textura puede referirse a distintas palabras sinónimas y que el sistema no sea tan restrictivo con el uso de las palabras, ya que se pueden añadir sinónimos más fácilmente. Una vez creada la base de datos con los datos descritos anteriormente. En la aplicación, cuando se llegase al punto de tener que renderizar una palabra, lo haría de la siguiente forma dentro de un *Blueprint* o de una clase de C++:

- Primero se haría una query buscando las texturas que tuviesen esa palabra asociada.
- El resultado puede ser de uno o varios, en caso de no encontrar la palabra dentro de la base de datos se elegiría una textura por defecto, como se hace ahora.
- Si es uno, se lleva esa textura al nodo *Spawn actor from class* y el sistema se encargará de renderizar esa textura en la posición que corresponda, ese paso no cambia.

- En caso de ser varios la query daría como resultado, una tabla o un set con las texturas correspondientes, de las cuales se enumerarían del 0 al tamaño de la tabla - 1, siendo el 0 la primera textura y tamaño de la tabla - 1 corresponde a la última textura, se escogería uno al azar de entre eso, luego haría el mismo procedimiento que cuando da un solo resultado.

De esta manera, si queremos aumentar la cantidad de texturas de la aplicación, solo es necesario añadirla a la base de datos y cargar la textura correspondiente dentro del sistema, haciéndolo más escalable.

9.1.3. Entrenamiento de un LLM para que se especialice en el área de generación de niveles

Tener un LLM entrenado o, más bien, especializado en generar JSON a partir de prompts de lenguaje natural. Donde los datos en los que se basa para dar una respuesta contienen datos sobre una variedad de biomas distintos donde cada uno tiene alturas distintas, objetos distintos, entre otras cosas que conforman un bioma. Con un LLM entrenado para ese propósito tenemos la certeza de que se podrían generar niveles más diversos, extensos y detallados. Podemos suponer qué, para entrenar este modelo propuesto haría falta una cantidad de datos bastante extensa, donde esa cantidad puede resultar bastante difícil de obtener.

Si entrenáramos a este supuesto modelo para que dé las respuestas en un formato dado y con la variedad de conocimientos descritos antes, se cree que las respuestas obtenidas con este LLM sean más precisas a lo obtenido con el modelo actual. Sobretodo se estima que su precisión en la localización espacial de los objetos que conformen sus respuestas mejoren bastante en relación al actual.

9.1.4. Cargar niveles pre-hechos y posibilidad de guardar niveles generados por el LLM que pueden estar o no modificados por el usuario.

Otro cambio o mejora que se puede implementar sería, una vez el usuario ha obtenido un mapa que se ajuste a sus necesidades, o que satisfaga lo que había pedido a la aplicación. Este usuario pueda tener la opción de guardarlo dentro de la aplicación si así lo desea y, después, si en algún momento dado decide que el mapa anteriormente guardado lo necesita en ese momento donde también se encuentren mapas ya cargados desde el principio a modo de "ejemplo".

Para que se pueda implementar esto hay que cambiar la interfaz de la aplicación en el *widget* que se encarga de mandar los mensajes dados por el usuario de la aplicación al LLM. Tenga otros dos botones: un botón de "guardar" y otro de "cargar" que se encontrarían al lado del botón de "enviar".

El botón de "cargar" se encargará de que al presionar sobré él, se abra una lista con los nombres puestos a unos mapas pre-hechos correctos, o lo que es lo mismo, archivos JSON bien formados, donde cuando el usuario haga click sobre uno de los nombres, el sistema mandará un mensaje al módulo de Python, o parte del LLM,

para que mande ese JSON directamente sin que el módulo pida una respuesta al LLM. Luego la aplicación renderiza ese JSON como cualquier otro. También se mostrarán los mapas guardados por el usuario, si es ese el caso, junto a la lista de mapas pre-hechos.

El botón de "guardar" se encargará de guardar lo que hay actualmente en el mundo, con los cambios añadidos por el usuario a ese mapa, este proceso es explicado más adelante. Una vez hecho eso, lo enviará al módulo de Python, o parte del LLM, donde se guardará con el resto de mapas ya pre-hechos. Estos mapas como están bien construidos pueden ser usados como material para hacer *fine–tunning* con el LLM que este usando el usuario en ese momento, aparte de lo que ya se le manda por defecto.

Mejorando así la calidad de las respuestas obtenidas debido a que tiene más material para hacer el *fine-tunning*.

El problema viene al preguntarse cómo se debería guardar el mapa que posiblemente esté modificado por el usuario, porque en caso de no estar modificado, este proceso sería más sencillo ya que solo haría falta que cada mensaje que manda el LLM con el JSON para renderizarse se tenga que guardar temporalmente hasta que el usuario mande a través del widget otro mensaje que entonces se descartaría esa generación de nivel y en caso de que lo hubiese querido guardar el mensaje simplemente se guardaría con el resto de niveles pre-hechos. Pero con este pensamiento viene un problema y es que el usuario puede modificar lo generado por el LLM.

En este caso el proceso de guardar el mapa creado se vuelve bastante más complicado porque no sabemos qué texturas ha puesto el usuario y con qué propósito porque hay dos posibilidades, que haya decidido poner una textura a modo de entorno, en la aplicación *environment*; o que haya puesto una textura a modo de objeto, en la aplicación *object*; eso provocaría problema a la hora de parsear el mapa a la inversa porque no sabríamos cómo estructurar el JSON para que no tuviese errores.

Además de que no se sabe a qué *environment* pertenece cada objeto porque desde UE5 no es posible saber qué área ocupa cada *environment*, tal y como está implementado en estos momentos. Por lo que no se puede saber qué *objects* están contenidos en cada *environment*. En caso de poder saber estos datos lo que tendría que hacerse es parsear cada textura para obtener sus datos:

- El tipo de objeto, si es *object* o *environment*
- La descripción en este proceso puede ir vacía
- Para guardar los datos en relación al nombre se tendría que saber de qué palabra viene o lo que es lo mismo hacer el proceso inverso al de renderizar las texturas, viendo que textura tiene y asignarle la palabra asociada a esa textura correspondiente
- Para la posición si es un *object* no hay problema porque de alguna manera obtendrías las coordenadas "x" e "y" de donde se encuentra la textura puesta en el mapa y se lo asignas al campo correspondiente.

  El problema viene como se ha comentado anteriormente antes, de las texturas con el campo *type* con valor *environment* porque no se puede saber cuánto

ocupa el área de este para saber qué puntos son los extremos y poder poner esas coordenadas dentro del archivo JSON. En caso de que saberlo, se obtendrían esos datos y se colocarían en el archivo JSON para mandarlo como mensaje a la parte del LLM donde se guardaría este mapa

Luego se tiene que estructurar el JSON de manera que, sabiendo lo que ocupa cada *environment*, ver qué objetos están dentro de él y poner en el campo *content* los objetos que componen al *environment* dado.

Una vez terminado el proceso de parseo inverso, se mandaría este mensaje al módulo de Python, o parte del LLM, para que lo guarde.

9.1.5. Poder configurar los valores de los parámetros que va a usar el LLM para la generación y poder visualizar con que parámetros exactos ha generado el nivel pedido en un momento dado

Otro cambio o adición interesante sería ver los parámetros que el propio LLM ha usado para generar la respuesta que ha mandado a UE5. Para lograr esto tendríamos que hacer que el módulo de Python, o parte del LLM, mande aparte de la propia respuesta, sus parámetros de generación como el valor del calor al generar esa respuesta, el top-k, el máximo de tokens/longitud, entre otros. Para que el usuario, en caso de estar interesado, pueda ver estos parámetros.

También se puede modificar la interfaz del widget para que el propio usuario, si está interesado en ello, pueda elegir con qué valores de los parámetros de generación, como por ejemplo, el calor, quiere que genere la respuesta sobre la cual ha hecho la petición.

Para poder implementar todo esto, sería necesario cambiar la interfaz gráfica, el widget como se ha mencionado anteriormente. Para que tenga dos nuevos botones que, por ejemplo, se pueden llamar "mostrar estadísticas" y "configurar estadísticas". Por ejemplo, cuando se haya generado un mapa, al pulsar "mostrar estadísticas" se muestran los valores de los parámetros que ha usado para generar ese nivel. A continuación se explicarán más en profundidad como se implementarían estos nuevos botones.

El botón "configurar estadísticas" al presionarse, abriría una nueva pantalla donde se configurarán los valores para los parámetros que usará el LLM para generar el nivel pedido, con unos valores por defecto que se mandan al LLM si el usuario decide no configurar esta parte. En caso de que sí, el usuario introduciría esos valores en los correspondientes campos y al mandar una nueva petición, se mandaría como mensaje la petición y los parámetros para que el LLM los trate y pueda generar una respuesta correcta.

El botón "mostrar estadísticas" como se ha mencionado antes, al pulsar el botón se mostrarían los datos con los que el LLM ha generado ese nivel, para ello el módulo de Python, o parte del LLM, necesitaría mandar aparte del JSON, sus estadísticas para que puedan ser mostradas desde la interfaz gráfica.

9.1.6. Poder pedir al LLM que escale o rote ciertos objectos elegidos por el usuario

Una última opción para añadir a la aplicación es que el usuario pueda hacer una petición sobre el mapa o nivel que ya ha generado el LLM para que escale o rote una textura o un conjunto de ellas, elegidas por el usuario. Para que se adapten mejor al contexto en el que se encuentran.

Este cambio puede parecer simple, pero para poder llevarlo a cabo se tendría que cambiar lo siguiente:

- La interfaz de la aplicación, en este caso; el widget. Para que sea capaz de mandar otro tipo de mensaje en el cual solo aparecen los objetos seleccionados y puedes elegir re-escalarlos o rotarlos según lo que el usuario ponga en el mensaje
- El paso de mensajes. Para que pueda soportar este nuevo tipo de mensaje
- El módulo de Python, o parte del LLM. Para saber cómo tratar estos mensajes y entrenarlo o decirle como tiene que responder
- Hacer un nuevo parseo donde se seleccionan los objetos que el usuario había seleccionado previamente y les cambie los atributos del tamaño o la rotación según lo que había pedido el usuario

Para conseguir todo esto, los principales problemas son: entrenar al LLM para que responda bien a estos mensajes, donde también necesita información de cómo son los atributos de los actores de UE5 y con qué tipo de valores se pueden modificar, además de qué rango de valores maneja UE5 para los atributos de sus actores. También es necesario tener algún medio para saber a qué objetos se refiere una vez conseguido el mensaje mandado desde UE5.

Para poder soportar todos estos cambios, habría que cambiar la estructura del JSON descrita en el proyecto que se usa para generar los niveles, donde se añada un campo "id" que tendrá que ser único para cada *JSON object* dentro del archivo JSON. Cuando se renderice el archivo, a cada textura se le asignará ese "id" como una variable de esta. Con esta información, se podrá saber qué objeto o grupos de objetos selecciona el usuario para modificarlos.

El proceso para mandar el mensaje descrito al principio de esta sección es el siguiente: el usuario deberá pulsar primero un nuevo botón llamado "escalar/rotar" del widget que contiene la interfaz de usuario, una vez pulsado. El usuario puede ir eligiendo objetos que desea modificar; cuando pulsa uno, el sistema recoge para la elaboración del mensaje que "id" le corresponde a esa textura. Una vez elegidos los objetos, el usuario deberá poner un mensaje con una estructura como esta:

- Escalame a más pequeño/grande los siguientes objetos:
- Rota estos objetos:

Después, cuando el usuario haya pulsado "enviar", el sistema añadirá al mensaje previo la lista de los "id" que el usuario ha seleccionado, separados por comas para mandarlos al LLM.

Luego el LLM le asignará una serie de valores para que los objetos sean rotados o escalados, cuando mande el mensaje y la aplicación lo reciba.
Para poder manejar el mensaje en UE5 tendremos que definir la estructura que tendrá el mensaje. Esta estructura puede ser, por ejemplo, un *JSON array* de *JSON objects* donde cada objeto tiene su "id" y una lista con los valores modificados, si son más de uno.

El mensaje, una vez recibido, se debe de tratar de la siguiente manera en UE5: se lee cada elemento de la lista, con el id se va al nivel a buscar el objeto correspondiente y se modifican sus valores con respecto a los valores mandados en el mensaje, eso con cada uno de los elementos del mensaje.

Esta mejora requiere de muchos cambios a la aplicación que por falta de tiempo para poder probar todo y falta de recursos para poder entrenar o experimentar con el LLM para que soporte estos mensajes no ha sido posible implementarla.

Introduction

As the world of video games has evolved and grown, they have become more complex. The first games were arcade games whose main objective was for players to complete more and more screens, whereas today video games can focus on conveying a message through storytelling and the depth of their characters, the immersion of the player achieved through graphics and music, and the playability and fluidity of the map design, levels and mechanics.

Each part of a video game requires a team of professionals who focus on different areas, some of which did not originate from video games, such as art, narrative and music. However, the design and creation of levels and maps did originate from video games. And, as in other disciplines, these two involve a long and complex process that is also creative, requiring the creation and linking of concepts in such a way that, after numerous revisions, everything is cohesive, which can involve a considerable amount of time and resources.

This has led to many games, both small and large productions, now considered great successes, having to delay their eagerly awaited release, such as Horizon Zero Dawn (*Guerrilla Games*, 2017) which, since it was first announced at E3 in 2015, the public had to wait until 2017 to get their hands on it; Kingdom Hearts III (*Square Enix*, 2018), which was announced in 2013 and did not go on sale until 2018; and Hollow Knight (*Team Cherry*, 2017) which was funded by a Kickstarter campaign that ended in 2014 and did not appear on the market until 2017.

Our work aims to create a tool that, by leveraging the computing power of LLMs, such as the possibly best-known ChatGPT with GPT-4, helps users generate maps and levels to alleviate the burden of map creation in video games, although it could also be scalable to other fields where map generation or replication is useful.

Motivation

Level design, also known as mapping, is one of the fundamental pillars of video game creation. This process not only directly impacts gameplay and user experience, but also influences narrative, exploration and immersion within the game universe. Due to its importance, level design often requires a considerable investment of time, human effort and financial resources, which can be a particularly significant obstacle

for small studios or independent developers. In this context, the advancement of large-scale language models (LLMs) offers an innovative opportunity to reduce some of this burden. Thanks to its content generation, autonomous learning and context understanding capabilities, an LLM can be used as an assistant in the generation of maps or levels in a more efficient way. The proposal consists of integrating these capabilities into an intuitive visual environment that allows the user to interact with the model in a direct and understandable way, without the need for in-depth technical knowledge.

One of the main objectives is to accelerate the trial and error phases typical of level design, as well as to facilitate the creation of coherent and aesthetic spaces, taking advantage of the model's ability to detect patterns and generate meaningful variations. In this way, the designer can focus on the creative and conceptual part of development, while the LLM assists with repetitive, structural or inspirational tasks. By creating a tool that is accessible, manageable and has a moderate learning curve, the door is opened to new user profiles interested in these fields: people with creative ideas but who have traditionally been limited by a lack of technical knowledge or economic barriers. Thus, the project not only seeks to improve development efficiency, but also to encourage the inclusion and participation of new talent in the field of video game design.

In summary, this proposal sits at the intersection of artificial intelligence and human creativity, with the aim of making level design a more accessible, agile and inspiring process, serving both professional developers and enthusiasts with limited resources.

Hypothesis

With the above in mind and the powerful technology currently available to LLMs. It is believed that by creating a simple and intuitive programme for designers that saves resources in basic areas, exploiting the power of LLMs in these fields, large developers can allocate the rest of their resources to other areas, and small creators can give free rein to their creativity without worrying so much about the budget. With this in mind, we can propose the following hypothesis:

> *Hypothesis 1: By exploiting the power of LLMs and using their capabilities, it is believed that they can generate playable, aesthetic and creative levels in a simple and inexpensive way.*

Objectives

The overall objective of this work is to create a tool that helps video game developers and designers when creating levels and maps in Unreal Engine 5. Below is a more detailed list of the specific objectives of this work.

1. Conceptual design of a tool for using LLMs to generate levels given any LLM, it reads a description of the idea to be represented, entered by the user from a

chat within Unreal itself, this message is then modified with formatting data to ensure that the LLM output is correct, which is then reconnected to Unreal Engine 5, which is responsible for reading the JSON and generating the level. From the editor, this level can be modified.

2. Determine the correct output format that the LLM must provide for the JSON Given any LLM, in the Python script it is possible to ensure that the output provided conforms to the JSON format required to parse the response in Unreal Engine 5.

3. Establish communication between the LLM and the application. Given any LLM, achieve two-way communication between the LLM and the different parts of the application in order to receive responses from the LLM and send requests in a fluid manner. The process of receiving responses and sending requests can be done indefinitely. Manage the LLM output and generate the level according to this output Given a well-formed JSON according to objective 2, with that JSON it is possible to generate a level given a process that receives a file such as the JSON mentioned above and generates a level as faithful as possible to the file. If this is not possible, it will still generate a level.

4. Conducting user tests Establish experiments to analyze the effectiveness of the application with respect to the manual design. These tests should include both versions of the level designs, the manual one and the one made with the application, as well as a brief context of the user on whom the experiment was performed.

Work plan

This section will describe the work plan to be followed to achieve the objectives described in the previous section.

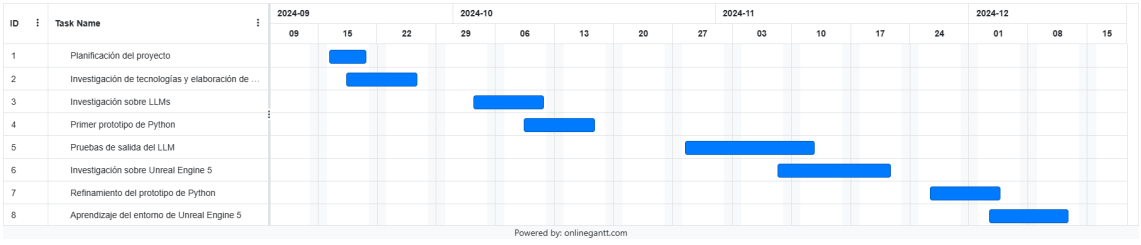

Figure 9.1: Gantt chart with the project schedule for September to December

Table 9.1: Work plan

Month	Objectives
September 2024	Plan the project, identify the technologies to be used, detail the idea
October 2024	Learn how to use LLMs locally, first prototype of the Python module that returned responses
November 2024	Test the prototype to see the types of responses, investigate Unreal Engine 5 technologies
December 2024	Refine the first prototype of the Python module, learn Unreal Engine 5 technologies
January 2025	Finish the basics of the Python module, first Unreal Engine 5 prototype
February 2025	Establish a connection between both parts, refine Unreal Engine 5 prototype
March 2025	Refine the connection between the parts, add textures to the project, begin writing the report
April 2025	Complete the level generation project, continue writing
May 2025	Obtain results from user evaluation and complete the report
June 2025	Present the project

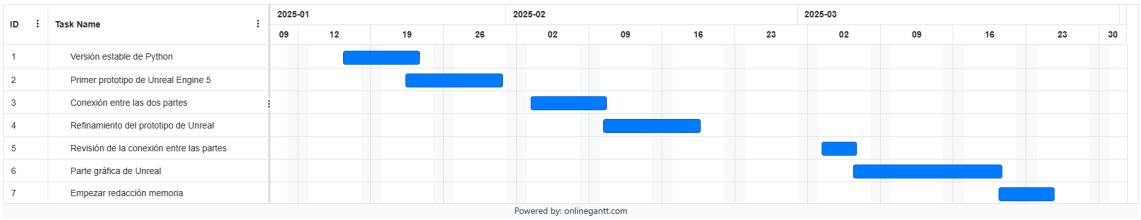

Figure 9.2: Gantt chart with the project schedule for January to March

Figure 9.3: Gantt chart with the project schedule for April to June

Structure of the rest of the document

This section provides an outline of the rest of the document's contents, such as its chapters and sections. It also shows the order in which the contents of the document are presented.

In Chapter 2 of the document, a state-of-the-art analysis is presented, exploring the different technologies related to this project. In addition, characteristics of these technologies that are not explicitly addressed in this project are presented. A study of work related to our project is also carried out to address different points of view on a common problem.

Chapter 3 of the project presents the internal architecture of this project. It defines the main modules of the project and how they are connected to each other.

Chapter 4 discusses the Python module and the LLM. It describes the classes used, the functions of the classes, and the rules used by the LLM. It explains the process of how it evolved from start to finish.

Chapter 5 is divided into two different sections: one on development in Unreal Engine 5 and the other on the textures used in the project. It delves into what type of Unreal Engine 5 project has been used and what the relevant content is within it. It also discusses the textures used in the project and the reasons for choosing those textures.

Chapter 6 analyses the metrics used to study the LLM's responses, divided into several sections, where the topic is introduced, how the data was obtained, how the process was carried out, and then the results are presented and analysed.

Chapter 7 describes the user evaluation process, explaining what actions users will perform for the evaluation and what will be evaluated. Once this has been described, the results obtained with each user are presented, along with some observations and, finally, the overall results.

Chapter 8 presents a discussion of the advantages and disadvantages of this project.

Chapter 9 discusses the conclusions and general results of the entire project, followed by a section presenting several lines of work to be carried out in the future that could not be completed within this timeframe due to various problems.

Finally, Chapter 10 contains the user manual for this project.

Conclusions and Future Work

After metrics and user testing, we can assure that our tool has a bright future. Even though it does not achieve the expected results, since our initial goal was to generate a complete, detailed space. After testing the first versions, we realised that this was not possible. So we reduced it to the creation of a smaller space, as we couldn't achieve a higher level of quality with our resources. We focused on making it easy to use, visual and useful for the user, and we have succeeded. Although the result may sometimes differ greatly from what the user expected from their request to the machine, we have managed to create a tool to help designers, which was our main objective.

We have managed to create a tool that is simple, intuitive and quick to use, highly scalable and visually appealing, as you run it from Unreal itself where you can see the same changes. We believe this, and more importantly, our test users think so too, which is why we believe that the vast majority of our problems are reduced to the use of small, if not tiny, LLMs compared to those we all know or have used, even for much simpler tasks than this one. If we used a very powerful LLM such as GPT-3.5 or 4, we could obtain highly specific results in much less time, making testing, changes and updates very efficient. Despite all that, we have to affirm that our project doesn't meet the requirements to approve the aforementioned hipotesis since even though we managed to create levels in a cheap and intuitive way, we don't think they are creative, aestethically pleasing nor playable.

It also has potential and room for improvement. Keep in mind that we have had to learn on the fly throughout the project, which has led to exponentially better results over time. With a larger and more professional team, more development time, or both, we are confident that it can become a truly useful tool in a professional environment, especially if we begin to delve deeper into the subject, for example, by training LLMs for this purpose or retraining them as you progress in the creation of your environment, generating cohesion with the previous one, or generating creation templates that the LLM would apply, or simply generating a quick idea or a cascade of them until the creator is convinced by one or comes up with a similar idea and can modify or tweak it.

For the rest of the limitations that do directly concern us, we can see ideas and improvements in future work that, if implemented, would further improve the user experience and the results obtained.

Future lines of work

The project aimed to create an application that would generate playable levels to help designers in their level creation work. The aim was for our application to make an initial sketch of the level and, through user tweaks and several more queries to the LLM, create a level that the user would expect. We have managed to create an application that generates initial sketches of levels, but we have not managed to allow the user to modify the level and, once modified, send it back to generate something more accurate. Having said this, we will now present several lines of work which, if implemented, would improve the performance of the application with respect to the initial objectives of this project and could even exceed them.
We will now explain the different lines that have been considered for the future of the project.

Rendering in editor mode

One consideration that was made at some point during the development of this tool was that it should work in UE5 editor mode because it is natural for a tool that helps to design, in this case levels, to work in editor mode. This is so that the different models that have been generated by LLM can be modified and retouched.

For the application to work in UE5 editor mode and not in game mode, we would have had to create our own text parser, as finding a parser that converts a String to a JSON file is complicated, given that the ones on the market that fit this project's budget are all in game mode. Therefore, this functionality could not be fully implemented. However, we will discuss what changes are necessary, in theory, for this functionality to be implemented.

The first thing that would need to be changed is the way the server is started in UE5 and when the connection to the client is established. In this case, it should be when the level itself is opened. To do this, the SocketerSpawner class would need to be modified so that it works in UE5's native OnConstruction method. This way, the server would run directly when the level is opened. It is also necessary to ensure that multiple servers are not opened if the level is run several times or if the blueprint of JSONSpawner is opened, as this could lead to errors. To do this, the code ensures that it can only be executed once, and for one level.

Then you would have to change the blueprint of JSONSpawner so that it runs in editor mode and not in game mode. To do this, you have to change when the blueprint is activated. Currently, it is done with the Event tick node, which only works when game mode is active.

To change this, you need to create a node that we can call Editor Tick. To create this Editor tick, you would need to implement a Blueprint Interface that contains a function that can be called at edit time, in Editor mode. Called Editor Tick.

Then, a blueprint would be created that inherits from EditorUtilityActor, which has the characteristics of "allow tick before play", "tick even paused" and "is Editor only Actor". This enables operation in editor mode.

In the "Construction Script" function of this blueprint that inherits from EditorUtil-

ityActor, ensure that the parent actor implements the Blueprint Interface explained above and also assign a timer for the "Editor Tick" function of the Blueprint Interface. However, to use delays within functions, it is necessary to create your own timer, since when running in edit time, Unreal's native timers do not run, making it necessary to keep track of the time elapsed by calculating it for each frame that passes in edit time, using your own method.

Once this is created, the blueprint SpawnerJSON in its "Construction Script" to add a Add child actor component node where the child class is the blueprint that inherits from EditorUtilityActor and then implement the previously created interface within this blueprint.

Scalable texture or sprite base

Another change that could not be implemented due to a lack of resources and knowledge in the area of 3D modelling is having a larger texture base.

If a larger texture base were to be used, the model proposed for texture rendering would not support this scalability, so it would need to be changed. We will explore how it could be changed to improve the scalability of the rendering.

One possibility would be to have a database where each texture is stored with the words it can be related to. This way, a texture can refer to different synonymous words and the system would not be so restrictive with the use of words, as synonyms can be added more easily. Once the database with the data described above has been created, in the application, when it comes to rendering a word, it would be done as follows within a Blueprint or a C++ class:

- First, a query would be made to find the textures associated with that word.
- The result may be one or more. If the word is not found in the database, a default texture would be chosen, as is currently done.
- If there is only one, that texture is taken to the Spawn actor from class node and the system will render that texture in the appropriate position. This step does not change.
- If there are several, the query would return a table or a set with the corresponding textures, which would be numbered from 0 to the size of the table - 1, with 0 being the first texture and the size of the table - 1 corresponding to the last texture. One would be chosen at random from among these, and then the same procedure would be followed as when there is only one result.

This way, if we want to increase the number of textures in the application, we only need to add it to the database and load the corresponding texture into the system, making it more scalable.

Training an LLM to specialise in level generation

Have an LLM trained, or rather specialised, in generating JSON from natural language prompts. The data on which it bases its responses contains information

about a variety of different biomes, each with different heights, objects, and other features that make up a biome. With an LLM trained for this purpose, we can be sure that more diverse, extensive, and detailed levels can be generated. We can assume that training this proposed model would require a fairly large amount of data, which may be quite difficult to obtain.

If we train this hypothetical model to give answers in a given format and with the variety of knowledge described above, we believe that the answers obtained with this LLM will be more accurate than those obtained with the current model. Above all, it is estimated that its accuracy in the spatial location of the objects that make up its answers would improve considerably in relation to the current model.

Load pre-made levels and the possibility of saving levels generated by the LLM that may or may not be modified by the user.

Another change or improvement that could be implemented would be that, once the user has obtained a map that meets their needs or satisfies what they requested from the application, they could have the option of saving it within the application if they wish and then, if at any given time they decide that they need the previously saved map, they could find it alongside the maps that were loaded at the beginning as "examples".

To implement this, the application interface must be changed in the widget that is responsible for sending the messages given by the application user to the LLM. It should have two other buttons: a "save" button and a "load" button, which would be located next to the "send" button.

The "load" button will open a list with the names of pre-made maps, or in other words, well-formed JSON files. When the user clicks on one of the names, the system will send a message to the Python module, or part of the LLM, to send that JSON directly without the module requesting a response from the LLM. The application then renders that JSON like any other. Any maps saved by the user will also be displayed, if applicable, alongside the list of pre-made maps.

The "save" button will save what is currently in the world, with the changes added by the user to that map. This process is explained later. Once that is done, it will send it to the Python module, or part of the LLM, where it will be saved with the rest of the pre-made maps. These maps, as they are well constructed, can be used as material for fine-tuning the LLM that the user is currently using, apart from what is already sent by default.

This improves the quality of the responses obtained because there is more material for fine-tuning.

The problem arises when asking how the map, which may be modified by the user, should be saved, because if it is not modified, this process would be simpler since it would only be necessary to temporarily save each message sent by the LLM with the JSON to be rendered until the user sends another message through the widget, which would then discard that level generation, and if they wanted to save the

message, it would simply be saved with the rest of the pre-made levels. But with this idea comes a problem, which is that the user can modify what is generated by the LLM.

In this case, the process of saving the created map becomes much more complicated because we do not know what textures the user has added and for what purpose, as there are two possibilities: they may have decided to add a texture as an environment or they may have added a texture as an object, this would cause problems when parsing the map in reverse because we would not know how to structure the JSON so that it would not contain errors.

In addition, it is not known which environment each object belongs to because, as currently implemented, it is not possible to know which area each environment occupies in UE5. Therefore, it is not possible to know which objects are contained in each environment. If this information were available, each texture would have to be parsed to obtain its data:

- The type of object, whether it is an object or an environment
- The description in this process can be left blank
- To save the data in relation to the name, you would need to know which word it comes from or, in other words, do the reverse of the texture rendering process, seeing which texture it has and assigning it the word associated with that corresponding texture
- For the position, if it is an object, there is no problem because you would somehow obtain the x and y coordinates of where the texture is placed on the map and assign it to the corresponding field.

  The problem arises, as mentioned above, with textures with the type field with the value environment because it is not possible to know how much area it occupies in order to know which points are the extremes and to be able to put those coordinates in the JSON file. If you did know, you would obtain that data and place it in the JSON file to send it as a message to the part of the LLM where this map would be stored

The JSON must then be structured so that, knowing what each environment occupies, we can see which objects are inside it and put the objects that make up the given environment in the content field.

Once the reverse parsing process is complete, this message would be sent to the Python module, or part of the LLM, for storage.

Be able to configure the values of the parameters that the LLM will use for generation and be able to visualise the exact parameters used to generate the requested level at a given moment

Another interesting change or addition would be to see the parameters that the LLM itself has used to generate the response it has sent to UE5. To achieve this, we

would have to make the Python module, or part of the LLM, send, apart from the response itself, its generation parameters such as the heat value when generating that response, the top-k, the maximum tokens/length, among others. This would allow the user, if interested, to view these parameters.

The widget interface can also be modified so that the user, if interested, can choose which generation parameter values, such as heat, they want to use to generate the response to their request.

In order to implement all this, it would be necessary to change the graphical interface, the widget as mentioned above. So that it has two new buttons that, for example, can be called "show statistics" and "configure statistics". For example, when a map has been generated, pressing "show statistics" displays the values of the parameters used to generate that level. Below, we will explain in more detail how these new buttons would be implemented.

When the "configure statistics" button is pressed, a new screen would open where the values for the parameters that the LLM will use to generate the requested level would be configured, with default values that are sent to the LLM if the user decides not to configure this part. If the user decides to configure these values, they would enter them in the corresponding fields and, when sending a new request, the request and parameters would be sent as a message to the LLM for processing and generation of a correct response.

The "show statistics" button, as mentioned above, when clicked, would display the data with which the LLM generated that level. To do this, the Python module, or part of the LLM, would need to send its statistics separately from the JSON so that they can be displayed from the graphical interface.

Ask the LLM to scale or rotate certain objects chosen by the user

A final option to add to the application is for the user to make a request on the map or level already generated by the LLM to scale or rotate a texture or set of textures chosen by the user. This is so that they better fit the context in which they are found.

This change may seem simple, but in order to implement it, the following would have to be changed:

- The application interface, in this case, the widget. So that it is able to send another type of message in which only the selected objects appear and you can choose to re-scale or rotate them according to what the user puts in the message
- Message passing. So that it can support this new type of message
- The Python module, or part of the LLM. To know how to handle these messages and train it or tell it how to respond

- Perform a new parsing where the objects that the user had previously selected are selected and their size or rotation attributes are changed according to what the user had requested

To achieve all this, the main problems are: training the LLM to respond correctly to these messages, where it also needs information about the attributes of UE5 actors and the types of values they can be modified with, as well as the range of values UE5 handles for its actor attributes. It is also necessary to have some means of knowing which objects are being referred to once the message sent from UE5 has been received.

In order to support all these changes, the JSON structure described in the project used to generate the levels would have to be changed, adding an "id" field that would have to be unique for each JSON object within the JSON file. When the file is rendered, each texture would be assigned that "id" as a variable. With this information, it will be possible to know which object or group of objects the user selects to modify.

The process for sending the message described at the beginning of this section is as follows: the user must first press a new button called "scale/rotate" on the widget containing the user interface. The user can select the objects they want to modify; when they click on one, the system collects the "id" corresponding to that texture for the message. Once the objects have been selected, the user must enter a message with a structure like this:

- Scale the following objects smaller/larger:
- Rotate these objects:

Then, when the user clicks "send", the system will add the list of 'ids' that the user has selected to the previous message, separated by commas, to send them to the LLM.

The LLM will then assign a series of values for the objects to be rotated or scaled when the message is sent and the application receives it.
In order to handle the message in UE5, we will need to define the structure of the message. This structure could be, for example, a JSON array of JSON objects where each object has its own "id" and a list of modified values, if there is more than one.

Once received, the message must be handled as follows in UE5: each element in the list is read, the id is used to go to the level to find the corresponding object, and its values are modified according to the values sent in the message. This is done for each of the elements in the message.

This improvement requires many changes to the application, which have not been possible to implement due to a lack of time to test everything and a lack of resources to train or experiment with the LLM to support these messages.

Contribuciones Personales

Gonzalo Bertolín Díez

Gonzalo, al igual que el resto del equipo, ha contribuido con gran compromiso y dedicación al desarrollo de este proyecto, cumpliendo con los objetivos dentro del plazo establecido. Con esta actitud, asumió la responsabilidad de las siguientes tareas:

Gonzalo estuvo encargado de la comunicación con el tutor en gran parte de la duración del trabajo, enviando correos y concertando las reuniones hasta que por motivos de falta de tiempo cedió esta responsabilidad a Alfonso. En estos correos exponía dudas y notificaba del progreso que se iba realizando en el trabajo.

En el comienzo del proyecto colaboró con David para poder ejecutar las primeras LLMs en Python usando la interfaz de Ollama, además de establecer el repositorio de Github usado durante la duración de todo el proyecto además de encargarse de la creación del proyecto de Unreal y su manejo y configuración de un *gitignore* para que fuera posible subirlo al programa de control de versiones.

Tras ser descartada la interfaz de Ollama para usar los *Llamafiles*, Gonzalo fue quién consiguió implementar la comunicación entre Python y la LLM.

Su principal tarea fue encargarse de la arquitectura e intercomunicación de los diferentes programas, tanto al principio usando el plugin *Socketer* como más adelante cuando desarrolló los *scripts* que comunican las aplicaciones como son *SocketerC*, *SocketerSpawner* y su integración dentro del *blueprint SpawnerJSON* en Unreal y *SocketerClient* por la parte de Python. Además, fue el encargado de implementar el *widget* con el cuál se realizan las peticiones a la LLM.

Una vez la arquitectura estaba funcionando, Gonzalo se dedicó a la realización de pruebas del sistema, ayudando a Alfonso a depurar la clase *SpawnerJSON*, además de encargarse después de redactar y realizar las pruebas con usuarios y recopilando los datos.

Gonzalo también fue el encargado de intentar la migración del modo de juego al modo de edición para el proyecto, consiguiendo rehacer toda la arquitectura para

este modo, además del nuevo tipo de *widget* de utilidad, pero cancelandolo al final debido al bloqueo que suponía la ausencia de un *parser* de texto que funcione en este modo.

En cuanto a la redacción de la memoria Gonzalo fue colaborador en la sección de estado de la cuestión, investigando sobre los diferentes algoritmos de generación procedimental y transcribiendo texto que Alfonso había escrito en *Google Documents*, además de redactar los capítulos de arquitectura y valoración de usuarios.

Alfonso García-Carrasco Puerta

Alfonso, al igual que sus compañeros, ha dado lo mejor de sí mismo para sacar adelante este proyecto de una manera satisfactoria, óptima y en el tiempo estipulado para el proyecto. Con esto en mente se encargó de las siguientes tareas:

Alfonso ha mantenido una comunicación fluida con el tutor a lo largo de todo el trabajo, mandando correos y yendo a las reuniones establecidas y solicitadas al tutor, especialmente en el último tercio del trabajo cuando el resto de integrantes, por trabajo, no pudieron asistir a estas reuniones. Donde en estas reuniones expusó de manera asertiva los problemas que había tenido él y también comunicó los problemas que tuvieron sus compañeros en las reuniones que no pudieron asistir.

Al principio del proyecto ayudó en la investigación sobre el uso de LLMs en local, así como en los primeros usos de esta parte en local para ver como funcionaba juntos al resto sus compañeros. También colaboró con la especificación de las normas de la estructura que debería tener el JSON que devuelva el LLM que se este ejecutando en ese momento.

Alfonso también fue el principal encargado de la investigación de las tecnologías de Unreal Engine 5, investigando como funcionaba el entorno. Además de investigar el funcionamiento de los blueprints de Unreal Engine 5 y como programar con ellos, a su vez de cuales eran sus limitaciones en el proyecto.

Además fue el principal encargado de encontrar tecnologías ya creadas que se pudiesen incluir en el proyecto, en este caso, en forma de *plugins*, donde al final de la búsqueda solo necesitamos un *plugin* en este fue *VaRest* que se encargaba de todo el manejo del parseo de los JSON, así como la decodificación de un String a un JSON. Con esta tecnología creó el blueprint *SpawnerJSON* donde este blueprint es el encargado principal de la conversión de un JSON a un nivel en Unreal Engine 5, durante el proceso Alfonso se dió cuenta de que por la naturaleza de la estructura del JSON, previamente diseñada, era necesario hacer este blueprint recursivo, investigó sobre ello y lo implementó.

Alfonso a lo largo del proyecto también investigó sobre como se podrían ejecutar los modelos más grandes de LLMs que había, pero durante el proceso se dio cuenta de que no era viable ejecutar un modelo tan grande en local. Por lo que descartó la idea comunicándolo al resto del equipo.

Luego una vez terminadas las conexiones entre el LLM, Python y el proyecto de Unreal Engine 5 fue el principal encargado de la elección de *assets* para el proyecto. Que al final se decidió por un entorno de ciudades.

Alfonso también se encargó del proceso para elegir que texturas corresponden a cada palabra. Para ese proceso, Alfonso investigó varias maneras de hacerlo, una de ellas fue usar la distancia Levenshtein con la palabra del objeto para asignarle una textura pero se descartó la idea y le propusieron que se hiciese desde Python, donde delegó la tarea en David.

Alfonso fue el encargado de la redacción del primer borrador de la memoria, escrita en *Google documents* que contenía: el resumen, la introducción y el estado del Arte o el estado de la Cuestión, donde luego se encargó de ampliar el último. También redactó el capítulo de Desarrollo del sistema y la sección de líneas de trabajo a futuro.

David Toral Castiñeira

David, al igual que sus compañeros, ha buscado aprovechar al máximo todas las herramientas al alcance de su mano para conseguir el mejor resultado posible para este proyecto. Con esto en mente se encargó de las siguientes tareas:

En un primer momento fue junto a Gonzalo el encargado de conseguir encauzar el proyecto con las primeras ejecuciones de distintas *LLMs*, usando la interfaz *Ollama*, tras buscar,encontrar y conseguir implementar en Python la primera versión "pre Unreal".

Una vez conseguida la ejecución fue el principal encargado en probar la LLM hasta conseguir dar con la manera de que el LLM mostrase un JSON de la manera que buscábamos, con sus respectivos campos nombres y organización también definidos por David, el filtrado del *output* para que se obtuviese la salida necesaria y su procesado y guardado en vistas a su posterior uso.

Una vez abandonada la interfaz Ollama junto a Gonzalo buscó la manera de utilizar la *Api* de *ChatGPT* y su contribuyó en su implementación y modificación consiguiendo una aplicación completamente funcional y análoga a la previa pero mucho más escalable.

Apoyó en las labores de conexión a Gonzalo buscando la manera de poder conectar mediante servidores todas las plataformas, para enviar los mensajes pertinentes consiguiendo conectarlo.

Y cuando el proyecto ya estaba controlado derivó su trabajo a ser el máximo encargado del funcionamiento, actualización, pruebas y aplicación de metodologías de las *LLMs*.

En cuanto a la actualización fue el encargado de probar distintas LLMs e ir cambiando la aplicación de Python todo lo necesario para seguir ejecutando en base a los cambios del proyecto.

Las pruebas son todos los cambios finales y temporales que fueron sucediendo, por ejemplo la aplicación y descarte de la librería *strict_json* el filtrado de LLMs, entradas, salidas, etc...

Las metodologías son la prueba y descarte de múltiples cambios que modifican y mejoran la funcionalidad del LLM, el *Rag* (descartado porque no suponía mejoras) y *Reflection* , la aplicación del filtrado de palabras para una mejora de funcionamiento de los *assets* mediante los *embeddings* probándolos en español (descartado por precisión) o en inglés.

Más allá de la propia aplicación de Python de la cual se encargó, fue el creador de aplicaciones con versiones de prueba que reducían la complejidad del programa para poder aprovechar mucho más el tiempo a la hora de implementar distintos cambios o hacer pruebas.

Respecto a la memoria David se encargó de la revisión transcripción y escritura de forma general, incluyendo la modificación de los párrafos menos coherentes y su diseño.

Y por último con la versión final se encargó de las pruebas respectivas para la obtención de las métricas, LLMs, tablas y gráficas.

Bibliografía

Abdullah, F., Taveekitworachai, P., Dewantoro, M. F., Thawonmas, R., Togelius, J. y Renz, J. The first chatgpt4pcg competition. *IEEE Transactions on Games*, vol. 16(4), páginas 971–980, 2024.

Cloudflare. ¿qué son los modelos llm? | grandes modelos de lenguaje. 2025. Consultado el 24 de abril de 2025.

Fournier, A., Fussell, D. y Carpenter, L. Computer rendering of stochastic models. *Communications of the ACM*, vol. 25(6), páginas 371–384, 1982.

Gallotta, R., Todd, G., Zammit, M., Earle, S., Liapis, A., Togelius, J. y Yannakakis, G. N. Large language models and games: A survey and roadmap. *IEEE Transactions on Games*, páginas 1–18, 2024.

Hochreiter, S. y Schmidhuber, J. Long short-term memory. *Neural Comput*, vol. 9, páginas 1735–1780, 2016.

Hu, C., Zhao, Y. y Liu, J. Game generation via large language models. En *2024 IEEE Conference on Games (CoG)*, páginas 1–4. 2024.

Liu, J., Snodgrass, S., Khalifa, A. et al. Deep learning for procedural content generation. *Neural Computing and Applications*, vol. 33, páginas 19–37, 2021.

Montoya, J. A. ¿comó se diseña un motor de videojuegos? 2017.

Perlin, K. An image synthesizer. *ACM SIGGRAPH Computer Graphics*, vol. 19(3), páginas 287–296, 1985.

Perlin, K. Improving noise. `https://mrl.cs.nyu.edu/~perlin/paper445.pdf`, 2002. Accessed: 2025-05-04.

Rouhiainen, L. Inteligencia artificial. *Madrid: Alienta Editorial*, páginas 20–21, 2018.

Sudhakaran, S., González-Duque, M., Freiberger, M., Glanois, C., Najarro, E. y Risi, S. Mariogpt: Open-ended text2level generation through large language models. En *Advances in Neural Information Processing Systems* (editado por A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt y S. Levine), vol. 36, páginas 54213–54227. Curran Associates, Inc., 2023.

Thorn, A. *Game engine design and implementation*. Jones & Bartlett Publishers, 2011.

Todd, G., Earle, S., Nasir, M. U., Green, M. C. y Togelius, J. Level generation through large language models. En *Proceedings of the 18th International Conference on the Foundations of Digital Games*, FDG '23. Association for Computing Machinery, New York, NY, USA, 2023. ISBN 9781450398558.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u. y Polosukhin, I. Attention is all you need. En *Advances in Neural Information Processing Systems* (editado por I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan y R. Garnett), vol. 30. Curran Associates, Inc., 2017.

Apéndice A

Manual de uso

Para poder ejecutar la aplicación correctamente hay que hacer lo siguiente:

1. Ejecutar la LLM en la consola de comandos, en caso de que sea un ejecutable, simplemente hacer doble *click*.
2. Abrir el proyecto de *Unreal* para ello ejecutar *Display.uproject*.
3. Una vez abierto el proyecto de *Unreal Engine*, abrir el nivel llamado *Display* del almacén de contenido.
4. Presionar al botón de *play* en la parte superior del motor. Esto abrirá el servidor.
5. Abrir en un editor de Python como puede ser *Pycharm* el proyecto entero, desde allí ejecutar en este orden, *ManagerServer.py*, *LLM.py*, esperar a que en la consola aparezca "Waiting data from Unreal..." y por último ejecutar *SocketerClien.py*.
6. Por último, para realizar peticiones escribir en el widget de texto la petición en inglés y darle al botón de "Enviar" una única vez. **Nota:** El texto no se borrar tras enviarlo.