UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

GENERACIÓN DE SHADERS UTILIZANDO LARGE LANGUAGE MODELS

Informe de Proyecto de Grado presentado por

Santiago Ferrando

Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación

Supervisores

Christian Clark
Tomás Laurenzo

Montevideo, 01 de Abril de 2025

RESUMEN

La generación automática de código a partir de descripciones en lenguaje natural es un área en rápida evolución, impulsada por los avances en modelos de lenguaje de gran escala (LLMs). Estudios previos han demostrado la eficacia de los LLMs en la generación de código en lenguajes tradicionales, pero la creación automática de shaders, programas especializados que se ejecutan en la GPU para calcular efectos visuales, sigue siendo un reto significativo debido a la necesidad de generar resultados que se correspondan visualmente con la descripción ingresada. Para abordar estas dificultades, el proyecto aplicó técnicas como Fine-Tuning y Retrieval-Augmented Generation (RAG), junto con el uso de bases de datos vectoriales, para mejorar la generación de shaders a partir de prompts en lenguaje natural. Los resultados muestran que, si bien el Fine-Tuning sobre un corpus reducido no fue suficiente para mejorar sustancialmente la generación de shaders, la integración de RAG permitió mejorar la precisión y coherencia del código generado. Este trabajo ofrece una plataforma interactiva y útil para la creación de visuales gráficos a partir de descripciones textuales, reduciendo la barrera técnica para usuarios sin experiencia en programación gráfica. Además, al incorporar retroalimentación de los usuarios sobre la calidad gráfica de los resultados, el sistema no solo mejora su rendimiento, sino que también contribuye a un corpus abierto, etiquetado y en constante expansión.

ABSTRACT

The automatic generation of code from natural language descriptions is a rapidly evolving field, driven by advances in large-scale language models (LLMs). Previous studies have demonstrated the effectiveness of LLMs in generating code for traditional programming languages, but automatic shader generation—specialized programs that run on GPUs to calculate visual effects—remains a significant challenge due to the need to generate results that visually correspond to the input description. To address these difficulties, the project applied techniques such as Fine-Tuning and Retrieval-Augmented Generation (RAG), alongside the use of vector databases, to improve shader generation from natural language prompts. The results show that while Fine-Tuning on a limited corpus was insufficient to substantially enhance shader generation, the integration of RAG improved the accuracy and coherence of the generated code. This work offers an interactive and practical platform for creating graphical visuals from textual descriptions, reducing the technical barrier for users without experience in graphical programming. Furthermore, by incorporating user feedback on the graphical quality of outputs, the system not only improves its performance but also contributes to an expanding, labelled, open corpus.

Tabla de Contenidos

1. Introducción

La generación automática de código ha sido un área de creciente interés en el campo de la inteligencia artificial, impulsada por el desarrollo de modelos de gran escala diseñados para procesar y generar texto en lenguaje natural, conocidos como *Large Language Models* (LLMs). Estos modelos han demostrado una notable capacidad para generar código en diversos lenguajes de programación, especialmente para aplicaciones en CPU[1]. Herramientas como *GitHub Copilot* [1] han evidenciado la utilidad práctica de estos modelos, ayudando a los desarrolladores a escribir código de manera más eficiente y efectiva [2], sin embargo, la generación de código con un componente visual marcado, como los *shaders*, ha sido menos explorada en comparación con otras aplicaciones de estos modelos [3].

Los shaders son programas que se ejecutan en la Unidad de Procesamiento Gráfico (GPU)[2] y se utilizan para calcular efectos visuales como la iluminación, el color y las texturas en gráficos por computadora. Debido a su naturaleza gráfica y altamente paralela, la generación automática de shaders presenta un desafío único, ya que requiere un control preciso sobre los resultados visuales.

Este proyecto aborda ese vacío mediante la creación de una plataforma, **Demokritos**, que utiliza LLMs para generar shaders a partir de descripciones en lenguaje natural. A lo largo del desarrollo de la plataforma se utilizaron técnicas avanzadas como *fine-tuning* y *Retrieval-Augmented Generation* (RAG) para optimizar los resultados. El *fine-tuning* es un proceso mediante el cual un modelo pre entrenado se ajusta utilizando un conjunto de datos específico, permitiendo que aprenda patrones y estructuras propias de un dominio particular, como los shaders. Por otro lado, RAG es una técnica que mejora la generación automática al recuperar ejemplos relevantes desde una base de datos vectorial, proporcionando al modelo un contexto adicional para guiar su respuesta. Estas técnicas, en conjunto, buscan maximizar la precisión y relevancia de los shaders generados.

Como primera aproximación al uso de LLMs, se llevó a cabo una prueba de concepto en el marco de una instalación artística en colaboración con el artista Fernando Foglino [4]. Esta experiencia consistió en la interacción entre dos LLMs, donde uno había sido ajustado mediante *fine-tuning* para adquirir conocimientos sobre artistas uruguayos. Este experimento sirvió para explorar las capacidades de personalización de los modelos y establecer las bases para el desarrollo del proyecto.

[1] CPU (*Central Processing Unit*): Unidad Central de Procesamiento. Es el componente principal de un sistema informático encargado de ejecutar instrucciones y realizar cálculos. A diferencia de la **GPU**, que está optimizada para procesar múltiples operaciones en paralelo, la CPU está diseñada para manejar tareas generales de manera secuencial y con alta flexibilidad.

[2] GPU (*Graphics Processing Unit*): Unidad de Procesamiento Gráfico. Es un componente de hardware especializado en realizar cálculos masivos en paralelo, principalmente orientados al procesamiento gráfico y, más recientemente, a tareas de inteligencia artificial y aprendizaje profundo.

El carácter colaborativo de Demokritos juega un rol central en su desarrollo. Inicialmente, se diseñó un sitio web para recopilar descripciones de shaders generadas por los usuarios, lo que permitió construir un corpus representativo y abierto. Esta base de ejemplos ha sido fundamental tanto para el *fine-tune* del modelo como para el sistema de recuperación de información implementado. Además, la plataforma no solo genera shaders, sino que también almacena los resultados generados por los usuarios, enriqueciendo continuamente el corpus con nuevos datos etiquetados y contextualizados.

El objetivo general del proyecto es proporcionar una plataforma que democratice la generación de shaders, permitiendo a usuarios sin experiencia en programación gráfica crear visuales gráficos de manera rápida y eficiente. Los resultados son evaluados desde una perspectiva técnica, mediante la capacidad del sistema para generar código funcional y compilable, y desde una perspectiva colaborativa, incorporando retroalimentación directa de los usuarios sobre la calidad gráfica de los resultados obtenidos. Además, la constante expansión del corpus gracias a la interacción con los usuarios asegura que la plataforma siga evolucionando.

Estructura del Documento

Este documento se organiza de la siguiente manera:

- **Capítulo 2: Estudio del Estado del Arte** – Se presenta un análisis de las tecnologías previas utilizadas en la generación de código con LLMs, cubriendo desde *transformers* hasta técnicas como el *fine-tuning* y RAG, y su aplicación en la generación de shaders.
- **Capítulo 3: Desarrollo del Proyecto** – Se describe en detalle el desarrollo de la plataforma Demokritos, el proceso de recopilación de corpus, la prueba de concepto y la implementación de la técnica RAG.
- **Capítulo 4: Evaluación y Análisis de Resultados** – Aquí se presentan los resultados de la generación automática de shaders, tanto en términos técnicos como de evaluación subjetiva por parte de los usuarios, con un análisis de las métricas relevantes como la tasa de compilación y la valoración visual.
- **Capítulo 5: Conclusiones y Trabajo a Futuro** – Se abordan las conclusiones clave del proyecto, junto con propuestas para el trabajo a futuro, como la expansión del corpus, el uso de otros modelos (como *LLaMA*[3]), y la optimización de las técnicas de generación automática de shaders.

[3] LLaMA (Large Language Model Meta AI): Modelo de lenguaje de gran escala desarrollado por Meta (Facebook) optimizado para eficiencia y accesibilidad en investigación.

2. Estudio del Estado del Arte

2.1 Introducción

El desarrollo de este proyecto, centrado en la generación automática de shaders a partir del lenguaje natural, se fundamenta en un conjunto de avances tecnológicos recientes que han revolucionado el campo de la inteligencia artificial. Este capítulo está dedicado a examinar el estado del arte en las tecnologías que sustentan el proyecto, tales como los shaders y su rol en los gráficos, los *Large Language Models* (LLMs), y técnicas avanzadas de procesamiento como la *Retrieval Augmented Generation* (RAG).

2.2 Shaders

Los shaders son pequeños programas que se ejecutan en la Unidad de Procesamiento Gráfico (GPU), en lugar de la Unidad Central de Procesamiento (CPU), y están diseñados para realizar cálculos especializados relacionados con gráficos por computadora [5]. Estos programas son fundamentales para crear efectos visuales, como sombras, reflejos, texturas, iluminación y muchos otros fenómenos gráficos que observamos en videojuegos, simulaciones y entornos 3D [6].

Funcionamiento de Shaders en la GPU

Los shaders se ejecutan en la GPU, que está diseñada para maximizar las tareas que se ejecutan de forma paralela [7]. Esto significa que, mientras la CPU se enfoca en ejecutar instrucciones de manera secuencial (una después de la otra), la GPU puede ejecutar miles de cálculos simultáneamente. Esto es especialmente útil en gráficos, donde cada vértice o cada píxel puede ser procesado de manera independiente de los demás [8].

El propósito principal de los shaders es ofrecer a los desarrolladores un control dinámico sobre el renderizado de imágenes en pantalla. A diferencia de los métodos gráficos tradicionales y más estáticos, los shaders permiten manipular diversos aspectos de los gráficos, como la luz, los colores y las interacciones con los objetos en una escena.

El proceso típico de renderizado involucra pasar datos de la CPU a la GPU, como los vértices de los objetos 3D y las texturas. Los shaders en la GPU transforman estos datos y los combinan con información sobre la iluminación y la cámara para generar la imagen final que se muestra en la pantalla.

Tipos de Shaders

Existen varios tipos de shaders, cada uno con una función específica en el *pipeline* gráfico[4]:

1. **Vertex Shaders**: Se encargan de transformar las coordenadas de los vértices de un objeto 3D al espacio de pantalla (*clip space*). También pueden calcular atributos adicionales, como normales para iluminación y coordenadas de textura, que serán usados por los *fragment shaders*.
2. **Geometry Shaders**: Permiten la manipulación de geometría, agregando o eliminando vértices antes de la rasterización[5]. Son útiles para efectos como generación de superficies a partir de puntos o creación de bordes extruidos, aunque su uso es menos frecuente debido a su costo computacional.
3. **Fragment (Pixel) Shaders**: Operan sobre cada fragmento[6] generado después de la rasterización, determinando el color final de cada píxel en la imagen renderizada. Aplican cálculos detallados sobre texturas, iluminación y efectos visuales, lo que los convierte en una parte clave del proceso de renderizado.
4. **Compute Shaders**: No forman parte del *pipeline* de renderizado tradicional, pero permiten ejecutar tareas de cómputo altamente paralelizadas en la GPU. Se utilizan en simulaciones físicas, procesamiento de imágenes, cálculos generales de alto rendimiento y técnicas avanzadas de renderizado como *ray tracing*, que simula el comportamiento físico de la luz para generar imágenes realistas.

Ejemplo de Shader

A continuación se presenta un ejemplo básico de un fragment shader en GLSL[7] [9], que simplemente asigna un color a cada píxel:

```
void main() {
   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); // Hex Rojo
}
```

[4] Pipeline gráfico: Secuencia de etapas en la GPU encargadas de procesar y renderizar gráficos en tiempo real, desde la transformación de geometría hasta la rasterización y sombreado de píxeles.
[5] Rasterización: Proceso en el pipeline gráfico que convierte primitivas geométricas (como triángulos) en fragmentos, los cuales luego son procesados por los *fragment* shaders para determinar el color de cada píxel en la imagen final
[6] Fragmento: Unidad de información generada en el proceso de rasterización que contiene datos sobre un posible píxel en la imagen final, incluyendo color, profundidad y coordenadas de textura.
[7] GLSL (OpenGL Shading Language) es un lenguaje de programación específico utilizado para escribir shaders en la API de OpenGL. Está diseñado para ejecutarse en la GPU y es ampliamente utilizado en gráficos por computadora para controlar el comportamiento visual de los objetos, como el color, la iluminación, las texturas y otros efectos visuales.

En este ejemplo, el shader pinta todos los píxeles en rojo (con valores RGB de 1.0, 0.0, 0.0). Un shader real incluiría cálculos mucho más complejos para determinar el color final de cada píxel, como el manejo de luces, texturas y sombras.

2.2.1 La GPU y su Papel en la Inteligencia Artificial

Además de su importancia en gráficos, la capacidad de procesamiento paralelo de la GPU ha sido crucial en el auge de la inteligencia artificial (IA), especialmente en áreas como el *deep learning*[8]. Los modelos de IA, como las redes neuronales[9] profundas, requieren procesar grandes cantidades de datos y ejecutar millones de operaciones matemáticas en paralelo. Esto convierte a las GPUs en una opción ideal para entrenar modelos de aprendizaje profundo, ya que pueden procesar muchas neuronas y capas de una red simultáneamente.

La habilidad de la GPU para manejar estos procesos complejos en paralelo ha permitido avances significativos en áreas como el procesamiento del lenguaje natural (NLP) y la visión por computadora[10], que son parte integral de las aplicaciones de IA modernas, incluidos los *Large Language Models* (LLMs).

Tanto en el ámbito gráfico como en la inteligencia artificial, la GPU ha demostrado ser una herramienta esencial para acelerar el procesamiento paralelo y manejar tareas de alta complejidad de manera eficiente [10].

[8] Deep Learning es una rama del aprendizaje automático (machine learning) que utiliza redes neuronales artificiales con múltiples capas (profundas) para modelar y resolver problemas complejos. Es especialmente eficaz en tareas como el reconocimiento de imágenes, el procesamiento del lenguaje natural, y la generación de texto y código.

[9] Las redes neuronales son modelos computacionales inspirados en la estructura y funcionamiento del cerebro humano. Están formadas por capas de nodos (neuronas) interconectadas, que procesan información y pueden aprender a realizar tareas específicas mediante el ajuste de pesos en las conexiones entre estas neuronas.

[10] Visión por computadora: Área de la inteligencia artificial que permite a las máquinas interpretar y procesar imágenes y videos, simulando la capacidad de visión humana.

2.3 Machine Learning y su Aplicación en la Generación de Código

El aprendizaje automático (*Machine Learning*, ML) ha transformado muchas áreas del desarrollo de software, incluyendo la generación automática de código. Antes de la llegada de modelos de lenguaje a gran escala (LLMs), varios enfoques de ML ya habían demostrado ser útiles para tareas como la traducción entre lenguajes de programación [11], la reparación automática de programas [12] y la generación de documentación de código [13].

Enfoques Tempranos en la Generación de Código

En los primeros trabajos sobre generación automática de código, los investigadores experimentaron con redes neuronales recurrentes (RNN) [14] y arquitecturas basadas en árboles sintácticos abstractos (AST) [15]. Estos modelos se entrenaban para transformar descripciones en lenguaje natural en código funcional, o para convertir código de un lenguaje de programación a otro [16].

Uno de los paradigmas más comunes era la generación de código a partir de descripciones textuales, también conocida como *description-to-code* [17]. Un enfoque popular en este ámbito fue el uso de *programming by example* (PBE) [18], donde los modelos aprendían a generar programas basados en ejemplos de entrada y salida. Este tipo de técnica, junto con la reparación automática de programas, mostró que el aprendizaje automático podía ayudar a los desarrolladores no solo en la generación de código, sino también en la mejora de la calidad y el mantenimiento del software.

Modelos y Técnicas Pre-LLM

Entre los modelos más utilizados en estos primeros enfoques de ML para la generación de código se encuentran las redes neuronales recurrentes, redes neuronales convolucionales (CNNs) [19] y técnicas híbridas que combinaban diferentes tipos de arquitecturas, como CNNs para procesar imágenes de interfaces gráficas y RNNs para generar código a partir de descripciones textuales [16]. Las RNNs, y en particular las variantes de LSTM (Long Short-Term Memory) [20], fueron las más empleadas para tareas como la generación de documentación y la traducción entre lenguajes de programación.

2.4 *Large Language Models* (LLMs)

Los *Large Language Models* (LLMs) son modelos de inteligencia artificial diseñados para procesar y generar texto basado en grandes cantidades de datos. Estos modelos se entrenan utilizando técnicas de *machine learning* sobre vastos corpus de texto, lo que les permite aprender las reglas subyacentes del lenguaje natural, incluyendo gramática, sintaxis, semántica y el contexto [21]. LLMs como GPT-3 y GPT-4o representan un avance significativo en el procesamiento del lenguaje natural (NLP) debido a su capacidad para generar respuestas coherentes a partir de entradas en lenguaje natural [22] [23].

¿Cómo funcionan los LLMs?

Los LLMs se basan en arquitecturas de redes neuronales profundas, específicamente en los *transformers*, introducidos en el artículo *Attention is All You Need* de Vaswani et al. en 2017 [24]. Los *transformers* se destacan por su capacidad para manejar secuencias de texto largas de manera más eficiente que modelos previos, como las redes neuronales recurrentes (*Recurrent Neural Networks*, RNN) o las *Long Short-Term Memory* (LSTM).

La arquitectura del *transformer* está compuesta por dos componentes principales: el *encoder* y el *decoder*. El *encoder* toma una secuencia de entrada, como una oración, y la transforma en una representación interna que captura las características más relevantes de dicha secuencia. Esta representación es utilizada posteriormente por el *decoder*, que genera la secuencia de salida (por ejemplo, la traducción o respuesta a una pregunta). En algunos modelos de LLM, como GPT-3 o GPT-4o, solo se utiliza la parte del *decoder* [21] ya que están diseñados para la generación de texto a partir de un *prompt*, en lugar de (específicamente) tareas de traducción [25] (por ejemplo), por lo que no requieren las funciones de codificación de entrada típicas de un *encoder*.

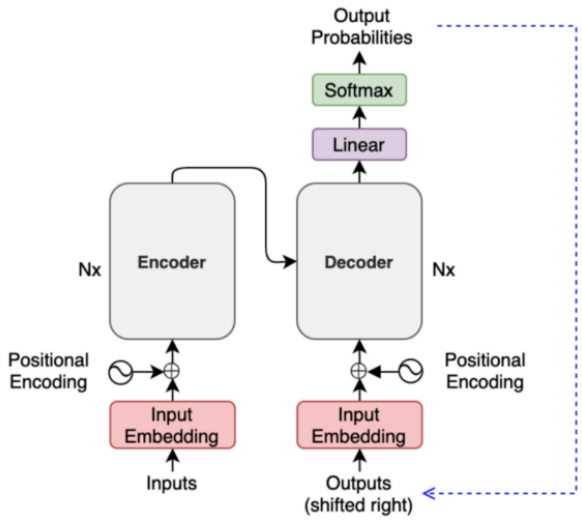


Figura 1 - Representación visual de la arquitectura del modelo Transformer propuesto en el artículo Attention Is All You Need. [26]

El mecanismo clave detrás de los transformers es la autoatención (*self-attention*), que permite al modelo enfocarse en diferentes partes de una secuencia de texto, ponderando la relevancia de cada palabra o grupo de palabras con respecto a las demás, sin importar si las palabras relevantes están cerca o lejos unas de otras en la oración. A diferencia de los modelos anteriores, que sólo consideraban las palabras más cercanas para interpretar una oración, los *transformers* pueden analizar toda la secuencia de texto simultáneamente, entendiendo mejor el contexto global.

A través de este mecanismo, el modelo evalúa la relación entre palabras de forma dinámica, incluso en secuencias largas, capturando no solo el significado de una palabra individual, sino también el contexto en el que se utiliza.

Este proceso se lleva a cabo utilizando tres componentes clave para cada palabra en la oración:

1. **Query (consulta)**: la palabra que está "preguntando" qué otras palabras son relevantes para ella.
2. **Key (clave)**: las palabras de la oración con las que la *Query* quiere relacionarse.
3. **Value (valor)**: el contenido semántico de cada palabra.

Cada palabra en la oración tiene su propio conjunto de *Query*, *Key* y *Value*. A través de un cálculo de similitud entre la *Query* y las *Keys*, el modelo determina qué palabras tienen más relación con la *Query* y, por lo tanto, deberían influir más en la predicción de la siguiente palabra en la secuencia.

Para entender esto mejor, pensemos en cómo *self-attention* procesa la oración: *"El perro que saltó sobre la cerca ladraba fuerte"*. Aunque la palabra "perro" y "ladraba" están separadas por varias palabras, el mecanismo de *self-attention* permite al modelo entender que ambas están estrechamente relacionadas. Para ello, asigna un mayor "peso" o relevancia a esta relación durante la generación de texto, haciendo que el modelo se enfoque más en esta conexión importante.

Cálculo de la atención

Este proceso de atención se resume en una operación matemática: para cada palabra, se calcula una "puntuación de atención" que indica qué tan relevante es cada una de las otras palabras en la oración. Esta puntuación determina el peso que cada palabra tiene en el resultado final. Luego, se realiza una combinación ponderada de todas las palabras para decidir qué palabra sigue en la secuencia[11].

El resultado de este mecanismo es que el *transformer* puede interpretar el contexto completo de una oración mucho mejor que los modelos secuenciales tradicionales, lo que le permite generar texto de manera más coherente y precisa. Esto es particularmente útil para la generación de código fuente de shaders, donde es crucial que el modelo entienda el contexto técnico para generar fragmentos de código funcionales y eficientes.

¿De Dónde Obtienen los Datos?

Los modelos de lenguaje como GPT se entrenan con cantidades masivas de datos textuales, tomados de diversas fuentes, como libros, sitios web, artículos de investigación y otras formas de texto público [21]. Este proceso de entrenamiento permite que los modelos "aprendan" las

[11] Bosquejo de un posible cálculo del vector de atención en el anexo

reglas del lenguaje, incluidos el significado de las palabras, la gramática y cómo estructurar el texto.

Por ejemplo, GPT-3 fue entrenado en un conjunto de datos inmenso que incluye una amplia variedad de temas, desde literatura hasta código de programación, lo que le permite generar texto en diferentes contextos. Sin embargo, estos modelos no "entienden" el texto de la manera en que lo haría un humano; en cambio, simplemente aprenden patrones en los datos y los replican cuando generan nuevo contenido [22].

Relación entre datos y la capacidad de los transformers

Cuando un transformer genera texto, no está inventando algo desde cero, sino que está "prediciendo" la próxima palabra o secuencia de palabras basada en las probabilidades que aprendió durante su entrenamiento. Si el modelo ha visto millones de ejemplos de cómo se utiliza una palabra en diferentes contextos, puede hacer una predicción más precisa sobre qué palabra sigue en una secuencia dada.

Por ejemplo, si el modelo ha visto muchas veces cómo se usa la palabra "perro" en contextos como "El perro corre en el parque" o “el perro ladraba”, será capaz de predecir que, tras la palabra "perro", podría venir "corre" o "ladraba", en lugar de algo que no tiene sentido como "avión".

Al entrenar a estos modelos con código, como en el caso de GLSL, el modelo aprende a asociar descripciones en lenguaje natural con fragmentos de código funcionales, aunque requiere más precisión en las descripciones para generar shaders adecuados debido a la naturaleza técnica del código.

Relevancia de los LLMs en la Generación de Código

Uno de los usos más destacados de los LLMs es su capacidad para generar código en varios lenguajes de programación. Los LLMs pueden interpretar instrucciones en lenguaje natural y traducirlas en bloques de código que cumplen con las especificaciones dadas. Esto se debe a que durante su fase de entrenamiento, los LLMs son expuestos a grandes conjuntos de datos que incluyen fragmentos de código y descripciones relacionadas. Como resultado, son capaces de aprender las reglas y estructuras subyacentes de diferentes lenguajes de programación, como Python, JavaScript, y más recientemente, lenguajes especializados como GLSL para la creación de shaders.

En el contexto de este proyecto, los LLMs juegan un papel crucial, ya que permiten la generación automática de código fuente de shaders a partir de descripciones en lenguaje natural. Esta capacidad permite a los usuarios no especializados en programación generar efectos visuales complejos sin necesidad de escribir manualmente el código shader, acelerando así el proceso creativo y reduciendo la barrera de entrada para quienes no tienen experiencia en programación gráfica. A medida que la investigación avanza, la utilización de

técnicas como *fine-tuning* y *Retrieval-Augmented Generation* (RAG) sigue ampliando las capacidades de los LLMs en la creación de código especializado como los shaders.

2.4.1 Fine-Tuning de LLMs

GPT-3 y GPT-4o tienen la capacidad de generar texto coherente y realizar tareas complejas gracias a su entrenamiento en grandes conjuntos de datos. Sin embargo, aunque estos modelos son muy poderosos, pueden necesitar ajustes adicionales para desempeñarse de manera óptima en tareas específicas. Este ajuste fino, o *fine-tuning*, se refiere al proceso de adaptar un modelo pre entrenado para una tarea particular, utilizando un conjunto de datos más pequeño y especializado [27].

¿Qué es el *Fine-Tuning*?

El *fine-tuning* es un proceso en el que un modelo pre entrenado se entrena nuevamente en un conjunto de datos específico relacionado con la tarea que se desea realizar. Durante este proceso, los parámetros del modelo se ajustan para mejorar el rendimiento en dicha tarea. En lugar de entrenar un modelo desde cero, lo cual requiere enormes cantidades de datos y recursos computacionales, el *fine-tuning* permite aprovechar el conocimiento general ya aprendido por el modelo durante su pre entrenamiento y afinarlo para casos de uso más especializados.

El concepto de *fine-tuning* se sustenta en la idea de que los modelos pre entrenados ya poseen una comprensión general del lenguaje (o código, en el caso de la generación de programas), pero necesitan ajustes específicos para mejorar en tareas más concretas. Esto es especialmente útil cuando el dominio de la tarea es altamente técnico o especializado, como la generación de shaders en gráficos por computadora.

El trabajo "*Language Models are Few-Shot Learners*" (Brown et al., 2020) [22] sentó las bases para comprender cómo los LLMs como GPT-3 pueden ser efectivos sin la necesidad de grandes cantidades de datos de entrenamiento para cada tarea nueva. Este trabajo mostró que los LLMs pueden realizar razonablemente bien en una amplia gama de tareas con pocos ejemplos, o incluso sin ejemplos específicos (conocido como *zero-shot learning*). Aunque los LLMs ya son capaces de generalizar bastante bien, el *fine-tuning* se vuelve una herramienta clave cuando se requiere un nivel mayor de precisión en una tarea muy especializada.

En este contexto, el *fine-tuning* es particularmente relevante cuando el modelo necesita generar código de shader, que es una tarea técnica y no común en los datos generales de entrenamiento.

2.5 Los LLMs Definen el Estado del Arte en la Generación de Código

En los últimos años, la generación automática de código ha avanzado significativamente gracias a los modelos de lenguaje de gran escala (LLMs). Herramientas como *GitHub Copilot*, basadas en modelos como *Codex* [28], han demostrado que es posible automatizar parte del proceso de desarrollo de software al generar código en respuesta a instrucciones en lenguaje natural. *Copilot* sugiere líneas de código basándose en un vasto corpus de código fuente disponible en repositorios públicos, reduciendo la necesidad de que los desarrolladores escriban manualmente cada línea de código.

Además de *Copilot*, otros sistemas como *AlphaCode* [29] de *DeepMind* han mostrado la capacidad de resolver problemas complejos en competencias de programación, marcando un hito en la automatización de tareas que requieren razonamiento lógico y estructuración del código. Estos avances abren la puerta a nuevos horizontes en la generación de código, no solo en lenguajes tradicionales como *Python* o *JavaScript*, sino también en áreas más especializadas como la generación de shaders, donde el código gráfico requiere una mayor precisión y especialización.

Los desafíos que enfrentan los LLMs en esta área incluyen la correcta interpretación de instrucciones en lenguaje natural y la generación de código que sea eficiente, legible y fácil de mantener. Sin embargo, técnicas como el *fine-tuning* y el *Retrieval-Augmented Generation* han demostrado avances importantes. Estas técnicas permiten que los modelos no solo generen código más preciso y contextualizado, sino que también sean capaces de ajustarse a tareas especializadas como la generación de shaders gráficos, mejorando la calidad del código generado y su aplicabilidad en contextos visuales específicos.

2.6 Utilización de *Retrieval-Augmented Generation* (RAG)

Retrieval-Augmented Generation (RAG) es una técnica que combina la generación automática de texto con la recuperación de información relevante desde una base de datos. En lugar de depender únicamente del modelo de lenguaje para generar texto o código, RAG permite que el modelo acceda a una base de conocimiento externa, como una base de datos vectorial, para obtener contexto adicional que mejore la precisión y relevancia de las respuestas generadas [30].

2.6.1 Base de Datos Vectoriales

Una base de datos vectorial es fundamental para el éxito de RAG, ya que permite que el sistema busque y recupere información semánticamente relevante. En lugar de utilizar búsquedas basadas en palabras clave o texto exacto, las bases de datos vectoriales almacenan representaciones numéricas de los datos, conocidas como vectores (*word embeddings* [31]). Estas representaciones capturan las características semánticas de las entradas, lo que permite búsquedas mucho más precisas y rápidas cuando el sistema necesita encontrar ejemplos similares o información relevante para un prompt dado.

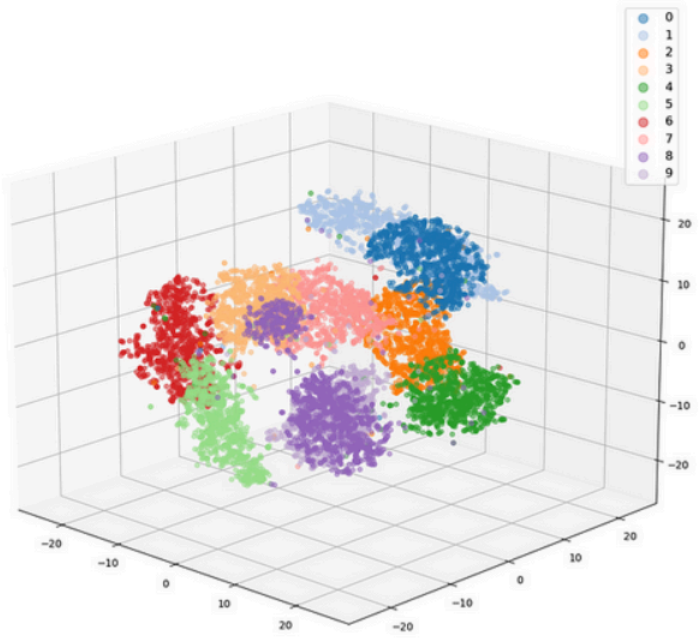


Figura 2 - Representación visual de una base de datos vectorial [32]

La figura muestra cómo se agrupan vectores en un espacio tridimensional según algún criterio definido.

En este trabajo, los vectores son definidos según su semántica, por lo que al vectorizar los prompts, aquellos que sean semánticamente similares estarán más próximos en el espacio vectorial.

2.6.1 ¿Cómo Funciona el Modelo RAG?

El flujo básico de RAG consiste en tres fases principales:

1. **Ingesta (*Ingestion*):** En esta fase, se recopilan y preparan los datos que se utilizarán en el proceso de recuperación. Los datos son transformados en vectores semánticos y almacenados en una base de datos vectorial, donde cada entrada es representada por un vector que captura sus características semánticas. Este paso es crucial para garantizar que la base de datos esté estructurada de manera eficiente para búsquedas rápidas y precisas en las fases posteriores.
2. **Recuperación (*Retrieval*):** Dado un *input* o prompt, se utiliza una base de datos para recuperar información relevante. En lugar de realizar búsquedas textuales tradicionales, RAG emplea bases de datos vectoriales, donde cada entrada está representada por un vector que captura sus características semánticas. Esto permite realizar búsquedas eficientes basadas en similitud de contenido.
3. **Generación (*Generation*):** Una vez recuperada la información relevante, esta se pasa al modelo de lenguaje (en este caso, un *Large Language Model*) como contexto adicional. El modelo genera una respuesta o código, utilizando tanto el prompt original como la información recuperada, lo que mejora significativamente la coherencia y relevancia del resultado final.

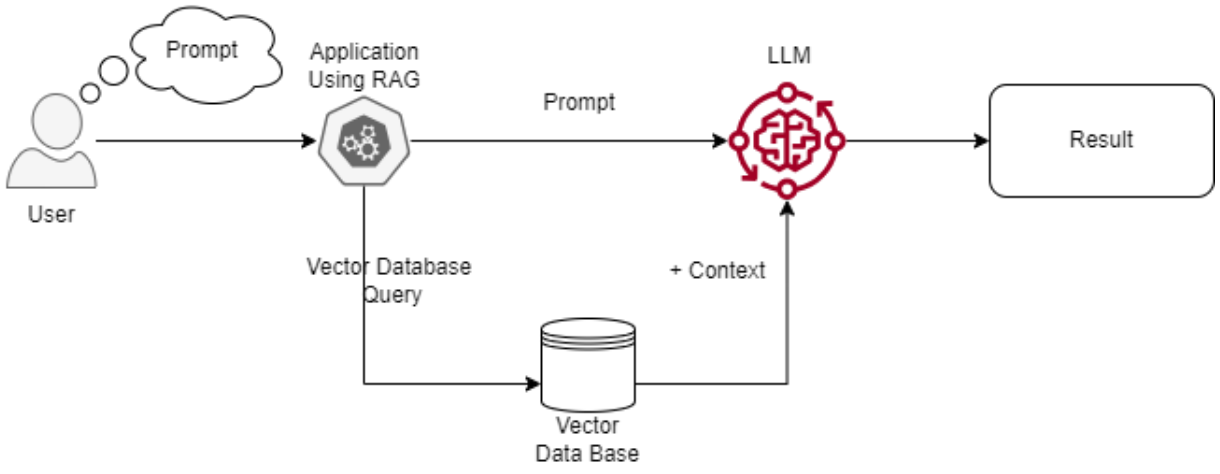


Figura 3 - Representación visual del modelo de Retrieval Augmented Generation utilizado en este proyecto.

2.7 Modelos GPT y su Evolución

En los últimos años, los *Large Language Models* (LLMs) han experimentado una evolución acelerada, con la familia de modelos GPT (*Generative Pretrained Transformer*) de *OpenAI* a la vanguardia de estos avances. Estos desarrollos han mejorado significativamente la capacidad de los modelos para generar texto y código de manera más precisa y coherente.

GPT-1: El Comienzo de la Generación de Texto Pre-entrenada

El modelo GPT-1, presentado en 2018 [33], fue el primer intento de *OpenAI* de utilizar el enfoque de pre-entrenamiento no supervisado seguido de ajuste fino supervisado para tareas específicas de procesamiento del lenguaje natural. Este modelo demostró que los *transformers* pre-entrenados en grandes cantidades de texto podían lograr un buen desempeño en varias tareas de NLP sin la necesidad de arquitecturas específicas para cada tarea.

GPT-2: Escalando los Modelos de Lenguaje

En 2019, *OpenAI* lanzó GPT-2 [21], un modelo significativamente más grande con 1.5 mil millones de parámetros. GPT-2 mostró habilidades avanzadas en la generación de texto coherente y continuaciones de texto dadas ciertas entradas. Sin embargo, debido a preocupaciones sobre su potencial uso indebido, inicialmente se lanzó una versión limitada, y posteriormente se liberó el modelo completo. GPT-2 demostró que aumentar el tamaño del modelo podía mejorar sustancialmente su capacidad para generar texto de alta calidad.

GPT-3: Un Salto Cuantitativo en Capacidad

Lanzado en 2020, GPT-3 representó un hito en el campo de los LLMs. Con 175 mil millones de parámetros, GPT-3 amplió enormemente las capacidades de generación de texto y mostró habilidades impresionantes en tareas como traducción, respuesta a preguntas y, notablemente, generación de código en lenguajes como *Python* y *JavaScript*. Su capacidad para realizar *few-shot* learning [22], donde el modelo puede adaptarse a nuevas tareas con pocos ejemplos, demostró el potencial de los LLMs para aplicaciones versátiles.

ChatGPT y GPT-3.5: Enfocados en la Interacción Conversacional

En noviembre de 2022, *OpenAI* introdujo *ChatGPT*, basado en la serie de modelos GPT-3.5. *ChatGPT* fue afinado utilizando técnicas de Aprendizaje por Refuerzo con Retroalimentación Humana, lo que permitió generar respuestas más alineadas con las expectativas humanas y mejorar la interacción conversacional. *ChatGPT* facilitó diálogos más naturales y coherentes, ampliando su aplicación en áreas como asistencia virtual, educación y soporte técnico. Además, mostró mejoras en la generación de código y en la capacidad para seguir instrucciones complejas [34].

GPT-4o: Mejoras en Comprensión y Creatividad

En marzo de 2023, *OpenAI* lanzó GPT-4o, la última iteración de la familia GPT hasta la fecha. GPT-4o trajo mejoras significativas en términos de comprensión, razonamiento y creatividad [23]. Aunque *OpenAI* no ha revelado el número exacto de parámetros, GPT-4o es capaz de procesar entradas más largas y manejar tareas más complejas que sus predecesores. Una característica destacada es su capacidad multimodal (en versiones limitadas), permitiendo interpretar tanto texto como imágenes como entradas, aunque la generación de imágenes aún no es una característica generalizada.

En el contexto de la generación de código, GPT-4o ha mostrado mejoras en la precisión y en el manejo de lenguajes de programación más especializados. Esto incluye una mejor comprensión de instrucciones en lenguaje natural para generar código en lenguajes como GLSL.

2.8 Antecedentes en la Generación de Shaders con Modelos de Lenguaje

A continuación, se presentan algunos trabajos previos relevantes en esta área. Si bien estos desarrollos son contemporáneos a este proyecto y no fueron utilizados como referencia directa, ilustran el creciente interés en la generación automática de shaders mediante modelos de lenguaje:

- **AIShader (Keijiro Takahashi)**[35]: Este proyecto experimental explora el uso de modelos de lenguaje para generar código de shaders en *Unity*[12]. *AIShader* utiliza una interfaz que permite a los usuarios describir un efecto visual en texto y recibe como salida un shader en *HLSL*[13], mostrando una integración directa con motores gráficos.
- **MaxiShader - GLSL Coding Assistant**[36]: Un asistente basado en modelos de lenguaje diseñado específicamente para ayudar a los desarrolladores a generar código GLSL. Su propósito principal es facilitar la escritura de shaders proporcionando sugerencias y generando fragmentos de código a partir de *prompts* mediante una interfaz estilo chat.
- **GLSL Code Generator (JIT.Dev)**[37]: Un servicio basado en inteligencia artificial que permite la generación de código GLSL mediante instrucciones en lenguaje natural. La plataforma busca agilizar el proceso de creación de shaders en entornos gráficos, aunque su enfoque está más orientado a la generación asistida que a la producción de código completamente funcional sin intervención humana.

[12] Unity: Motor de desarrollo de videojuegos y gráficos en tiempo real ampliamente utilizado en la industria. Permite la creación de experiencias interactivas en 2D, 3D, realidad aumentada (AR) y realidad virtual (VR).
[13] HLSL (High-Level Shader Language): Lenguaje de programación desarrollado por Microsoft para escribir shaders en DirectX. Permite la creación de efectos gráficos avanzados y se utiliza ampliamente en aplicaciones de renderizado en tiempo real.

- **LLM Shader Toy (John Pertoft)**[38]: Un experimento que utiliza los modelos de lenguaje de *OpenAI*, permitiendo la generación automática de shaders a partir de descripciones en lenguaje natural. Este proyecto ofrece una interfaz interactiva donde los usuarios pueden probar directamente los *shaders* generados. Necesita de una *OpenAI API Key*[14] para poder utilizarse.
- **LLM Shader Art (Sabrina Verhage)**[39]: Proyecto que explora la generación de arte visual a través de shaders creados con modelos de lenguaje. La plataforma permite a los usuarios ingresar prompts y visualizar el código generado en tiempo real, enfatizando la dimensión artística de la generación automática de shaders. La interfaz tipo chat de la aplicación permite al usuario refinar y modificar el código generado de manera iterativa mediante nuevos prompts, facilitando la exploración y ajuste del resultado visual.

Estos proyectos han demostrado que los modelos de lenguaje pueden ser utilizados como herramientas para la creación de shaders, aunque la mayoría aún dependen de la intervención manual para corregir errores sintácticos y mejorar la coherencia del código generado. En este contexto, Demokritos busca avanzar en esta línea, integrando técnicas como RAG y bases de datos vectoriales para mejorar la precisión de la generación de shaders y reducir la dependencia de ajustes manuales.

[14] *OpenAI API Key*: Clave de acceso única proporcionada por *OpenAI* que permite a los desarrolladores autenticar y utilizar los servicios de la API de *OpenAI*, incluyendo modelos de lenguaje como GPT, para la generación de texto, código y otras aplicaciones.

3. Desarrollo del Proyecto

3.1 Introducción

El desarrollo de este proyecto siguió una progresión incremental de varias fases, comenzando con la creación de una plataforma web para la recolección de corpus y avanzando hacia la implementación de la herramienta Demokritos. Durante este proceso, se emplearon técnicas avanzadas como el *fine-tuning* y el *Retrieval-Augmented Generation* (RAG) para optimizar la capacidad de los *Large Language Models* (LLMs) en la generación de código shader.

El primer paso en el proyecto fue el desarrollo de la plataforma web de recopilación de corpus, diseñada para recolectar ejemplos de prompts generados por usuarios a los que se les solicitaba que describieran shaders obtenidos de la web. Este corpus de descripciones y código fue utilizado para alimentar el modelo GPT-3.5 en etapas posteriores, y así especializar el modelo en la generación de shaders. Durante esta fase inicial, se centró la atención en acumular suficientes datos para un eventual ajuste fino (*fine-tuning*).

Mientras se utilizaba la plataforma de recolección de datos, se realizó una prueba de concepto enfocada en la interacción entre modelos GPT-3.5 ajustados mediante *fine-tuning* para sostener conversaciones coherentes sobre artistas uruguayos. Esta prueba sirvió como una primera aproximación al *fine-tuning* antes de aplicarlo al corpus de shaders. El éxito de este experimento permitió adquirir experiencia práctica en el proceso de ajuste fino y sentó las bases para su posterior aplicación en la generación de código shader.

Finalmente, en la fase de desarrollo de Demokritos, se implementó una plataforma capaz de generar shaders a partir de descripciones en lenguaje natural. Aunque en las primeras fases se experimentó con el *fine-tuning* del modelo GPT-3.5 utilizando un corpus reducido de prompts y código shader, los resultados no fueron lo suficientemente robustos debido al tamaño limitado del corpus. A pesar de ello, este enfoque permitió generar algunos shaders más complejos, aunque con una mayor tasa de errores de compilación. Posteriormente, se optó por utilizar RAG para mejorar la precisión del código generado, permitiendo al modelo acceder a ejemplos relevantes y ajustados al contexto proporcionado en cada prompt.

A través de la recuperación de ejemplos previos almacenados en una base de datos vectorial, RAG proporcionó al modelo acceso a información contextual relevante, lo que aumentó la precisión de los resultados. *FAISS* [40] y *Sentence Transformers* [41] fueron componentes clave en la implementación de RAG. FAISS permitió realizar búsquedas de similitud entre los vectores semánticos de los prompts, mientras que *Sentence Transformers* convirtió los prompts en vectores que se almacenaban y recuperaban según su relevancia. Esto ayudó a guiar la generación de código shader de manera más ajustada a los requerimientos del usuario.

3.2 Prueba de Concepto: “Cómo Enseñarle Las Imágenes a una Artista Muerta”

Como parte de este proyecto, se llevó a cabo una prueba de concepto en colaboración con el artista uruguayo Fernando Foglino[15]. Inspirada en la icónica obra de Joseph Beuys, *"Wie man dem toten Hasen die Bilder erklärt"* (*Cómo explicarle las imágenes a una liebre muerta*), la instalación artística recontextualizó este concepto en un entorno digital, donde dos inteligencias artificiales, basadas en modelos GPT, simulaban una conversación sobre artistas uruguayos seleccionados por Foglino [42].

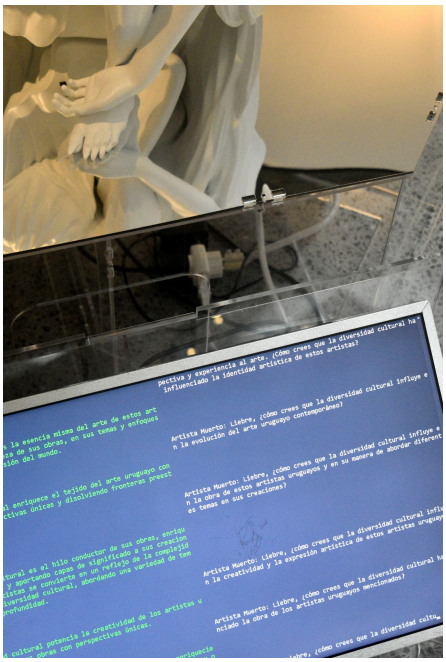
Desarrollo del Concepto de la Obra
(By Fernando Foglino)

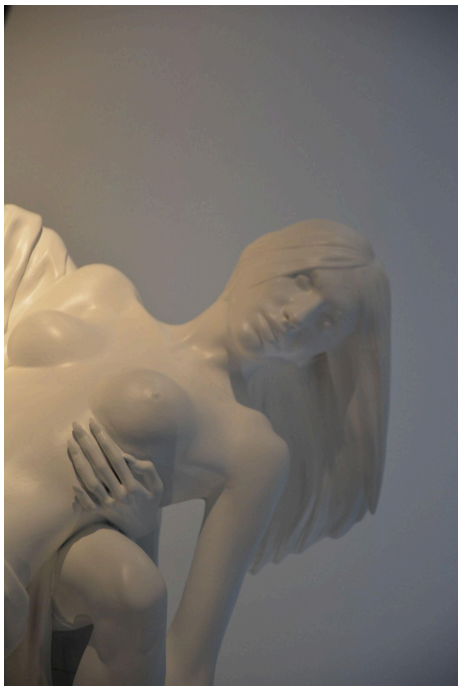
“La obra propone una ficción en dónde la liebre muerta de Joseph Beuys cobra vida en el plano virtual a través de los datos que el artista le susurró durante la performance “Wie man dem toten Hasen die Bilder erklärt” de 1965. La inteligencia artificial se basa en datos y los datos son un reflejo de la historia. El pasado vive en los algoritmos. Ya no hay nadie vivo en el plano real. La inteligencia artificial es la que habla y el artista se hace a un lado para escucharla. Dos máquinas hablan entre sí, infinitamente. La liebre le explica al artista muerto sobre su obra. ¿Qué dicen de nosotros? ¿Bajo qué categoría los han etiquetado?. Las imágenes nos miran, nos juzgan, nos educan o interpelan. Nuestra propia imagen nos ha traicionado.” [43]

Figura 4 - Instalación de obra “Cómo Enseñarle Las Imágenes a una Artista Muerta” en el Centro Cultural España

[15] Fernando Foglino es un artista visual uruguayo cuya obra explora la relación entre la tecnología, el arte y la sociedad. Con un enfoque interdisciplinario, Foglino utiliza medios como la escultura, la instalación y el arte digital para cuestionar las formas en que interactuamos con el mundo digital y las implicaciones de la tecnología en la cultura contemporánea.

Imágenes de la instalación en el Centro Cultural España [44]





Figuras 5,6,7 y 8 - Instalación de obra "Cómo Enseñarle Las Imágenes a una Artista Muerta" en el Centro Cultural España

El objetivo principal de esta prueba de concepto fue doble: en primer lugar, explorar la capacidad de los modelos GPT-3.5 ajustados mediante *fine-tuning* para sostener conversaciones especializadas y coherentes. En segundo lugar, se buscaba adquirir experiencia práctica en el uso del *fine-tuning* como preparación para su posterior aplicación en la generación automática de shaders en este proyecto.

Desarrollo Técnico de la Aplicación

Para lograr la interacción entre los dos modelos GPT, se desarrolló una aplicación en *Python* 3.8 [45] que se comunicaba con la API de *OpenAI* [46]. La estética elegida fue retro, emulando una consola de 8 bits, dividida en dos paneles verticales donde se mostraba la conversación: un panel para el modelo ajustado y otro para el modelo sin ajuste. Esta interfaz gráfica minimalista reforzaba el carácter experimental de la obra, evocando las primeras interfaces computacionales.

Algunas de las bibliotecas utilizadas:

- **openai**: utilizada para interactuar con la API de OpenAI, enviando prompts y recibiendo respuestas de los dos modelos.
- **keyboard**: para gestionar la visualización en pantalla completa.
- **windows-curses** [47]: para dividir la ventana en dos paneles y simular un chat en el que se diferenciaban ambos interlocutores mediante el uso de diferentes colores.

El modelo ajustado, denominado "Liebre", tenía conocimientos sobre los artistas uruguayos, extraídos de un archivo de *fine-tuning* generado a partir de información proporcionada por Foglino. Este archivo fue creado manualmente[16] consolidando datos sobre biografías, obras y estilos de los artistas. Por otro lado, el segundo modelo, representando al "cuerpo de la artista", usaba GPT-3.5 sin ajuste, pero con directrices predefinidas sobre cómo debía interactuar. La conversación se centraba en el modelo base intentando aprender sobre los artistas a través del diálogo con el modelo ajustado. Aunque el modelo base no aprendía realmente durante la interacción, se simulaba una dinámica de enseñanza y aprendizaje entre ambos modelos, creando una experiencia interactiva y artística.

[16] Archivo adjunto en la sección de Enlaces Relevantes

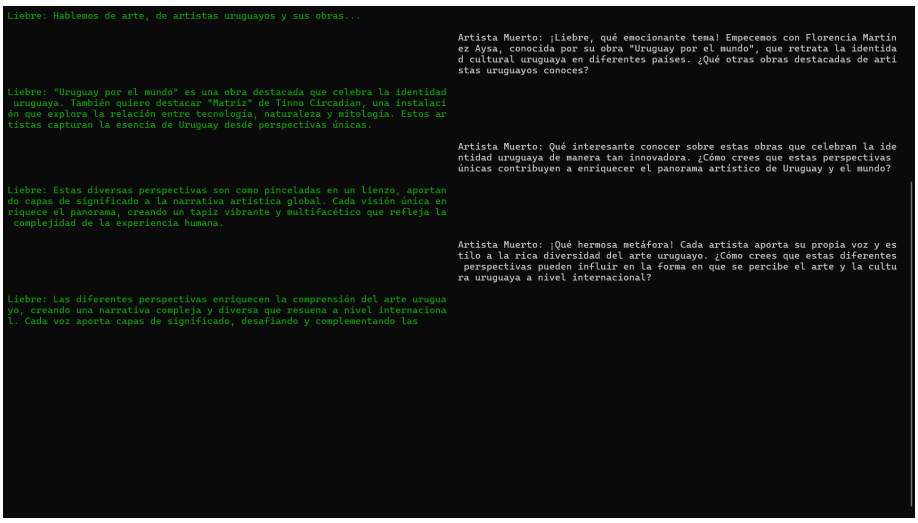


Figura 9 -

“Cómo enseñarle las imágenes a una artista muerta”

Captura de la aplicación que genera diálogo entre dos AI en ejecución.

Objetivos y Resultados

El propósito de la instalación fue evaluar si el modelo ajustado podía proporcionar información coherente y detallada sobre los artistas uruguayos seleccionados. Además, se utilizó como primera aproximación para familiarizarse con el proceso de ajuste fino, técnica que luego sería crucial en la fase de generación de código shader para este proyecto. Esta experiencia no solo permitió entender mejor los desafíos del *fine-tuning*, sino que también ofreció un espacio para experimentar con las capacidades creativas y conversacionales de los modelos GPT en un contexto artístico.

3.3 Aplicación Web para la Recolección de Corpus

Como parte del proyecto, se desarrolló una aplicación web dedicada a la recolección de prompts que describen shaders, con el fin de construir un corpus específico que sirviera para entrenar el modelo de lenguaje (LLM) mediante *fine-tuning*. El objetivo de esta aplicación era facilitar la recopilación de ejemplos reales y sintéticos que representaran una amplia variedad de descripciones y características de shaders, asegurando que el LLM tuviera suficiente contexto para generar código shader coherente y relevante.

Figuras 10 y 11 - Imágenes de la aplicación web para recolectar corpus de shaders y sus respectivos prompts.

3.3.1 Objetivo

El principal objetivo del sitio web fue el de crear una plataforma accesible donde los usuarios pudieran contribuir con descripciones de shaders obtenidos de la web, principalmente desde la plataforma *Shadertoy*[17]. Estos prompts generados por los usuarios servirían como ejemplos para alimentar el proceso de *fine-tuning* del modelo, enriqueciendo su capacidad de generar shaders basados en lenguaje natural.

Algunos de los objetivos específicos del sitio web fueron:

1. **Recolección de Prompts de Calidad**: Facilitar a los usuarios la tarea de escribir y compartir descripciones detalladas de shaders que pudieran ser útiles para entrenar el LLM.
2. **Diversidad de Ejemplos**: Asegurar una variedad de descripciones que cubrieran diferentes aspectos de la generación de shaders, como iluminación, texturas, efectos visuales, etc.
3. **Alimentación Continua del LLM**: Proporcionar un flujo constante de ejemplos que no solo enriquecieran el corpus inicial, sino que también pudieran ser utilizados posteriormente en el proceso de generación de shaders mediante técnicas como RAG.

3.3.2 Funcionalidad del Sitio Web

El sitio web fue diseñado con una interfaz sencilla e intuitiva, que permitiera a los usuarios describir shaders obtenidos de diversas fuentes en la web. Cada entrada consistía en un campo de texto donde los usuarios podían describir el shader, además de metadatos incluidos en la información obtenida desde la plataforma *Shadertoy*, como nombre del creador, título y descripción.

Una vez recopilados, estos prompts se almacenaban en una base de datos diseñada para soportar el procesamiento posterior. Este corpus se utilizó inicialmente para realizar *fine-tuning* al modelo GPT-3.5, con el propósito de mejorar su capacidad de generar código shader. Posteriormente, estos mismos ejemplos fueron integrados en la técnica de *Retrieval-Augmented Generation* (RAG), lo que permitió que el LLM pudiera buscar ejemplos relevantes y utilizarlos como contexto durante el proceso de generación.

3.3.3 Importancia del Corpus para el Proyecto

Una funcionalidad clave de este proyecto es la capacidad de expandir el corpus de shaders a medida que la aplicación es utilizada. Tanto la plataforma desarrollada para recolectar prompts de shaders como la aplicación diseñada para generar código shader permiten esta expansión dinámica. Cada vez que se genera un nuevo shader a partir de un prompt, ese código se

[17] Shadertoy es una plataforma en línea que permite a los desarrolladores y artistas crear, compartir y experimentar con shaders en tiempo real, principalmente utilizando GLSL. Ofrece un entorno interactivo donde los shaders se pueden visualizar directamente en el navegador, lo que facilita la exploración de efectos visuales avanzados.

guarda junto con el prompt, lo que permite enriquecer el conjunto de ejemplos disponibles de manera continua.

La creciente adopción de herramientas de inteligencia artificial generativa para la creación de código, como *GitHub Copilot*, ha provocado una disminución de la actividad en plataformas como *Stack Overflow* [48]. Esto subraya la importancia de contar con corpus de calidad que permitan a los modelos de lenguaje generar código de manera precisa y eficiente, reduciendo la dependencia de las fuentes tradicionales de conocimiento colaborativo. En este sentido, la capacidad de la aplicación para expandir continuamente su propio corpus permite la generación de ejemplos cada vez más relevantes y alineados con las necesidades de los usuarios.

Es importante destacar que, si bien se inició con un corpus relativamente pequeño de aproximadamente doscientos casos recolectados por los usuarios a través de la plataforma web, la parte más significativa del corpus que fue utilizado posteriormente, fue generada de manera sintética. Estos prompts sintéticos se generaron a partir de códigos shader obtenidos mediante *scraping*[18] de la web, que fueron subidos a *GitHub* por un tercero y dejados disponibles para su uso. Se utilizó un modelo GPT-4o para crear descripciones sintéticas de lo que hacía cada código shader, empleando directrices específicas para esta tarea. Este proceso permitió incrementar la base de datos inicial con ejemplos adicionales.

El propósito central es, por tanto, expandir el corpus de manera continua con nuevos ejemplos obtenidos tanto de los usuarios como del uso cotidiano de la herramienta. Si bien esto no garantiza una mejora directa en la calidad o relevancia del código generado, la intención es seguir enriqueciendo el conjunto de datos para que refleje un espectro más amplio de casos de uso y descripciones de shaders.

3.3.4 Contribución del Sitio Web al *Fine-Tuning* y RAG

Los ejemplos recolectados en el sitio web fueron fundamentales para el *fine-tuning* inicial del LLM. El modelo fue ajustado utilizando este corpus, lo que le permitió familiarizarse con las características específicas de los shaders y mejorar su capacidad de traducir descripciones en código funcional.

Más adelante, el corpus también jugó un papel central en la implementación de la técnica de RAG. Mediante el uso de FAISS y *Sentence Transformers*, el LLM podía recuperar ejemplos relevantes del corpus al momento de generar nuevos shaders, mejorando así la precisión y relevancia del código producido.

[18] Scraping es una técnica utilizada para extraer datos de sitios web de manera automatizada. Un software o script se utiliza para navegar por las páginas y extraer información relevante del contenido HTML, lo que permite recoger grandes cantidades de datos de forma eficiente.

3.3.5 Desarrollo Técnico de la Aplicación

El proceso de creación de la aplicación web destinada a la recolección de corpus siguió una serie de pasos técnicos específicos que permitieron alcanzar el objetivo deseado: recolectar descripciones de shaders aportadas por usuarios y vincularlas a un sistema que permita su uso en procesos posteriores de generación de código shader.

1. Elección del Framework: Django

Se seleccionó el *Django Framework* [49] debido a su simplicidad y facilidad de uso, lo que lo convierte en una opción ideal para el desarrollo web. *Django* es un framework de alto nivel escrito en Python [50] que promueve el desarrollo rápido con un diseño limpio, que ofrece características integradas como ORM[19] (Mapeo Objeto-Relacional), autenticación, y administración de base de datos, facilitando la creación de aplicaciones escalables y seguras. Durante esta fase inicial se exploraron tanto la estructura de archivos y patrones de diseño de *Django* como el uso de *custom commands*, que permiten automatizar tareas específicas dentro del proyecto. Estos comandos, definidos dentro de la estructura de *Django*, se utilizan para realizar funciones recurrentes o complejas de manera programática, como procesamiento de datos en lote, actualizaciones automáticas de la base de datos, o generación de informes. También se interiorizó el uso de *fixtures*, archivos que permiten precargar la base de datos con datos de prueba o iniciales, mejorando así el proceso de desarrollo y pruebas.

2. Interacción con Shadertoy y Visualización Web

Una parte clave del proyecto fue investigar cómo interactuar con la plataforma *Shadertoy* [51] y visualizar sus shaders dentro de la aplicación. Se exploraron varios métodos para acceder a los shaders disponibles en *Shadertoy* y cargar esos shaders en la interfaz de usuario. Esta fase implicó la investigación de herramientas para la visualización de shaders utilizando WebGL[20] [52], asegurando una experiencia fluida y funcional para los usuarios al interactuar con los shaders directamente desde la aplicación web.

3. Desarrollo de la Aplicación con Django

Con los conocimientos adquiridos sobre *Django*, se procedió al desarrollo de la aplicación. Esta fase incluyó la creación de los modelos de datos, vistas, y plantillas necesarias para implementar la funcionalidad principal, centrada en la recolección de descripciones de shaders y su almacenamiento en una base de datos para su posterior uso en el sistema. Los usuarios

[19] *ORM* (Object-Relational Mapping) es una técnica que permite interactuar con bases de datos relacionales mediante objetos en el código, sin necesidad de escribir consultas SQL directamente. Esto facilita el desarrollo al mapear las tablas de la base de datos a clases en el lenguaje de programación, permitiendo trabajar con datos de manera más intuitiva y abstracta.

[20] WebGL (Web Graphics Library) es una API basada en OpenGL que permite renderizar gráficos 2D y 3D en navegadores web sin la necesidad de plugins adicionales. Utiliza el poder de la GPU para ejecutar código gráfico, como los shaders, directamente en el navegador, facilitando la creación de aplicaciones gráficas interactivas.

podían aportar descripciones a través de un formulario web, que luego se almacenaban de manera estructurada.

4. Carga Inicial de Shaders y Armado de Fixtures

Se realizó una carga inicial de shaders a partir de datos disponibles públicamente, y se armaron *fixtures* para poblar la base de datos con ejemplos reales. Esto sirvió como un punto de partida para que los usuarios comenzaran a aportar sus propias descripciones de shaders, incrementando de manera progresiva el corpus disponible.

5. *Hosting* en *Elastic Cloud*[21]

Una vez completado el desarrollo de la aplicación, se procedió a realizar el hosting en un servidor en la nube utilizando ***Elastic Cloud*** [53]. Se configuraron los recursos necesarios y se desplegó la aplicación, permitiendo que estuviera accesible públicamente en internet para los usuarios finales. Esto incluyó la configuración de bases de datos, servidores y el despliegue del código en un entorno de producción.

6. *Fine-Tuning* con los Resultados del Corpus

El siguiente paso en el desarrollo del proyecto fue aplicar *fine-tuning* al modelo utilizando los datos recopilados por la aplicación. A medida que los usuarios aportaban descripciones de shaders, estos datos fueron utilizados para ajustar y refinar el modelo de lenguaje, permitiendo que se adaptara mejor a las necesidades del dominio específico de los shaders gráficos.

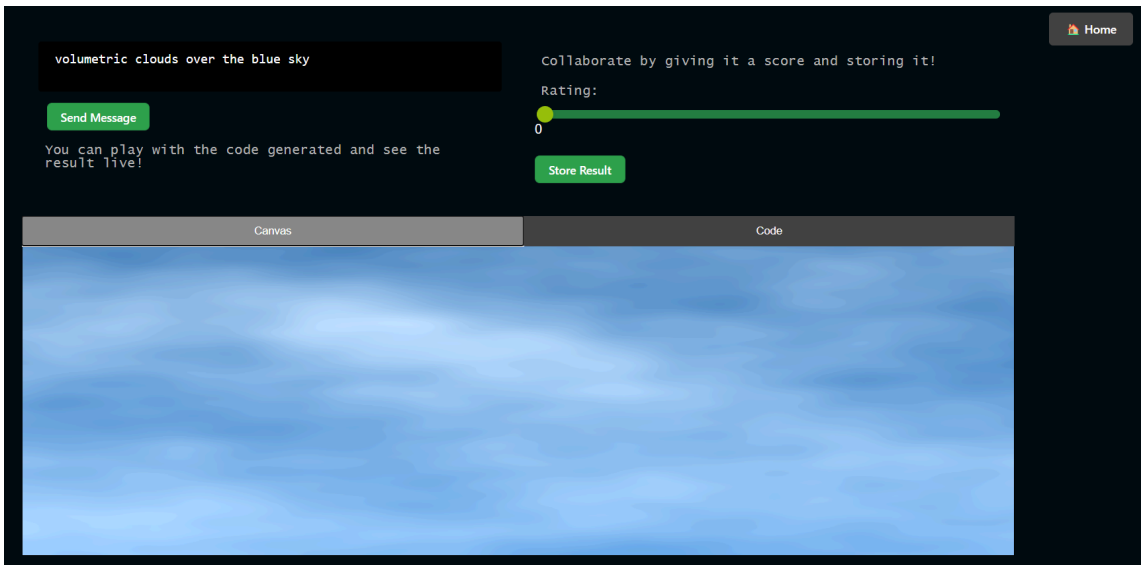
7. Desarrollo de un Intérprete de *Shadertoy* en *WebGL*

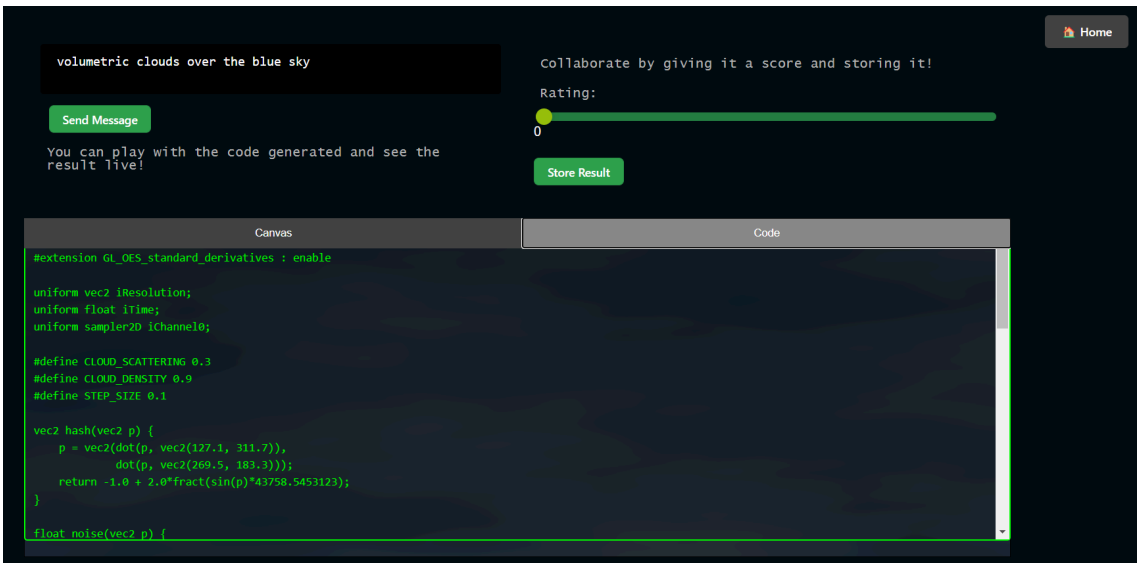
Finalmente, se creó un prototipo de intérprete de *Shadertoy* en *WebGL* con el fin de evaluar visualmente los resultados generados por el modelo GPT-3.5 refinado mediante *fine-tuning*. Este prototipo no fue diseñado para el uso de usuarios externos, sino que fue utilizado exclusivamente como herramienta de evaluación interna. A medida que el proyecto avanzaba, y se estudiaba la posibilidad de incluir RAG y una base de datos vectorial, este prototipo sirvió de base a lo que luego sería la aplicación Demokritos.

[21] *Elastic Cloud* es una plataforma de alojamiento en la nube que permite a los usuarios desplegar y gestionar instancias de *Elasticsearch, Kibana* y otros componentes del *Elastic Stack*. Ofrece soluciones escalables para búsqueda, almacenamiento y análisis de datos sin necesidad de gestionar la infraestructura subyacente, facilitando la implementación y operación en entornos distribuidos.https://www.elastic.co/cloud

3.4 Demokritos - Una Aplicación Web Colaborativa para Generar Shaders a Partir de Lenguaje Natural

La plataforma Demokritos fue desarrollada con el objetivo de facilitar la generación automática de código shader a partir de descripciones en lenguaje natural. Utilizando técnicas avanzadas como *fine-tuning* y *Retrieval-Augmented Generation (RAG)*, Demokritos permite que usuarios, sin necesidad de experiencia en programación gráfica, puedan generar ejemplos funcionales de shaders basados en sus propias descripciones o prompts. Esta plataforma integra el corpus recopilado previamente a través del sitio web de recolección de prompts y utiliza los datos procesados para mejorar la precisión y coherencia del código shader generado.





Figuras 12 y 13 - Imágenes de la aplicación Demokritos.

3.4.1 Motivación y Objetivo de la Plataforma Demokritos

Demokritos fue creada para estudiar la posibilidad de generar código con un componente visual fundamental a partir de lenguaje natural. A lo largo del desarrollo, se exploraron diferentes enfoques para lograr este objetivo, y finalmente se decidió investigar cómo los modelos de lenguaje podían interpretar un prompt en lenguaje natural (por ejemplo, "crear un shader que simule el agua con reflejos de luz") y devolver el código shader correspondiente. La plataforma tiene como objetivo experimentar con la capacidad de los *Large Language Models* (LLMs) para generar código visualmente significativo.

3.4.2 Uso de RAG

Durante la generación del código, Demokritos emplea RAG para buscar ejemplos relevantes en el corpus previamente almacenado. Para ello, se utiliza FAISS (*Facebook AI Similarity Search*) como motor de búsqueda de base de datos vectorial, lo que permite realizar búsquedas de similitud eficiente entre los vectores generados por *Sentence Transformers* a partir de los prompts. Este enfoque asegura que el código generado esté en sintonía con las descripciones y ejemplos previamente recogidos, mejorando la precisión y coherencia de los resultados.

FAISS (*Facebook AI Similarity Search*)

FAISS es una biblioteca de código abierto desarrollada por *Facebook* AI *Research*, diseñada para la búsqueda eficiente de similitud entre vectores en bases de datos de gran escala. Es ampliamente utilizada en sistemas de recuperación de información y para búsquedas rápidas de alta dimensionalidad.

Se encarga de realizar búsquedas de similitud entre los vectores semánticos que representan los prompts. Estos vectores se generan con *Sentence Transformers* y se almacenan en la base de datos del proyecto. Al recibir un nuevo prompt, FAISS compara este nuevo vector con los vectores almacenados y recupera los más similares. Esto facilita la búsqueda de ejemplos relevantes que guían la generación del código shader.

Sentence Transformers

Sentence Transformers es un framework basado en BERT [27] y otras arquitecturas, diseñado para convertir oraciones o párrafos en representaciones vectoriales. Estas representaciones permiten comparar la similitud semántica entre oraciones de manera eficiente, lo que es útil en tareas de búsqueda y recuperación de información.

Los vectores que representan los prompts se generan utilizando el modelo *all-MiniLM-L6-v2* [54] de *Sentence Transformers*, un modelo que convierte el texto en representaciones vectoriales semánticamente significativas. Cada prompt, ya sea uno generado por los usuarios en la plataforma web, creado sintéticamente para ampliar el corpus, o generado en tiempo real en la aplicación de generación de shaders, se transforma en un vector. Estos vectores se almacenan en la base de datos del proyecto y, al recibir un nuevo prompt, FAISS busca y

recupera los vectores más similares, proporcionando ejemplos relevantes que mejoran la precisión del código generado al proveer contexto relevante.

3.4.3 Funcionalidad y Flujo de Trabajo

El flujo de trabajo de la plataforma Demokritos sigue los siguientes pasos:

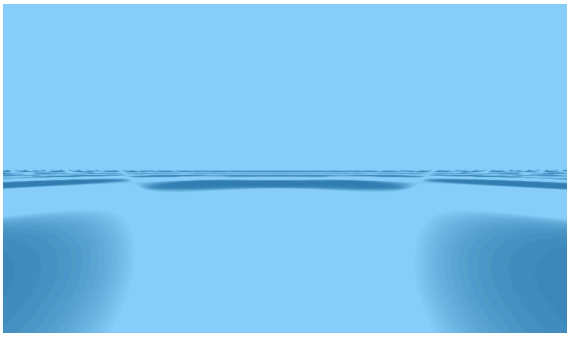
Entrada del Prompt: El usuario proporciona una descripción en lenguaje natural del shader que desea generar.

Recuperación de Contexto mediante RAG: A través de RAG, la plataforma busca en su base de datos vectorial ejemplos relevantes de shaders previamente descritos que puedan servir como contexto. Estos ejemplos ayudan al modelo a generar código más preciso y alineado con las expectativas del usuario.

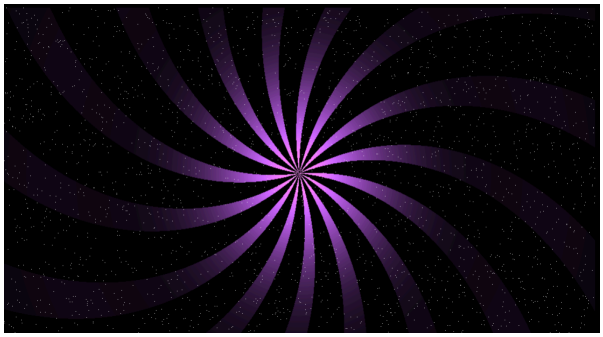
Generación del Código Shader: Una vez que se ha recuperado el contexto relevante, el modelo LLM genera el código shader, utilizando como base los ejemplos recuperados y el prompt original.

Visualización del Resultado: La plataforma proporciona un canvas dónde el usuario podrá ver los efectos gráficos del código de shader resultante directamente en la interfaz. También pone a disposición el código que genera el resultado que se está visualizando y lo permite editar, compilando el resultado en tiempo real.

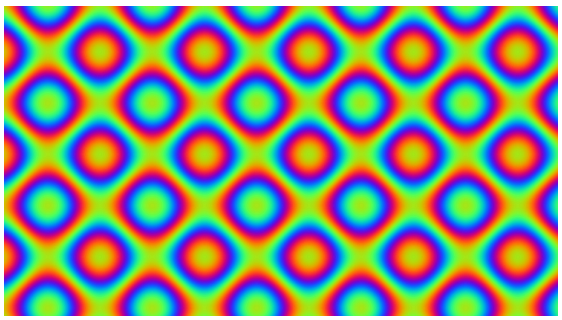
3.4.4 Resultados y Uso Práctico



Prompt: “a reflecting surface, reflecting the sky and the sun”



Prompt: “Pulsating galaxy”



Prompt: “Crippling depression”

Prompt: “LSD”

Figuras 14, 15, 16 y 17 - Imágenes de shaders obtenidos desde la plataforma Demokritos

Demokritos ha demostrado ser una herramienta práctica para la generación rápida de shaders. La combinación de *vector database* para prompts y RAG permite que el sistema entregue código shader funcional y coherente con las expectativas del usuario, basado en las descripciones que proporcionan. Los usuarios pueden experimentar con diferentes descripciones y obtener una variedad de resultados visuales, lo que facilita tanto la exploración creativa como el desarrollo técnico.

El enfoque de Demokritos reduce la barrera técnica para la creación de shaders y proporciona una plataforma accesible tanto para desarrolladores como para artistas, quienes pueden aprovechar la tecnología de los modelos de lenguaje para crear gráficos complejos sin necesidad de profundos conocimientos en programación gráfica.

3.4.5 Desarrollo Técnico de la Aplicación

La aplicación Demokritos fue desarrollada utilizando una combinación de tecnologías para gestionar tanto el backend como el frontend, con el objetivo de generar shaders a partir de descripciones en lenguaje natural. A continuación se detallan los principales componentes técnicos utilizados:

1. Django como Backend y Frontend

Así como con la aplicación de recolección de corpus, se utilizó *Django* como *framework* tanto para el *backend* como para el *frontend* de la aplicación en html y *javascript* [55], aprovechando su sistema de plantillas (*templates*). El desarrollo de Demokritos se basó en el prototipo previamente creado para interpretar el código generado por el modelo ajustado con los ejemplos obtenidos del sitio web de recolección de corpus. Este prototipo fue ampliado y mejorado dentro del mismo proyecto, permitiendo que ambas aplicaciones convivieran y generaran un ecosistema integrado. En este ecosistema, la recolección de corpus y la generación/visualización de shaders funcionan en conjunto, maximizando la eficiencia y coherencia del flujo de trabajo en un único sistema *backend*.

2. SQLite como Base de Datos

Para almacenar los datos de los usuarios y la información generada por el sistema, se utilizó *SQLite* [56] como base de datos. Esta herramienta fue suficiente para manejar el volumen de datos relacionados con los prompts, códigos shader generados y la información asociada a las búsquedas de ejemplos, dada la naturaleza del proyecto.

3. Manejo de Bases de Datos Vectoriales con FAISS

El sistema incorpora la librería FAISS para gestionar una base de datos vectorial. FAISS se utilizó para realizar búsquedas en un conjunto de vectores que representan los prompts de shaders ya generados, ayudando a mejorar las respuestas del modelo al sugerir ejemplos similares basados en descripciones previas.

4. Interacción con la API de OpenAI (GPT-4o)

La aplicación interactúa con la API de *OpenAI*, utilizando el modelo GPT-4o, para generar shaders a partir de los prompts ingresados por los usuarios. GPT-4o interpreta las descripciones en lenguaje natural y genera código GLSL basado en dichas descripciones. La API es esencial para la conversión de los prompts en código shader.

5. Corpus de Shaders y Generación de Prompts

Se utilizó un corpus de aproximadamente veinte mil shaders obtenidos de la plataforma *GLSLSandbox*[22] mediante *scraping*. Para estos shaders, se le solicitó a GPT-4o generar prompts descriptivos que explicaran lo que renderizaba cada shader. Posteriormente, se tomó una muestra de estos prompts y shaders para verificar que las descripciones coincidieran con los resultados visuales, utilizando plataformas de visualización de shaders online como el propio *GLSLSandbox*.

6. Sentence Transformer para la Generación de Vectores

Para la creación de los vectores semánticos necesarios para las búsquedas, se utilizó la librería *Sentence Transformer*, en particular el modelo *all-MiniLM-L6-v2*. Los vectores se generaron sobre los prompts, lo que permite realizar búsquedas semánticas de manera más precisa cuando los usuarios ingresan nuevas descripciones.

7. Desafíos Técnicos y Compilación de Shaders

Durante el desarrollo de Demokritos, surgieron algunos desafíos en la compilación de shaders debido a la mezcla de shaders provenientes de diferentes plataformas. Esto provocó errores como redefiniciones de variables uniformes (`iResolution`, `iTime`) y funciones (`main`). Estos problemas se debieron a diferencias en cómo los shaders fueron escritos en las distintas plataformas, lo que requirió algunos ajustes para evitar conflictos durante la compilación. Por

[22] https://glslsandbox.com/

ejemplo, *Shadertoy* maneja variables uniformes nativas, de forma implícita y ya integradas a la plataforma para que el usuario no tenga que definirlas cada vez, pero si son utilizadas estas tienen que estar definidas para que el código compile en GLSL. Así también como la función `main`, la cual no es utilizada por *Shadertoy*.

También, se detectó en muchas ocasiones que los errores de compilación se debían a incompatibilidad en los tipos de variables a la hora de asignar valores, o comparar con otras variables, por lo que se intentó mitigar utilizando expresiones regulares [57].

Entonces, para intentar solucionar la mayoría de estos errores se implementó una función de preprocesamiento del código generado, que se encarga de:

Eliminación y re-declaración de precisión:

- Elimina cualquier declaración de precisión (`precision lowp|mediump|highp float;`) que ya esté en el código y añade una declaración de precisión `highp` al inicio del código para asegurar consistencia.

Inserción de uniformes específicos de Shadertoy:

- Si el código generado no incluye uniformes específicos de *Shadertoy* (`iResolution`, `iTime`, `iMouse`), el preprocesador los agrega automáticamente. Estos uniformes permiten que el shader reciba información externa, como la resolución de pantalla, el tiempo o la posición del ratón.

Inserción de la función principal:

- Si el código no contiene una función `main`, se crea una estructura envolvente que llama a `mainImage`, permitiendo la integración del shader con el flujo de trabajo de Shadertoy.

Ajuste de índices en bucles:

- Para evitar problemas de tipos de datos en bucles, utiliza una expresión regular (regex) para transformar los bucles `for` que utilizan enteros en su índice. Esta transformación se asegura de que los valores en los bucles sean consistentes con la declaración de tipo en GLSL.

Corrección de asignaciones de vectores:

- En el código GLSL, los vectores de distintos tamaños (e.g., `vec2`, `vec3`) no pueden asignarse entre sí directamente. Para evitar estos errores, el preprocesador detecta, mediante regex, las asignaciones de vectores y ajusta el tamaño de los componentes de modo que coincidan.

- Si un vector de mayor dimensión intenta asignarse a uno de menor dimensión, los componentes adicionales se eliminan. Si ocurre lo contrario, se añaden valores `0.0` por defecto para completar el vector.

Para finalizar, es importante destacar que esta función de preprocesamiento ha sido desarrollada de manera incremental. A medida que se identificaban errores comunes en la generación de código, relacionados principalmente con problemas de sintaxis y no con la lógica subyacente, se fueron añadiendo estas soluciones o "parches". Estos ajustes permiten solventar errores recurrentes, facilitando el proceso de generación y validación del código, logrando una mayor cantidad de resultados exitosos.

3.5 Divulgación y Presentaciones Académicas

Como parte del proceso de validación y difusión del proyecto, se presentó un artículo académico titulado *Demokritos: Interactive, Community-Centred, Self-Improving Shader Generation using Large Language Models* en el *International Symposium on Electronic Art* (ISEA2025), uno de los eventos más reconocidos en la intersección entre arte, tecnología y sociedad.

El comité organizador de ISEA2025 destacó la innovación del proyecto en su aplicación de modelos de lenguaje a la generación de código shader y aceptó su presentación en la categoría Póster. Además, el artículo será publicado en las Memorias del Simposio (*Proceedings*), lo que permitirá que el trabajo tenga mayor visibilidad dentro de la comunidad académica y artística.

La participación en eventos como ISEA2025 no solo permite la difusión del trabajo, sino que también abre la posibilidad de recibir retroalimentación de especialistas en inteligencia artificial, arte generativo y programación gráfica. Asimismo, refuerza el carácter interdisciplinario de Demokritos, integrando el aprendizaje automático, la programación gráfica y la colaboración comunitaria en una plataforma de código abierto.

El artículo enviado puede consultarse en el enlace a Google Drive en el Anexo número 6.

4. Resultados y Evaluación

Se evalúan los resultados de la aplicación hasta el mes de Septiembre (inclusive) del año 2024.

4.1 Evaluación del Rendimiento del Sistema

La evaluación del rendimiento de la plataforma Demokritos se centró en analizar la capacidad del sistema para generar shaders funcionales y coherentes con los prompts en lenguaje natural proporcionados por los usuarios. Este análisis implicó medir la precisión, coherencia y relevancia del código shader generado con respecto a las descripciones iniciales, así como la calidad visual del resultado final.

4.1.1 Criterios de Evaluación

Para evaluar el desempeño del sistema en la generación de shaders a partir de prompts en lenguaje natural, se establecieron varios criterios clave que permitieron medir tanto la funcionalidad técnica como la satisfacción del usuario:

1. **Tasa de Compilación Exitosa**: Se registró el porcentaje de shaders generados que compilaban correctamente frente a aquellos que presentaban errores de compilación. Este criterio es fundamental para determinar la capacidad del modelo de producir código sintácticamente correcto y ejecutable. Una alta tasa de compilación exitosa indica que el sistema es competente en la generación de código shader funcional.
2. **Valoración del Usuario**: Después de generar y visualizar el shader, el usuario tenía la opción de asignar una puntuación al resultado en una escala del 0 al 10. Esta valoración subjetiva refleja la satisfacción del usuario con respecto al shader generado, considerando aspectos como la correspondencia con el prompt inicial, la calidad visual y la utilidad del shader. Puntuaciones más altas indican una mayor satisfacción y percepción de calidad por parte del usuario.
3. **Eficiencia en el Procesamiento**: Se evaluaron los tiempos de demora en la generación y compilación de los shaders, así como el número de intentos fallidos antes de obtener un resultado funcional. Este criterio considera la eficiencia del sistema en términos de velocidad y optimización de recursos. Las mejoras implementadas en el preprocesamiento del código generado antes de intentar compilar contribuyeron a reducir los tiempos y aumentar la tasa de éxito en las compilaciones.
4. **Mejoras en el Preprocesamiento del Código**: Se analizaron las modificaciones realizadas en el preprocesamiento del código shader generado, orientadas a corregir o ajustar el código antes de la compilación. Esto incluyó la detección y corrección automática de errores comunes, la normalización del código y la adaptación a las especificaciones del entorno de ejecución. Evaluar el impacto de estas mejoras permitió optimizar el flujo de trabajo y aumentar la eficacia del sistema.

Estos criterios permitieron una evaluación integral del sistema, combinando medidas objetivas de desempeño técnico con apreciaciones subjetivas de los usuarios. Al centrarse en aspectos

clave como la funcionalidad del código, la satisfacción del usuario y la eficiencia operativa, se pudo identificar áreas de fortaleza y oportunidades de mejora en el proceso de generación de shaders a partir de lenguaje natural.

4.1.2 Resultados de la Evaluación

1. **Tasa de Compilación Exitosa**

   El rendimiento del modelo en la generación de shaders se evaluó en función de la tasa de compilación exitosa de los códigos generados. Los resultados obtenidos muestran lo siguiente:

Códigos generados compilados satisfactoriamente	156
Códigos generados compilados con error	64
Tasa de compilación exitosa	70.9%

   Esto indica que, si bien la tasa de éxito en la compilación es moderada, sigue siendo prometedora dado que muchos shaders generados por el modelo logran compilar correctamente. Es importante resaltar que este resultado puede variar según la complejidad del prompt ingresado, ya que algunos prompts presentan mayores desafíos para el modelo a la hora de generar código shader funcional.

   Para comprender mejor el desempeño del modelo, se analizaron cinco *prompts* representativos ingresados por los usuarios. Cada uno de estos prompts fue evaluado en función de si los shaders generados lograban compilar correctamente. Los resultados son los siguientes:

Prompt	Compilación Exitosa	Compilación con Error	Tasa de Compilación Exitosa
light beams passing through clouds in the sky	4	4	50%
fire	16	3	84.21%
fire and smoke	9	17	34.62%
Generate a GLSL shader that creates a realistic metallic surface with anisotropic reflections and dynamic specular highlights, using environment mapping for light sources.	1	14	6.67%
A forest	6	1	85.71%

Los resultados obtenidos de la tasa de compilación por prompt ofrecen varias conclusiones relevantes sobre el comportamiento del modelo en la generación de shaders:

1. **Complejidad del Prompt y Compilación Exitosa**: Los prompts más simples y generales, como *"fire"* y *"A forest"*, presentaron tasas de compilación significativamente más altas (84.21% y 85.71%, respectivamente). Esto sugiere que el modelo maneja mejor los prompts con menos requisitos técnicos y complejidad visual, lo que facilita la generación de código que compila correctamente. En estos casos, la simplicidad del objetivo reduce las probabilidades de errores en el código generado.
2. **Mayor Complejidad, Mayor Riesgo de Error**: Por el contrario, los prompts más complejos, como *"Generate a GLSL shader that creates a realistic metallic surface with anisotropic reflections and dynamic specular highlights, using environment mapping for light sources"*, obtuvieron una tasa de compilación exitosa muy baja (6.67%). Esto refleja que, a medida que los prompts incluyen especificaciones técnicas más detalladas y avanzadas, el modelo encuentra más dificultades para generar código que compile correctamente, debido a la mayor probabilidad de cometer errores relacionados con la complejidad del *shader* solicitado.
3. **Prompts Intermedios con Resultados Variables**: Un prompt intermedio en complejidad, como *"fire and smoke"*, mostró una tasa de compilación baja (34.62%), lo que sugiere que cuando los prompts incluyen múltiples elementos técnicos, pero no

están claramente definidos, el modelo tiene más dificultades para generar código efectivo. La combinación de varios efectos en un solo prompt, sin especificar cómo deben interactuar, parece aumentar las posibilidades de errores de compilación.

4. **Balance entre Simplicidad y Complejidad**: Un ejemplo interesante es el prompt *"light beams passing through clouds in the sky"*, que tuvo una tasa de compilación del 50%. Este caso sugiere que un equilibrio entre simplicidad y especificidad técnica puede producir resultados moderadamente exitosos, pero no garantiza la estabilidad total en la generación de shaders. Los efectos visuales más abstractos o difíciles de simular sin instrucciones muy precisas parecen resultar en una mayor variabilidad en el éxito de la compilación.

Conclusión General

El análisis muestra que los prompts más simples tienden a generar código que compila con mayor facilidad, mientras que los prompts más complejos y técnicos aumentan la probabilidad de errores. Esto subraya la importancia de encontrar un equilibrio entre la especificidad del prompt y la complejidad visual requerida. A medida que los prompts se vuelven más detallados y avanzados, se requieren técnicas de preprocesamiento y refinamiento adicionales para aumentar la tasa de éxito en la compilación.

2. Valoración del Usuario

Los usuarios valoraron el código shader generado con un puntaje que iba del 0 al 10, según su percepción del resultado visual obtenido. Aunque se trata de un criterio de evaluación subjetivo, esta metodología se considera una de las maneras más fieles para aproximarse a la calidad visual del shader, ya que refleja de forma directa la satisfacción del usuario con el resultado.

Dado que la apreciación estética y la relevancia visual son altamente personales y varían según el contexto de uso, esta evaluación nos proporciona una visión complementaria a las métricas técnicas, como la tasa de compilación. El análisis de estas valoraciones no solo nos permite entender el éxito técnico del modelo, sino también su capacidad para generar shaders que cumplan con las expectativas creativas y visuales de los usuarios.

Resultados obtenidos:

Rating	0	1	2	3	4	5	6	7	8	9	10
Cantidad	5	6	4	3	1	53	19	18	25	5	11

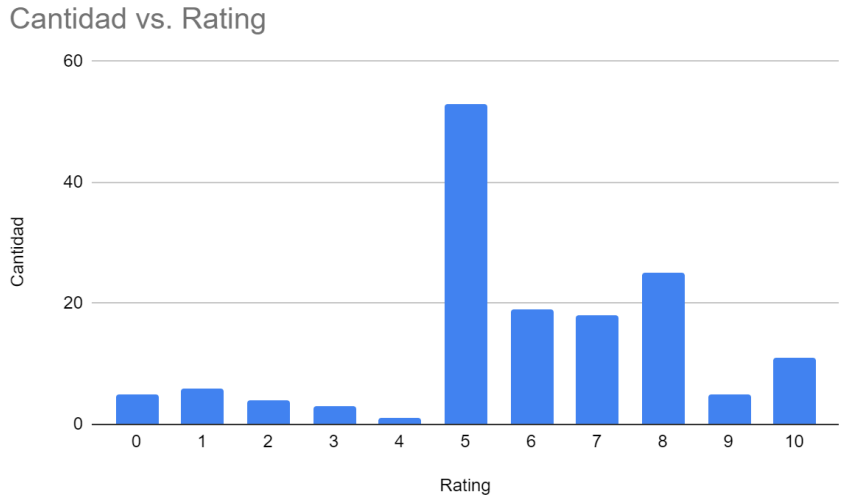


Figura 18 -

Gráfico que ilustra la cantidad de resultados guardados según la calificación ingresada por el usuario.

El resultado subjetivo del código generado, basado en las valoraciones de los usuarios, muestra una tendencia favorable hacia las valoraciones intermedias y altas. La mayoría de los usuarios calificaron los shaders generados con puntajes de **5** (53 casos), lo que indica que el código generado cumple con las expectativas de funcionalidad y calidad visual de manera aceptable, aunque sin destacar significativamente.

Es notable que también haya una considerable cantidad de valoraciones altas entre **8** y **10** (41 casos en total), lo que sugiere que, en algunos casos, los shaders generados fueron considerados altamente satisfactorios y visualmente atractivos. Sin embargo, las valoraciones más bajas, entre **0** y **4** (15 casos), indican que hay margen de mejora, ya que algunos usuarios encontraron que el código no cumplía con sus expectativas o fallaba en generar los efectos deseados.

En general, la mayoría de los resultados parecen estar en un nivel aceptable o superior, con más valoraciones positivas que negativas, lo que sugiere que el sistema es capaz de generar código shader que satisface a la mayoría de los usuarios en términos de funcionalidad y calidad visual.

Un dato relevante a tener en cuenta es que, aunque existen códigos generados que compilaron satisfactoriamente, los usuarios decidieron no guardarlos. Esto sugiere que, a pesar de que el código fue funcional desde un punto de vista técnico, no cumplió con las expectativas visuales y/o creativas del usuario en una proporción significativa de los casos, o por algún otro motivo decidió no guardar el resultado.

Este resultado indica que la mera compilación exitosa no es suficiente para garantizar la aceptación o el uso de un shader generado. Los usuarios parecen tener criterios adicionales, como la estética o la relevancia visual del *shader*, que no siempre se reflejan en una

compilación exitosa. Esto subraya la importancia de no solo enfocarse en la tasa de compilación, sino también en la calidad visual y el ajuste al prompt solicitado.

Este comportamiento resalta la necesidad de continuar optimizando tanto los prompts como la lógica interna del sistema de generación para que los resultados no solo sean técnicamente correctos, sino también atractivos y útiles para los usuarios finales.

3. Eficiencia en el Procesamiento

El tiempo promedio de generación de código shader fue de aproximadamente 5 a 7 segundos por prompt, lo que representa un tiempo de respuesta adecuado para una aplicación web, garantizando una experiencia fluida sin largos tiempos de espera para el usuario. En casos donde el código generado no compila, la plataforma realiza hasta dos intentos adicionales, lo que puede extender el tiempo total a un máximo de 15 a 20 segundos. Si tras estos intentos no se logra una compilación exitosa, el usuario recibe una notificación para modificar el prompt o volver a intentarlo.

Tabla de registro de tiempos de compilación para el prompt “fire and smoke”.

fire and smoke	1	2	3	4	5	6	7	8	9	10
intentos	1	2	3	1	2	1	4	1	2	3
segundos	5.92	22.19	15.61	6.47	9.04	5.62	17.89	7.23	15.72	19.59

Incluso en los casos en los que excede los tres intentos de compilación, y por ende el modelo no fue capaz de mostrar un resultado, el tiempo se mantiene por debajo de treinta segundos, lo que entendemos son tiempos correctos para este tipo de aplicaciones.

4. Mejoras en el Preprocesamiento del Código

A medida que avanzaba el desarrollo de la aplicación web, se fueron identificando patrones de errores recurrentes en la generación de shaders. Esto llevó a implementar una función de preprocesamiento del código antes de su compilación, ajustando automáticamente ciertas condiciones que frecuentemente causaban fallos. Además, se mejoraron las directrices dadas al modelo, optimizando las instrucciones para generar código que cumpliera con las condiciones necesarias para una compilación exitosa. Este proceso iterativo de ajuste continuo permitió mitigar los errores de compilación más comunes, mejorando significativamente los resultados.

Comparación antes y después de incluir pre procesamiento en el código de shader previo a la compilación

	shaders	compilación	tasa de compilación
Antes de implementar pre procesamiento	162	Con Error	8.47%
	15	OK	
Después de implementar pre procesamiento	84	Con Error	66.8
	169	OK	

Si bien los resultados mejoraron significativamente, todavía hay margen para optimizar el preprocesamiento y las directrices dadas al modelo. Se podrían desarrollar nuevas heurísticas para identificar de manera más eficiente errores potenciales en el código antes de su compilación. Además, se podría explorar la posibilidad de incluir técnicas más avanzadas de análisis estático para prever fallos más complejos.

Otro posible enfoque sería mejorar la capacidad del sistema para ajustar automáticamente los shaders basándose en el historial de errores y en las características del prompt proporcionado. Este ajuste automático podría continuar aumentando la tasa de compilación y generar resultados más robustos sin intervención manual.

4.1.3 Análisis de los Resultados

Los resultados obtenidos durante la implementación y pruebas de la plataforma Demokritos son alentadores y demuestran que la combinación de *Large Language Models* (LLMs), *Retrieval-Augmented Generation* y técnicas de preprocesamiento puede ofrecer una solución viable para la generación automática de shaders. A continuación, se destacan los aspectos clave de este análisis:

1. **Tasa de Compilación Exitosa**
   La plataforma logró una tasa de compilación exitosa considerablemente alta. La mayoría de los shaders generados a partir de prompts ingresados por los usuarios fueron compilados sin errores, lo que indica que el sistema es capaz de generar código funcional en la mayoría de los casos. Esta tasa de éxito refleja la capacidad del modelo para traducir instrucciones en lenguaje natural en código que cumple con los requisitos de compilación.

2. **Mejoras en el Preprocesamiento del Código**
   Uno de los mayores avances fue la introducción de una función de preprocesamiento que corregía automáticamente errores de sintaxis o estructura antes de que el código shader intentara ser compilado. Esta mejora, implementada tras la identificación de patrones de fallos recurrentes, incrementó significativamente la tasa de compilación exitosa y optimizó la eficiencia del proceso de generación de shaders.

3. **Eficiencia en los Tiempos de Generación**
   La velocidad de generación del código shader fue evaluada y mostró resultados prometedores. Con un tiempo promedio de 5 a 7 segundos por prompt (en casos exitosos), la plataforma demostró ser lo suficientemente rápida como para mantener una experiencia fluida para los usuarios. Incluso en escenarios más complejos, los tiempos de respuesta fueron aceptables. Cuando los shaders no compilaron correctamente en los intentos iniciales, el sistema implementó reintentos automáticos con el mismo prompt antes de notificar al usuario, lo que ayudó a mejorar la tasa general de éxito.

4. **Evaluación del Usuario**
   Otro aspecto importante fue la evaluación subjetiva de los shaders generados por parte de los usuarios. El sistema permite a los usuarios valorar el código generado con una puntuación que varía entre 0 y 10. Los resultados muestran que, en general, los shaders obtenidos fueron bien recibidos, con una mayoría de valoraciones positivas. Esto sugiere que el sistema no solo genera código funcional, sino que también ofrece resultados visualmente atractivos y alineados con las expectativas de los usuarios.

5. **Impacto de las Directrices Claras en la Generación**
   Los prompts con descripciones detalladas y específicas generaron shaders más ricos visualmente y complejos en términos de código. Sin embargo, esta complejidad también incrementó las probabilidades de errores de compilación, ya que el código generado tiende a ser más sofisticado y propenso a fallos. En contraste, los prompts más simples, aunque producían resultados menos atractivos, presentaban una mayor tasa de éxito en la compilación debido a que el código generado era más básico y directo.

   Este hallazgo sugiere que el sistema es altamente sensible al nivel de complejidad y detalle del input. Mientras que los usuarios experimentados, capaces de proporcionar prompts más específicos, pueden obtener resultados visuales más interesantes, también corren un mayor riesgo de fallos de compilación. En consecuencia, se podría mejorar la interfaz del usuario para ofrecer orientación sobre cómo equilibrar la complejidad y la claridad en los prompts, quizás mediante plantillas o ejemplos que ayuden a optimizar la relación entre calidad visual y tasa de compilación exitosa.

La plataforma Demokritos ha demostrado ser eficaz en la generación automática de shaders, gracias a la implementación de preprocesamiento de código, la optimización de tiempos de generación, y la capacidad del modelo GPT-4o para interpretar instrucciones en lenguaje natural. Las evaluaciones positivas por parte de los usuarios y la alta tasa de compilación reflejan que el sistema ofrece resultados prácticos y satisfactorios. No obstante, futuras mejoras en la interfaz de usuario y el refinamiento de las directrices para la generación de código podrían aumentar aún más la calidad de los shaders producidos y mejorar la experiencia global del usuario.

4.2 Comparación de Resultados entre Diferentes Modelos

Para evaluar la capacidad de distintos modelos de lenguaje en la generación de shaders a partir de prompts en lenguaje natural, se implementó un comando en *Django* que replicaba el proceso de generación, preprocesamiento y compilación de código GLSL utilizado en la plataforma Demokritos. Por cada prompt ingresado, el sistema generaba el código utilizando el modelo de lenguaje correspondiente, aplicaba la misma función de preprocesamiento empleada en Demokritos, e intentaba compilar el código hasta lograr una compilación exitosa o agotar los intentos predeterminados.

El proceso de evaluación abarcó varios modelos de lenguaje: GPT-3.5, GPT-3.5 *fine-tuned*, GPT-4o, Claude.ai, y GPT-4o con *Retrieval-Augmented Generation* (Demokritos). A cada modelo se le proporcionaron las mismas indicaciones y condiciones de compilación para garantizar una comparación justa. Los resultados obtenidos fueron los siguientes:

Modelo	Compilaciones Fallidas	Compilaciones Exitosas	Tasa de Compilación Exitosa (%)
Claude.ai	246	104	29.7%
GPT-3.5	286	101	26.1%
GPT-3.5 fine-tuned	472	16	3.3%
GPT-4o	125	150	54.5%
Demokritos	64	192	75.0%

Análisis de los Resultados

- **Claude.ai** y **GPT-3.5** demostraron tasas similares de compilación exitosa, con una proporción de alrededor de 1 éxito por cada 2 o 3 intentos fallidos. Estos resultados reflejan una competencia comparable en la generación de código gráfico, aunque la tasa de errores sigue siendo relativamente alta.
- **GPT-3.5 fine-tuned**, a pesar de estar ajustado específicamente para la generación de shaders utilizando un corpus especializado, presentó el peor rendimiento, con una alta proporción de compilaciones fallidas. Esto puede atribuirse al tamaño insuficiente del corpus utilizado para el ajuste fino, lo que limitó su capacidad para generalizar correctamente en los prompts proporcionados.
- **GPT-4o** destacó por su rendimiento superior, logrando una tasa de compilación exitosa notablemente mayor que *Claude.ai* y GPT-3.5, con más compilaciones exitosas que fallidas. Esto indica que GPT-4o es más capaz de generar código GLSL funcional desde el inicio, lo que lo convierte en una opción preferida para tareas más complejas.
- **Demokritos** presentó los mejores resultados, con una tasa de compilación exitosa mucho más alta que cualquier otro modelo. El uso de RAG parece haber mejorado significativamente la coherencia y precisión del código generado, al permitir que el modelo acceda a ejemplos relevantes del corpus durante la generación. Este enfoque ha demostrado ser más efectivo para reducir los errores y mejorar los resultados finales.

Conclusión

La comparación entre los diferentes modelos revela que GPT-4o, especialmente cuando se utiliza con técnicas de *Retrieval-Augmented Generation* (RAG) como en Demokritos, ofrece un rendimiento superior en la generación de shaders funcionales. GPT-4o logra una mayor tasa de compilación exitosa en comparación con los modelos más antiguos como GPT-3.5 y Claude.ai, lo que indica su capacidad para manejar tareas complejas de manera más eficiente.

Si bien GPT-3.5 y Claude.ai mostraron una tasa de compilación exitosa aceptable, requieren más intentos para generar código funcional, lo que los hace menos eficientes para este tipo de tareas. Por otro lado, el rendimiento del modelo GPT-3.5 fine-tuned fue decepcionante, probablemente debido a un corpus insuficiente durante el ajuste fino, lo que sugiere que este enfoque puede no ser efectivo sin una base de datos más robusta.

Demokritos, que combina GPT-4o con RAG, ha demostrado ser la opción más eficiente y precisa, alcanzando una tasa de compilación exitosa significativamente superior. Esto confirma que la incorporación de RAG, al proporcionar contexto adicional y relevante, mejora la coherencia del código generado y reduce los errores. De cara al futuro, estos resultados sugieren que la combinación de GPT-4o con RAG es el enfoque más prometedor para optimizar la generación automática de shaders.

4.3 Comparación con Otras Técnicas de Generación de Código

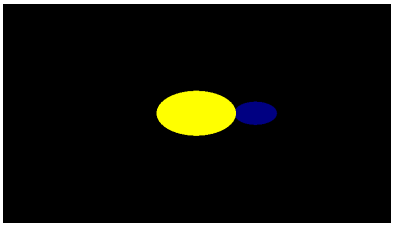
La generación automática de código ha evolucionado considerablemente, dando lugar a una variedad de enfoques que intentan automatizar la creación de programas. En esta sección, se comparan las técnicas utilizadas en el proyecto Demokritos, que incluyen *fine-tuning* en GPT-3.5 y la posterior implementación de *Retrieval-Augmented Generation* en GPT-4, con otros métodos de generación de código ampliamente utilizados.

4.3.1 Primer Experimento: Fine-Tuning con GPT-3.5

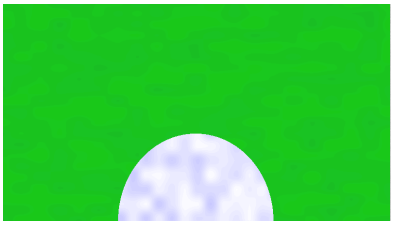
El primer enfoque utilizado en el desarrollo de Demokritos fue realizar fine-tuning sobre el modelo GPT-3.5. Este ajuste fino se llevó a cabo utilizando un corpus específico de shaders que se recopiló previamente, con el objetivo de mejorar la capacidad del modelo para generar shaders a partir de descripciones en lenguaje natural. La hipótesis inicial era que, mediante el fine-tuning, el modelo podría especializarse en la generación de código shader en GLSL y adaptarse mejor a este dominio específico.

Las primeras pruebas con GPT fueron previas a la existencia de la versión “chat” de GPT. Los resultados utilizando el “*playground*” de OpenAI fueron muy simples y no era viable generar un código de shader con atractivos visuales.

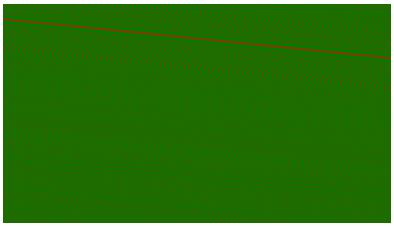
Primeros Resultados



Prompt: write a shadertoy code that renders the earth orbiting the sun



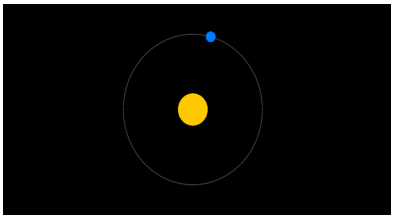
Prompt: write a shadertoy code that renders a football rolling



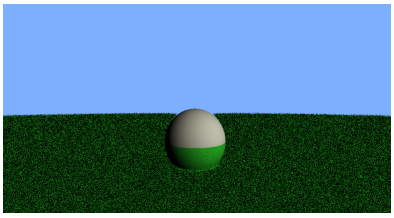
Prompt: write a shadertoy code that renders a forest

Figuras 19, 20 y 21 - Shaders generados desde GPT-3 Playground

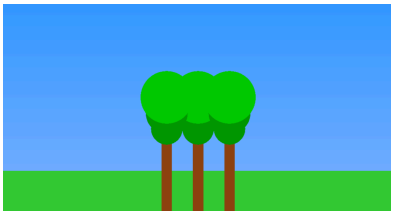
Resultados con prompts similares en Demokritos



Prompt: The earth orbiting the sun



Prompt: A football rolling over a green field



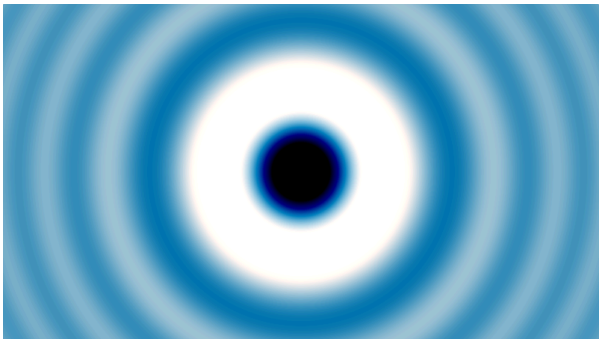
Prompt: A forest

Figuras 22, 23 y 24 - Shaders generados con Demokritos

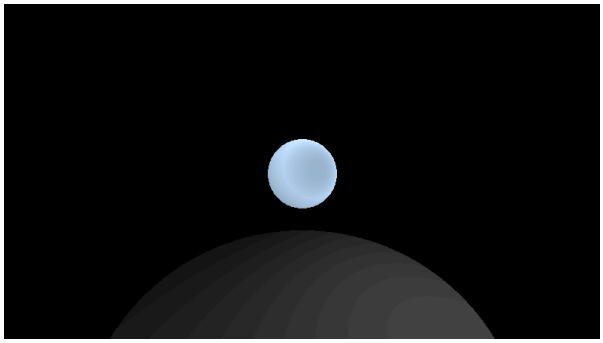
Resultados y Evaluación del Fine-Tuning

A pesar de las expectativas, los resultados obtenidos del modelo ajustado fueron inferiores en cuanto a compilación satisfactoria del código generado. De cuatrocientos ochenta y ocho intentos de generación de alrededor de ciento setenta prompts, solo diez y seis shaders lograron compilar exitosamente, mientras que cuatrocientos setenta y dos resultaron en compilaciones fallidas.

Resultados obtenidos haciendo *fine-tuning* con 250 ejemplos



Prompt: Sky and Clouds



Prompt: A phong sphere

Figuras 25 y 26 - Shaders generados a partir del modelo GPT-3.5 ajustado con 250 ejemplos obtenidos de la aplicación de recolección de corpus

El *fine-tune* realizado con el corpus obtenido a través de la aplicación web no fue lo suficientemente robusto para mejorar significativamente el desempeño del modelo GPT-3.5 en la generación de shaders. Este resultado subraya la importancia de contar con un corpus más amplio y diverso cuando se trata de tareas técnicas específicas. Además, con la llegada de GPT-4, que ofreció mejoras sustanciales en la generación de código sin requerir ajustes

adicionales, se decidió abandonar el enfoque de *fine-tuning* en favor de explorar técnicas como *Retrieval-Augmented Generation*.

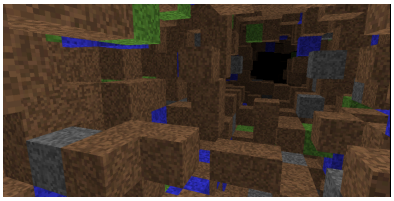
4.3.2 Segundo Experimento: RAG con GPT-4o

Con el lanzamiento de GPT-4o, se decidió experimentar con *Retrieval-Augmented Generation* (RAG) y una base de datos vectorial en lugar de continuar con *fine-tuning*. GPT-4o ofrecía una mejor capacidad para comprender prompts y generar respuestas, pero no permitía *fine-tuning*. En su lugar, RAG se utilizó para recuperar ejemplos de shaders generados previamente que coincidían con prompts similares.

Ventajas: Al utilizar RAG, se podía buscar en una base de datos vectorial (implementada con FAISS) ejemplos de shaders relacionados con el prompt actual. Esto permitió proporcionar contexto relevante para mejorar la precisión del código generado sin necesidad de *fine-tuning*. Los resultados fueron altamente satisfactorios, logrando una mayor coherencia entre el prompt ingresado y el código shader generado.

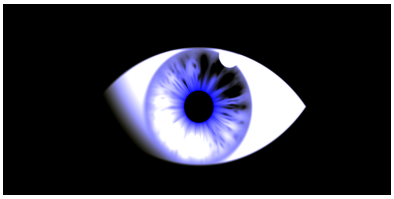
Desventajas: Aunque RAG fue efectivo, la dependencia en la base de datos de ejemplos significa que su desempeño puede variar según la cantidad y calidad de los ejemplos almacenados.

Resultados obtenidos con RAG



Prompt: A voxel scene like minecraft (Ver capítulo 4.5)

Prompt: rain with a rainbow



Prompt: geometrical composition reminiscent of an eye

Figuras 27, 28 y 29 - Shaders generados con Demokritos

4.4 Conclusión Comparativa

En comparación con otras técnicas de generación de código, el enfoque utilizado en Demokritos, tanto con *fine-tuning* en GPT-3.5 como con RAG en GPT-4, ofrece ventajas significativas en la generación de shaders. Mientras que el *fine-tuning* permitió mejorar la precisión del modelo en un dominio específico, la transición a RAG con GPT-4o permitió aprovechar el poder superior del modelo sin necesidad de ajustes adicionales, al mismo tiempo que proporcionaba contexto relevante a través de una base de datos vectorial.

4.5 Aspectos Éticos Identificados durante el Desarrollo

Dado que el sistema genera código gráfico funcional que puede considerarse también como obra visual, resulta pertinente reflexionar brevemente sobre los aspectos éticos que surgen de este tipo de generación asistida por modelos de lenguaje.

La utilización de LLMs en la generación de código, y en particular para la creación de shaders, plantea importantes consideraciones éticas, especialmente en lo que respecta a la propiedad intelectual y los derechos de autor. Este proyecto ha hecho uso de un corpus compuesto por ejemplos de shaders obtenidos de diversas fuentes, lo que abre una discusión sobre las posibles implicaciones legales y éticas en relación con el uso de estos trabajos visuales.

- ➔ **Copyright y Propiedad Intelectual en la Generación de Shaders**

Los shaders, aunque son programas técnicos que se ejecutan en la GPU, generan resultados visuales que pueden tener un carácter artístico. Esta dualidad plantea desafíos en términos de propiedad intelectual, ya que el código puede estar protegido por derechos de autor, al igual que su resultado gráfico.

Durante el desarrollo del proyecto, se utilizaron ejemplos de shaders disponibles en línea para construir el corpus de entrenamiento del sistema. Esto plantea una preocupación relevante: ¿hasta qué punto el código generado por el modelo puede reproducir o estar influenciado por obras preexistentes sin el consentimiento de sus autores? Esta incertidumbre sobre la trazabilidad del entrenamiento es señalada como un riesgo significativo en el artículo *On the Opportunities and Risks of Foundation Models* [58] que alerta sobre la posibilidad de apropiación indebida y la falta de transparencia en la generación de contenido.

- ➔ **Transparencia y Crédito a los Autores Originales**

Otro aspecto ético relevante es la transparencia en el uso de los datos. Los desarrolladores de sistemas basados en LLMs deben ser transparentes acerca de las fuentes de datos utilizadas para entrenar sus modelos (lo mismo que con los datos de sus usuarios [59]). Además, cuando sea posible, se debería reconocer a los creadores originales de shaders o trabajos visuales en los que se basaron los ejemplos de entrenamiento.

El artículo mencionado anteriormente también subraya la necesidad de una mayor transparencia en los modelos fundacionales para garantizar la trazabilidad de los datos de entrenamiento y mitigar los riesgos asociados a la atribución inadecuada de obras generadas por estos sistemas.

- ➔ **Originalidad y Creatividad en la Producción Automática**

Un aspecto crucial en el uso de LLMs para generar código shader es la cuestión de la originalidad. Cuando el modelo genera código basado en ejemplos preexistentes, surge la pregunta de hasta qué punto el trabajo resultante puede considerarse original. En el contexto

del arte visual, los shaders generados podrían verse como trabajos derivados en lugar de completamente nuevos y originales.

Este tema se conecta nuevamente con las preocupaciones éticas planteadas en *On the Opportunities and Risks of Foundation Models*, donde se destaca que las creaciones generadas automáticamente por estos modelos podrían estar en una zona gris entre la originalidad y la derivación de obras existentes [58].

- ➔ **Responsabilidad en el Uso de LLMs**

Finalmente, los desarrolladores y usuarios de LLMs deben ser conscientes de su responsabilidad en el uso ético de estos sistemas. Esto incluye asegurarse de que no se infrinjan los derechos de otros al utilizar ejemplos de código o resultados visuales, así como ser transparentes acerca de cómo se utilizan los datos para entrenar los modelos. De acuerdo con el artículo mencionado, la responsabilidad ética es clave al utilizar modelos fundacionales que pueden afectar diversas áreas de la creación artística y técnica.

5. Conclusiones y Trabajos a Futuro

5.1 Conclusiones

Este proyecto ha explorado la capacidad de los *Large Language Models (LLMs)* para generar código *shader* a partir de descripciones en lenguaje natural, utilizando dos enfoques distintos: *fine-tuning* y *Retrieval-Augmented Generation* (RAG). A través de esta investigación, se han alcanzado varios resultados clave que demuestran el potencial y las limitaciones de estas técnicas en la generación automática de shaders.

5.1.1. Capacidad de los LLMs para Generar Código de Shader

Uno de los resultados más importantes es que los LLMs, cuando están bien ajustados o combinados con técnicas como RAG, pueden generar código *shader* coherente y funcional a partir de descripciones en lenguaje natural. La capacidad de los modelos para entender prompts y traducirlos en código específico para gráficos por computadora representa un avance significativo en la simplificación del desarrollo de gráficos visuales complejos. Esto reduce considerablemente la barrera técnica para aquellos que no tienen experiencia en programación gráfica.

5.1.2. Efectividad del *Fine-Tuning* en GPT-3.5

El proceso de *fine-tuning* del modelo GPT-3.5 con un corpus específico de *shaders* no logró los resultados esperados en cuanto a la precisión del código generado. Si bien el objetivo era adaptar el modelo a los requerimientos técnicos del lenguaje GLSL y mejorar la coherencia con los prompts proporcionados, la cantidad limitada de ejemplos en el corpus utilizado resultó insuficiente. Esto se reflejó en una tasa baja de compilaciones exitosas, lo que sugiere que el modelo no logró una adaptación adecuada al dominio de los shaders. En futuras iteraciones, un corpus más amplio y variado podría mejorar significativamente los resultados de *fine-tuning*.

5.1.3. Transición a GPT-4o y RAG

La transición a GPT-4o, junto con la implementación de RAG, marcó un avance importante en la generación de código shader. A pesar de que GPT-4o no permitía *fine-tuning* (Lo comienza a permitir en Agosto del 2024), su capacidad superior para manejar tareas complejas, junto con el uso de una base de datos vectorial que proporcionaba contexto relevante, resultó en shaders de alta calidad y coherencia. RAG permitió recuperar ejemplos de shaders similares y utilizarlos para mejorar la precisión y relevancia del código generado, eliminando la necesidad de entrenar el modelo específicamente para cada tarea.

5.1.4. Limitaciones Identificadas

A pesar de los resultados positivos, se identificaron algunas limitaciones durante el desarrollo del proyecto. En primer lugar, la calidad de los shaders generados depende en gran medida de la claridad y especificidad del prompt. Descripciones ambiguas o incompletas resultaron en código menos funcional o visualmente menos satisfactorio. Los prompts con mayor complejidad técnica presentan una mayor probabilidad de fallar en la compilación del código, aunque a menudo pueden generar resultados visualmente más atractivos (ver Anexo de Imágenes). Esto implica un compromiso entre la calidad visual del resultado y la probabilidad de una compilación exitosa, dependiendo directamente de la complejidad del prompt.

Además, la implementación de RAG requiere una base de datos de ejemplos suficientemente rica y diversa para garantizar que el sistema pueda recuperar contextos útiles para la generación de shaders.

5.1.5. Implicaciones para el Futuro de la Generación Automática de Shaders

Este proyecto ha demostrado que es posible automatizar la creación de shaders utilizando LLMs y técnicas avanzadas como *fine-tuning* y RAG. Si bien se han obtenido resultados satisfactorios, este campo aún presenta oportunidades para mejorar la precisión y la flexibilidad del código generado. En el futuro, el desarrollo de modelos más especializados o la expansión de bases de datos de ejemplos podrían mejorar aún más los resultados. Además, la integración de nuevas técnicas de generación automática de código podría abrir nuevas posibilidades para automatizar tareas complejas en el desarrollo de gráficos por computadora.

5.1.5. Consideraciones Éticas

Es importante destacar que la utilización de modelos de lenguaje para la generación automática de código gráfico plantea interrogantes éticos vinculados a la autoría y el uso de datos de entrenamiento. Si bien este trabajo no se propuso abordar en profundidad estas cuestiones, se reconoce la necesidad de continuar explorando estos aspectos en investigaciones futuras, especialmente en contextos donde el código generado puede adquirir también un valor artístico.

5.2 Trabajo a Futuro

El proyecto Demokritos ha demostrado la capacidad de utilizar LLMs para generar código shader a partir de lenguaje natural, pero hay varios aspectos que pueden mejorarse y ampliarse en futuras investigaciones y desarrollos.

A continuación, se proponen algunas direcciones para trabajo a futuro que podrían optimizar y diversificar los resultados obtenidos hasta ahora:

Exploración de Otros Modelos de Lenguaje

Una dirección prometedora es evaluar el desempeño de otros modelos de lenguaje de gran escala en la generación de shaders, tales como LLaMA de *Meta AI* o Claude de *Anthropic*. Incluyendo las opciones dentro de la propia aplicación, permitiendo al usuario elegir con qué modelo quiere generar el código. Estos modelos podrían ofrecer diferentes capacidades y enfoques en comparación con GPT-4. Realizar experimentos similares utilizando *fine-tuning* y técnicas de *Retrieval-Augmented Generation* con estos modelos permitiría comparar su eficacia en la generación de código shader a partir de lenguaje natural. Esto ayudaría a determinar si ofrecen ventajas en términos de precisión, coherencia o eficiencia en la generación de código gráfico.

Ampliación y Diversificación del Corpus

El corpus de datos es un factor crítico en el éxito de cualquier modelo de lenguaje. A futuro, se propone ampliar y diversificar el corpus de shaders utilizado en este proyecto. Incorporar shaders provenientes de una variedad de fuentes y plataformas, así como recolectar ejemplos de distintos estilos gráficos, mejoraría la capacidad del modelo para generar código shader más variado y adaptado a diferentes necesidades visuales. Al hacerlo, se espera que el modelo pueda generalizar mejor y responder a una gama más amplia de solicitudes de los usuarios con una tasa de compilación alta. Además, se podría realizar un *fine-tuning* sobre modelos como LLaMA o GPT-3.5, con un corpus significativamente más amplio. Este enfoque permitiría mejorar los resultados obtenidos en la experimentación previa, donde el tamaño del corpus limitado afectó el rendimiento del modelo ajustado. Al aumentar el volumen de ejemplos, se espera una mayor coherencia y precisión en la generación de código shader.

Implementación de Evaluaciones Cuantitativas

El proyecto actualmente cuenta con un sistema que utiliza comandos *Django* para registrar las compilaciones exitosas y fallidas de shaders generados por varios modelos, como GPT-4o (sin RAG), GPT-3.5, GPT-3.5 fine-tuned y Claude.ai. Para mejorar estas evaluaciones cuantitativas, se propone:

- **Integrar nuevos modelos**, como LLaMA, para poder comparar la tasa de compilación exitosa entre diferentes modelos.
- **Evaluar tiempos de generación y compilación**, analizando la eficiencia de los modelos.
- **Mejorar el preprocesamiento del código**, optimizando la corrección automática de errores antes de intentar compilar, reduciendo así la cantidad de intentos fallidos.

Estas métricas proporcionarían una visión más precisa del rendimiento de cada modelo y ayudarían a identificar áreas clave para futuras optimizaciones.

Implementación de Evaluaciones Cualitativas

Además de las evaluaciones cuantitativas, es importante contar con herramientas que permitan realizar una evaluación cualitativa del código shader generado. En este sentido, sería valioso desarrollar un sistema que permita analizar las características del código desde un punto de vista estético y técnico, utilizando heurísticas predefinidas que evalúen diferentes aspectos del shader.

Se podría desarrollar un comando *Django* que evalúe el código generado en base a los siguientes criterios:

- **Cantidad de funciones y complejidad estructural**: Analizar el número de funciones y su complejidad, lo que puede indicar la sofisticación del código generado.
- **Variedad de colores y efectos visuales**: Evaluar la paleta de colores utilizada y la calidad de los efectos visuales producidos, asegurando que los shaders generen una experiencia visual atractiva.
- **Uso eficiente de recursos gráficos**: Implementar heurísticas que valoren si el shader utiliza de manera óptima los recursos de la GPU, evitando sobrecargas innecesarias.
- **Creatividad y novedad**: A través del análisis de patrones comunes en shaders, se podría desarrollar una métrica que evalúe cuán original es el shader comparado con ejemplos previamente generados o almacenados.

Estas evaluaciones cualitativas ofrecerían un panorama más completo del desempeño del modelo, no solo desde una perspectiva técnica, sino también estética y funcional.

Optimización del Proceso de *Fine-Tuning*

Otro aspecto a desarrollar es la optimización del proceso de *fine-tuning*, especialmente en términos de eficiencia y costo computacional. Técnicas como *Low-Rank Adaptation* (LoRA) [60] o *Adapter Layers* [61] podrían permitir ajustes más precisos y menos costosos en los modelos, mejorando su rendimiento en tareas específicas como la generación de shaders sin la necesidad de ajustar todos los parámetros del modelo base. Estas técnicas harían el proceso de *fine-tuning* más accesible y escalable para futuras investigaciones.

Interacción Directa con los Resultados Generados

Una mejora importante para futuras versiones de la plataforma sería permitir a los usuarios una interacción directa con los resultados generados. Actualmente, la plataforma genera shaders estáticos basados en los prompts. A futuro, se podría agregar la posibilidad de subir imágenes que funcionen como un canvas para aplicar el shader generado, permitiendo al usuario ver los efectos visuales de manera interactiva, como en un filtro blur o en otras aplicaciones visuales. Esta funcionalidad haría que la plataforma fuera más útil para artistas y desarrolladores gráficos que busquen experimentar con shaders en tiempo real.

Otra forma de interacción puede ser a través de redes sociales, permitiendo a través de un botón, hacer un *posteo* en *Instagram* o *X* (ex *Twitter*) un fragmento de diez segundos de la reproducción del shader en formato gif por ejemplo, acompañado de un texto predeterminado y del prompt que generó el código.

Control de Estilo en la Generación de Shaders

Otra línea de trabajo futuro sería incorporar mecanismos de control de estilo en los shaders generados. Al permitir que los usuarios especifiquen estilos o características particulares mediante etiquetas o parámetros adicionales en los prompts, se podría generar código shader que esté más alineado con las preferencias visuales del usuario. Esto sería especialmente útil en contextos donde se requieren gráficos visuales con características estilísticas específicas.

Exploración de Nuevas Aplicaciones Visuales

Finalmente, sería interesante expandir el enfoque del proyecto hacia otras áreas de la programación gráfica, tales como la generación de scripts para motores de renderizado 3D, animaciones o efectos visuales en tiempo real. Evaluar cómo los LLMs pueden contribuir en la automatización de estas tareas podría abrir nuevas oportunidades y aplicaciones prácticas para la plataforma Demokritos [62].

Consideraciones éticas para futuros desarrollos

En futuros desarrollos, sería recomendable construir el corpus de entrenamiento exclusivamente a partir de ejemplos que cuenten con licencias abiertas, como *Creative Commons*[23] o *GPL*[24], para evitar posibles conflictos legales relacionados con derechos de autor. Asimismo, mantener un registro claro de las fuentes utilizadas permitiría, en caso de ser necesario, atribuir la autoría correspondiente a los creadores originales cuando los resultados generados sean utilizados públicamente. También debería considerarse que los shaders generados podrían interpretarse como obras derivadas, lo que exigiría una gestión responsable

[23] Un conjunto de licencias que permiten a los creadores otorgar permisos sobre su obra de forma flexible, especificando qué usos están permitidos bajo ciertas condiciones, como la atribución o la prohibición de uso comercial.
[24] GPL (General Public License): Una licencia de software libre que garantiza a los usuarios la libertad de usar, modificar y redistribuir el software, siempre y cuando las modificaciones también se distribuyan bajo la misma licencia.

de los derechos de autor. Por último, se sugiere establecer políticas claras sobre el uso de datos protegidos durante el entrenamiento de modelos, junto con mecanismos preventivos para evitar posibles infracciones. Estas medidas apuntan a fomentar un desarrollo ético y transparente en la aplicación de modelos de lenguaje en contextos creativos.

5.3 Reflexiones Finales sobre la Capacidad de los LLMs en la Generación de Shaders

La capacidad de los Large Language Models (LLMs) para generar código shader ha sido evaluada en este proyecto a través de la implementación de distintos modelos como GPT-3.5, GPT-4, *Claude.ai*, y variantes con *fine-tuning* y *Retrieval-Augmented Generation*. Los resultados obtenidos permiten extraer varias conclusiones relevantes sobre el potencial y las limitaciones actuales de los LLMs en esta tarea específica.

1. Precisión y Tasa de Compilación:

Los modelos más avanzados, como GPT-4o y GPT-4o con RAG, mostraron una tasa de compilación exitosa significativamente superior a otros modelos, alcanzando hasta un 75% con RAG. Este resultado sugiere que la inclusión de técnicas de recuperación de contexto, como RAG, mejora considerablemente la precisión del código generado. A pesar de ello, aún existe un margen de error debido a la complejidad inherente a los shaders y a la naturaleza específica del lenguaje GLSL, donde los detalles técnicos son cruciales para el éxito de la compilación.

2. Limitaciones del *Fine-Tuning*:

El *fine-tuning* aplicado al modelo GPT-3.5 no alcanzó los resultados esperados, con una tasa de compilación exitosa de apenas un 3.3%. Esto se atribuye en parte al tamaño reducido del corpus utilizado para el ajuste fino (alrededor de doscientos cincuenta ejemplos), insuficiente para que el modelo logre una adaptación adecuada a las características específicas de los shaders. Este resultado pone en evidencia que para tareas altamente especializadas como la generación de código shader, es crucial contar con un corpus más amplio y diverso para que el *fine-tuning* sea efectivo.

3. Influencia del Prompt en la Calidad del Código Generado:

Durante el proceso de pruebas, se observó que los prompts más complejos generaban resultados visualmente más ricos y atractivos. Esto se debe a que al incluir detalles técnicos específicos, el modelo tiene mayor capacidad para generar shaders con efectos avanzados, como iluminación realista, simulaciones dinámicas o texturas complejas. Sin embargo, esta mayor complejidad en los prompts también genera código más denso y elaborado, lo que incrementa el tiempo de ejecución y la probabilidad de errores de compilación.

Por otro lado, los prompts simples producen código más básico y menos visualmente atractivo, pero con una clara ventaja: al ser más sencillo, es menos probable que contenga errores de compilación. Este contraste evidencia una relación directa entre la complejidad del prompt y la calidad del resultado visual, pero también subraya la importancia de equilibrar la riqueza del shader con la estabilidad del código generado. En este sentido, es importante que los usuarios ajusten sus expectativas dependiendo de su objetivo final, ya que un prompt más detallado puede llevar a tiempos de desarrollo más largos debido a la mayor tasa de errores.

4. Eficiencia en la Generación:

El tiempo promedio para generar un shader funcional fue inferior a los diez segundos en la mayoría de los casos, lo que es adecuado para una aplicación web interactiva. Este resultado sugiere que los LLMs, especialmente los modelos más recientes como GPT-4, pueden generar código shader en tiempos razonables sin comprometer la experiencia del usuario. A pesar de los tiempos de espera ligeramente superiores cuando ocurren errores de compilación, el sistema ofrece una eficiencia aceptable que puede mejorarse aún más mediante técnicas de preprocesamiento del código.

5. El Rol del Preprocesamiento y la Mejora Continua:

A lo largo del desarrollo del proyecto, se implementaron mejoras en el preprocesamiento del código generado, basadas en los errores de compilación más comunes. Estas mejoras, junto con directrices más claras para el modelo, permitieron mitigar algunos de los problemas recurrentes y aumentar la tasa de compilación exitosa. Esto sugiere que, además de los avances en los modelos de lenguaje, el ajuste y optimización del pipeline de generación es crucial para mejorar los resultados en tareas tan especializadas como la programación gráfica.

Conclusiones Finales

La generación automática de shaders utilizando LLMs ha mostrado ser factible, especialmente cuando se utilizan modelos avanzados como GPT-4o con técnicas de recuperación de contexto como RAG. Sin embargo, los resultados también subrayan la necesidad de contar con corpus más extensos y específicos para optimizar los modelos a través de *fine-tuning*, así como la importancia de un proceso de preprocesamiento robusto que pueda corregir o guiar el código generado antes de la compilación.

A medida que los LLMs continúan evolucionando, es razonable esperar que su capacidad para generar código shader con mayor precisión y eficiencia mejore, lo que abre nuevas oportunidades para integrar estas herramientas en entornos creativos y de desarrollo gráfico.Además, se enfatiza la importancia de generar y compartir corpus de manera pública y accesible, como un recurso clave para impulsar futuros avances en la generación automática de código shader y en el uso de LLMs en general.

6. Referencias

[1] “Github Copilot.” [Online]. Available: https://github.com/features/copilot#features

[2] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, “The Impact of AI on Developer Productivity: Evidence from GitHub Copilot.” [Online]. Available: https://arxiv.org/pdf/2302.06590

[3] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A Survey on Large Language Models for Code Generation.” 2024. [Online]. Available: https://arxiv.org/abs/2406.00515

[4] Fernando Foglino, “Sitio de Fernando Foglino.” [Online]. Available: https://www.foglino.me/

[5] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer graphics: principles and practice (2nd ed.)*. USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[6] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering, Fourth Edition*, 4th ed. USA: A. K. Peters, Ltd., 2018.

[7] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0131387685

[8] NVIDIA, *Mythbusters Demo GPU versus CPU*. [Online Video]. Available: https://www.youtube.com/watch?v=-P28LKWTzrI

[9] “OpenGL.” [Online]. Available: https://www.opengl.org/

[10] Brian Caulfield, “What’s the Difference Between a CPU and a GPU?” [Online]. Available: https://blogs.nvidia.com/blog/whats-the-difference-between-a-cpu-and-a-gpu/

[11] D. KC and C. T. Morrison, “Neural Machine Translation for Code Generation.” 2023. [Online]. Available: https://arxiv.org/abs/2305.13504

[12] N. Jiang, T. Lutellier, and L. Tan, “CURE: Code-Aware Neural Machine Translation for Automatic Program Repair,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, May 2021, pp. 1161–1173. doi: 10.1109/icse43902.2021.00107.

[13] S. Sree Harsha, A. C. Sohoni, and K. Chandrasekaran, “NeuralDoc-Automating Code Translation Using Machine Learning,” in *Artificial Intelligence Driven Circuits and Systems*, B. Mishra, J. Mathew, and P. Patra, Eds., Singapore: Springer Singapore, 2022, pp. 125–138.

[14] T. Mikolov, M. Karafiát, L. Burget, J. Černocký, and S. Khudanpur, “Recurrent neural network based language model,” in *Interspeech 2010*, 2010, pp. 1045–1048. doi: 10.21437/Interspeech.2010-343.

[15] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0321486811

[16] E. Dehaerne, B. Dey, S. Halder, S. Gendt, and W. Meert, “Code Generation Using Machine Learning: A Systematic Review,” *IEEE Access*, vol. 10, pp. 1–1, Jan. 2022, doi: 10.1109/ACCESS.2022.3196347.

[17] X. V. Lin, C. Wang, L. Zettlemoyer, and M. D. Ernst, “NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System.” 2018. [Online]. Available: https://arxiv.org/abs/1802.08979

[18] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai, “A Machine Learning Framework for Programming by Example,” in *Proceedings of the 30th International Conference on Machine Learning*, S. Dasgupta and D. McAllester, Eds., in Proceedings of Machine Learning Research, vol. 28. Atlanta, Georgia, USA: PMLR, Jun. 2013, pp. 187–195. [Online]. Available: https://proceedings.mlr.press/v28/menon13.html

[19] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to

document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, doi: 10.1109/5.726791.
[20] S. Hochreiter and J. Schmidhuber, "Long Short-term Memory," *Neural Comput.*, vol. 9, pp. 1735–80, Dec. 1997, doi: 10.1162/neco.1997.9.8.1735.
[21] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language Models are Unsupervised Multitask Learners," 2018, [Online]. Available: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf
[22] T. B. Brown *et al.*, "Language Models are Few-Shot Learners," pp. 1–75, May 2020.
[23] OpenAI, "GPT-4 Technical Report," Mar. 2023, [Online]. Available: https://arxiv.org/pdf/2303.08774.pdf
[24] A. Vaswani *et al.*, "Attention Is All You Need," Jun. 2017, [Online]. Available: https://arxiv.org/pdf/1706.03762
[25] Y. Wu *et al.*, "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation." 2016. [Online]. Available: https://arxiv.org/abs/1609.08144
[26] "Transformer's Encoder-Decoder Let's Understand The Model Architecture," Kikaben. [Online]. Available: https://kikaben.com/transformers-encoder-decoder/
[27] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." 2019. [Online]. Available: https://arxiv.org/abs/1810.04805
[28] M. Chen *et al.*, "Evaluating Large Language Models Trained on Code." 2021. [Online]. Available: https://arxiv.org/abs/2107.03374
[29] Y. Li *et al.*, "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022, doi: 10.1126/science.abq1158.
[30] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." 2021. [Online]. Available: https://arxiv.org/abs/2005.11401
[31] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space." 2013. [Online]. Available: https://arxiv.org/abs/1301.3781
[32] "Riding the AI Wave with Vector Databases: How they work (and why VCs love them)," LunaBrain. [Online]. Available: https://lunabrain.com/blog/riding-the-ai-wave-with-vector-databases-how-they-work-and-why-vcs-love-them/
[33] A. R. Ilya Sutskever, Tim Salimans, Karthik Narasimhan, "Improving Language Understanding by Generative Pre-Training", [Online]. Available: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf
[34] "Introducing ChatGPT." [Online]. Available: https://openai.com/index/chatgpt/
[35] Keijiro Takahashi, *AIShader*. GitHub Repository. [Online]. Available: https://github.com/keijiro/AIShader
[36] *MaxiShader - GLSL Coding Assistant.* [Online]. Available: https://www.yeschat.ai/gpts-9t557p1IPt6-MaxiShader-GLSL-Coding-Assistant
[37] *GLSL Code Generator*. [Online]. Available: https://jit.dev/glsl-code-generator
[38] John Pertoft, *LLM Shader Toy*. GitHub Repository. [Online]. Available: https://github.com/johnPertoft/llm-shader-toy
[39] Sabrina Verhage, *LLM Shader Art*. [Online]. Available: https://llm-shader-art.sabrina.dev/
[40] M. Douze *et al.*, "The Faiss library," 2024.
[41] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks." 2019. [Online]. Available: https://arxiv.org/abs/1908.10084
[42] Fernando Foglino, "Sitio de Fernando Foglino - Liebre," Liebre. [Online]. Available:

https://www.foglino.me/liebre/
[43] F. Foglino, “COMO ENSEÑARLE LAS IMAGENES A UN ARTISTA MUERTO Posible titulo COMO ENSEÑARLE LAS IMAGENES A UNA ARTISTA MUERTA COMO ENSEÑARLE LAS IMAGENES A UNX ARTISTA MUERTX.” 2024.
[44] “Trepida: escuchando parentescos extraños,” CCE Montevideo. [Online]. Available: https://cce.org.uy/evento/trepida/
[45] “Python 3.8 Documentation.” [Online]. Available: https://docs.python.org/3.8/
[46] “OpenAI API Documentation.” [Online]. Available: https://platform.openai.com/docs/overview
[47] “windows-curses Documentation.” [Online]. Available: https://pypi.org/project/windows-curses/
[48] M. del Rio-Chanona, N. Laurentsyeva, and J. Wachs, “Are Large Language Models a Threat to Digital Public Goods? Evidence from Activity on Stack Overflow.” 2023. [Online]. Available: https://arxiv.org/abs/2307.07367
[50] “Python 3.13 Documentation.” [Online]. Available: https://docs.python.org/3.13/
[51] “Shadertoy Documentation.” [Online]. Available: https://www.shadertoy.com/howto
[52] “WebGL.” [Online]. Available: https://get.webgl.org/
[53] “Elastic Cloud Documentation.” [Online]. Available: https://www.elastic.co/cloud
[54] “all-MiniLM-L6-v2 Documentation.” [Online]. Available: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
[55] “Javascript Documentation.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript
[56] “SQLite Documentation.” [Online]. Available: https://www.sqlite.org/docs.html
[57] “Python 3.13 Documentation, regex.” [Online]. Available: https://docs.python.org/3/howto/regex.html
[58] R. Bommasani *et al.*, “On the Opportunities and Risks of Foundation Models,” Aug. 2021, [Online]. Available: https://arxiv.org/pdf/2108.07258.pdf
[59] A. Priyanshu, Y. Maurya, and Z. Hong, “AI Governance and Accountability: An Analysis of Anthropic’s Claude.” 2024. [Online]. Available: https://arxiv.org/abs/2407.01557
[60] E. J. Hu *et al.*, “LoRA: Low-Rank Adaptation of Large Language Models.” 2021. [Online]. Available: https://arxiv.org/abs/2106.09685
[61] N. Houlsby *et al.*, “Parameter-Efficient Transfer Learning for NLP.” 2019. [Online]. Available: https://arxiv.org/abs/1902.00751
[62] K. Fukaya, D. Daylamani-Zad, and H. Agius, “Intelligent Generation of Graphical Game Assets: A Conceptual Framework and Systematic Review of the State of the Art”, [Online]. Available: https://arxiv.org/abs/2311.10129v1

7. Anexos

7.1 Enlaces Relevantes

1. **Link de la aplicación**

   https://demokritos.xyz/start/

2. **Archivo para el *fine-tuning* del modelo GPT para la obra "Cómo Enseñarle Las Imágenes a una Artista Muerta":**

   https://docs.google.com/spreadsheets/d/1iPdQnGZDtEGqZN7prk1BSzo2uysi3ZEqnbwgj8_0jCo/edit?usp=sharing

3. **Repositorios Demokritos**

   https://github.com/es-eph/demokritos

4. **Shadertoy**

   Link a perfil de shadertoy. Todos los shaders son generados por algún LLM.

   https://www.shadertoy.com/user/sferrando

5. **Documentos de la obra de Fernando Foglino**

   Boceto inicial de la obra.

   https://drive.google.com/file/d/12m3dQG5obmooGied7bgrZKVyp7RiG6ok/view?usp=drive_link

   Documentos de dónde se obtuvo el corpus para entrenar al LLM (Toda la información contenida en los documentos se encuentra disponible en la web).

   https://drive.google.com/drive/folders/1J732ghsbDrlPUd1-YDWXpwirAG6FMoG3?usp=drive_link

   Imágenes de la obra en exposición en el Centro Cultural España.

   https://drive.google.com/drive/folders/12vHZR9PsL2iIqcyEhZekFMUOetZDoB-z?usp=drive_link

6. **Artículo postulante a ISEA 2025**

   Paper Track:
   https://drive.google.com/file/d/14fhaXjrebcBG_JAtIwiRGRqAzRGEQZBP/view?usp=drive_link

   Poster Track:
   https://drive.google.com/file/d/1pdx4iGOW2HwdOWxFpzcRl4I1FbXTVQgD/view?usp=drive_link

7.2 Ejemplo de la Ecuación de Attention

La ecuación de atención tiene la siguiente forma:

$$Attention(Q, V, K) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Donde:

- **Q**: es la matriz de *queries* (consultas), que representa las palabras de entrada transformadas.
- **K**: es la matriz de *keys* (claves), que también son las palabras transformadas.
- **V**: es la matriz de *values* (valores), que contiene la información original de las palabras.
- **dkd_kdk**: es la dimensión del espacio de *keys* y *queries* (normalmente se usa para evitar que los valores crezcan demasiado).
- **Softmax**: es una función que convierte los valores en probabilidades entre 0 y 1, sumando 1.

Ejemplo sencillo:

Supongamos que tenemos una frase muy corta: "*El gato persigue al ratón*". Nuestro objetivo es calcular cómo cada palabra (consulta) debe "atender" a las otras palabras en la secuencia.

Paso 1: Representación de palabras

Primero, las palabras se convierten en vectores numéricos (esto lo hace el modelo automáticamente) llamados *queries* (Q), *keys* (K) y *values* (V). Supongamos que el modelo asigna los siguientes valores simplificados a las palabras:

- "El" → [1,0]
- "gato" → [0,1]
- "persigue" → [1,1]
- "al" → [0,0]
- "ratón" → [0,2]

Paso 2: Cálculo de la similitud (*QK^T*)

Ahora, se calcula la similitud entre cada palabra (consulta) y todas las demás (claves). Esto se hace multiplicando la matriz de *queries* por la transpuesta de la matriz de *keys*. La similitud mide qué tan relevante es cada palabra para otra.

El resultado sería algo así:

	"El"	"gato"	"persigue"	"al"	"ratón"
"El"	1	0	1	0	0
"gato"	0	1	1	0	2
"persigue"	1	1	2	0	2
"al"	0	0	0	0	0
"ratón"	0	2	2	0	4

Paso 3: Aplicar *softmax*

Luego, se aplica la función *softmax* fila por fila para convertir estas similitudes en probabilidades (números entre 0 y 1 que sumen 1). Por ejemplo, para la palabra "persigue", la fila de la matriz podría transformarse a algo como:

"El"	"gato"	"persigue"	"al"	"ratón"
0.15	0.15	0.4	0.05	0.25

Esto significa que "persigue" presta más atención a sí misma (0.4) y al resto de las palabras según sus probabilidades.

Paso 4: Ponderar los *values* (V)

Finalmente, multiplicamos estas probabilidades por los valores (*values*) correspondientes. Si asumimos que los valores (*V*) son los mismos que las representaciones iniciales de las palabras, la atención para "persigue" se calcularía como una suma ponderada de los *values*:

$$Attention(\text{"}persigue\text{"}) = 0.15\,x\,[1,0] + 0.15\,x\,[0,1] + 0.4\,x\,[1,1] + 0.05\,x\,[0,0] + 0.25x[0,2]$$

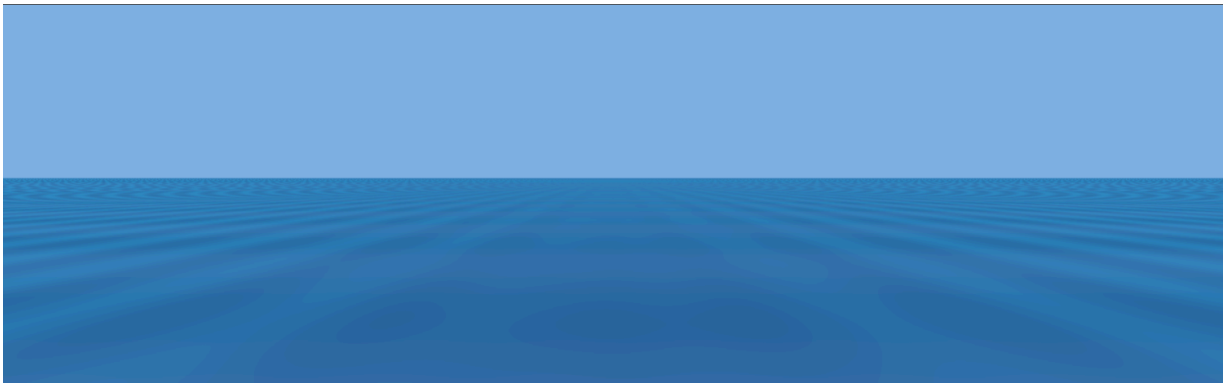
Esto nos da un nuevo vector:

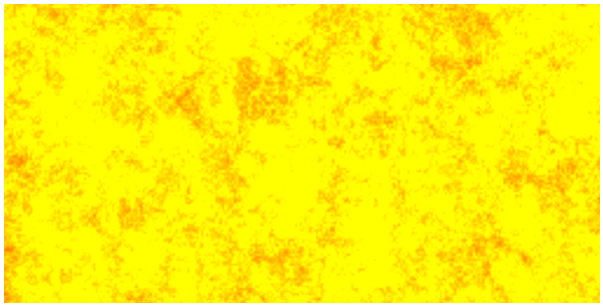
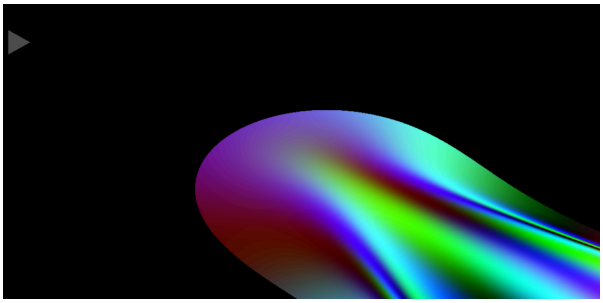
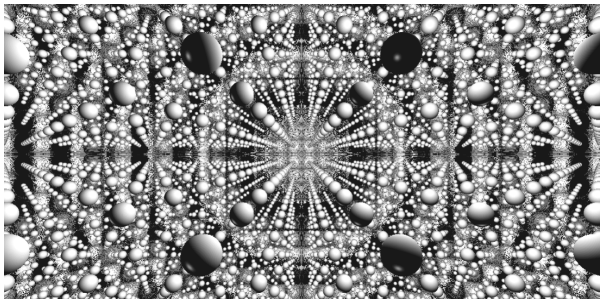
$$Attention(\text{"}persigue\text{"}) = [0.15\,,\,0] + [0\,,\,0.15] + [0.4\,,\,0.4] + [0\,,\,0] + [0\,,\,0.5] = [0.55\,,\,1.05]$$

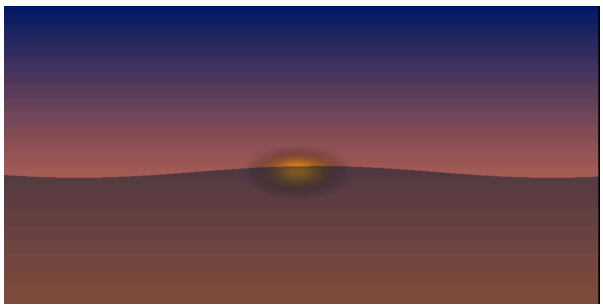
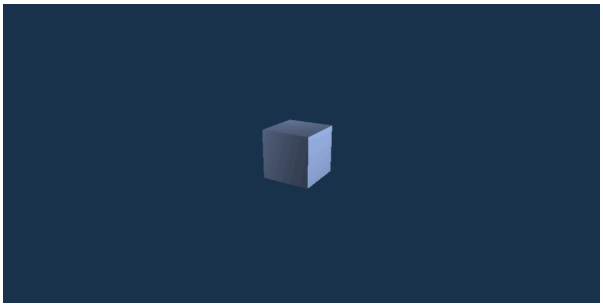
Este vector resultante es el que representa a la palabra "persigue" en el contexto de toda la frase. Así, el modelo ha aprendido qué partes de la oración son importantes para esta palabra y ajusta su representación en consecuencia.

7.3 Anexo de imágenes

Comparación de resultados con prompt complejos y un prompt simple que describa lo mismo.

Create a GLSL shader that simulates dynamic ocean waves with reflection and refraction of light based on Fresnel's equations, adjusting wave height with Perlin noise and animating water movement over time.

Ocean Waves


Prompt Complejo	Prompt Simple
Develop a shader that simulates fire with realistic flame flickering, using noise functions for randomness, gradient coloring for heat intensity, and transparency for smoke.	Fire
	
Generate a GLSL shader that creates a realistic metallic surface with anisotropic reflections and dynamic specular lights, using environment mapping for light sources.	Metal surface
	
An animated sunset scene with soft, gradient transitions for sky colors, dynamic light scattering for clouds, and reflective water.	Sunset scene

	
A rotating 3D cube with realistic shadows and reflections, using Phong shading and normal mapping for added depth.	Rotating 3D cube
	
Volumetric clouds over a blue sky	clouds