Introducción:

Para esta práctica y debido a la descripción del enunciado optare por una implementación en C++, debido a su alto desempeño computacional frente a otros lenguajes.
Evitando largos tiempos de ejecución durante las pruebas, comprobaciones y comparaciones durante el desarrollo.

Diseño del Algoritmo y Explicación del algoritmo principal:

El objetivo principal de la práctica consiste en aproximar una imagen en escala de grises mediante el trazado de hilos rectos entre un conjunto de puntos (clavos) dispuestos en la periferia de la imagen. Dado que el espacio de búsqueda es inmenso, se ha optado por un enfoque basado en una heurística voraz (*greedy*), donde en cada iteración se toma la decisión localmente óptima (el hilo que más reduce el error) sin reconsiderar decisiones pasadas o mejores a más adelante.

Indexación y preparación de las estructuras de datos:

Para garantizar la eficiencia temporal y espacial del algoritmo, un aspecto crítico que computará miles de iteraciones, hemos usado la siguiente estructura de datos.

Gestión de la Imagen (image_loader.h y image_loader.cpp):

Donde la estructura Imagen encapsula un único std::vector<int> pixeles unidimensional, en vez de matrices bidimensionales.

- El acceso a las coordenadas (x, y) bidimensionales se resuelve aritméticamente mediante las funciones integradas at(x, y) que calculan el índice plano como y * ancho + x. Los vectores unidimensionales garantizan la localidad espacial.Esto minimiza seguro los fallos de caché al leer y modificar las intensidades.

- La lectura y escritura se ha restringido al estándar P2 ASCII del formato PGM, permitiendo inicializar rápidamente el objetivo y muestra la imagen en escala de grises.

Geometría y Generación de Clavos (geometry.h):

La función generarClavos distribuye equitativamente el número solicitado de clavos n a lo largo del perímetro de un círculo centrado en la imagen, ya que por lo general como se mencionaba en el enunciado esta es la forma habitual de trabajar en hilorama.

- Para garantizar que los n clavos queden distribuidos de manera perfectamente equidistante a lo largo de la circunferencia, usamos ángulo = 2.0π * i / n y más tarde con sen y cos tendremos su x,y.

- Y puesto que las coordenadas podrían generar múltiples clavos en el mismo píxel exacto. El diseño incluye una comprobación que descarta duplicados, ahorrando así ciclos de cómputo redundantes.

Implementación del algoritmo Voraz:

La lógica central implementa el proceso de optimización iterativo, con la finalidad de trazar los hilos que mejor recreen la imagen original.

El algoritmo mantiene dos representaciones: la imagen original que se desea imitar y un lienzo en blanco que se va oscureciendo iterativamente. Usando una evaluación de Error Cuadrático entre ambas. Al restar los valores de los píxeles del lienzo modificado frente al valor original y elevar al cuadrado la diferencia. Cuanto mayor sea la reducción de este error total al simular un hilo, mejor será el candidato. Ya que penaliza de forma mucho más severa las grandes diferencias de intensidad que una simple diferencia lineal.
Y por parte de la función LineaBresenham será capaz de operar en modo simulación, para calcular cuán prometedor es un hilo o modificar una imagen de forma permanente.

En cada iteración del bucle principal, se selecciona un clavo de origen u. A partir de este origen, el algoritmo recorre todos los posibles clavos. Para cada par (u, v) válido, se ejecuta el cálculo de Bresenham para obtener su puntuación (reducción de error). El criterio voraz consiste en priorizar estrictamente aquellas líneas con la máxima reducción positiva de error.

Donde el algoritmo itera combinando extracción del mejor clavo y dibujo de líneas hasta que el límite absoluto en caso de usarse una máxima cantidad de hilos estipulada realista. O si se alcanzan MAX_FALLOS (fijado en 5), el algoritmo en 5 iteraciones consecutivas no ha encontrado ni un solo hilo que mejore la convergencia.

Mejoras y Heurísticas Implementadas

Para optimizar tanto la calidad visual de la imagen generada como el tiempo de ejecución del algoritmo, se han diseñado e integrado varias heurísticas avanzadas que superan el planteamiento base del problema.

Elección de clavo inicio ponderado:

El enunciado plantea dos posibles estrategias de selección (elegir un clavo origen y probar todas sus líneas o mediante distancias deLineaBresenham). Se ha optado por potenciar la primera estrategia mediante una heurística de selección ponderada.
En lugar de elegir el clavo de inicio de forma aleatoria y pre-cálcular al inicio del programa se analiza la oscuridad de todos los clavos media de los píxeles de la imagen original en su vecindad (trazando un semicírculo proyectado hacia el centro de la imagen, lo que nos interesa). Con estos, se asigna una probabilidad a cada clavo. Esta decisión técnica es crucial porque garantiza que el origen de las líneas nazca en áreas que necesitan gran concentración de tinta (hilos) acelerando la convergencia del error. Proyectando una distribución normal en función a la oscuridad.

Selección de los mejores hilos:

Al evaluar todas las posibles líneas desde un clavo origen hacia el resto de destinos, nos quedamos con los s mejores hilos siempre y cuando generen mejora, reduciendo en gran medida las iteraciones necesarias hasta la aproximación de la solución a coste de algo de precisión. Como mejora algorítmica fundamental, en lugar de guardar todas las reducciones en un vector y ordenarlo al final (lo cual tendría un coste de O(N log N) por iteración), se ha implementado un Montículo de Mínimos (Min-Heap). A medida que se evalúa cada destino v, el candidato se inserta en el montículo; si este supera el tamaño máximo s, se extrae automáticamente la raíz. Esta heurística reduce el coste de selección a O(N log s), garantizando un filtrado extremadamente rápido de los mejores hilos sobre la marcha.

Mapa de Importancia por Gradiente:

Si se deja al algoritmo operar únicamente basándose en la oscuridad (error cuadrático), tiende a rellenar de hilos las zonas anchas y oscuras, difuminando los contornos que es lo más significativo de las imágenes. Para solucionarlo, se ha implementado un Mapa de Importancia basado en el gradiente espacial de la imagen. Donde antes de comenzar el trazado, se calcula la diferencia de intensidad entre píxeles adyacentes en los ejes X e Y para toda la imagen. Una magnitud de gradiente alta indica la presencia de un borde, un vértice o un cambio brusco de contraste. Los píxeles pertenecientes a estos bordes reciben un multiplicador de importancia (de hasta 3.0 veces). Durante la simulación, la reducción de error calculada por la línea de Bresenham se modula multiplicándola por la importancia de los píxeles que cruza. Esto engaña positivamente al algoritmo voraz, forzándolo a cazar y definir los contornos y detalles finos de la imagen.

Multithreading:

El cálculo de la reducción de error para cientos o miles de hilos en cada iteración es un proceso altamente costoso, afortunadamente podemos ejecutarlos en paralelo. Para llevar el rendimiento al máximo, se ha implementado una arquitectura de multiprocesamiento paralelo haciendo uso de std::thread. El espacio de búsqueda se divide equitativamente en trozos, asignando cada bloque a un hilo de ejecución (aprovechando la concurrencia hardware máxima de la CPU). Donde cada hilo de ejecución posee su propio Min-Heap local. Cada subproceso calcula y filtra sus mejores s candidatos de forma totalmente aislada. Una vez que todos los subprocesos finalizan su tarea, el hilo principal fusiona los montículos locales en uno global.

Análisis de Complejidad y Experimentación

Complejidad Teórica:

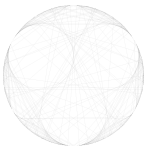
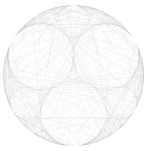
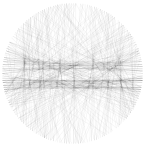
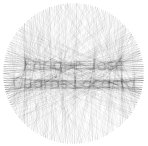
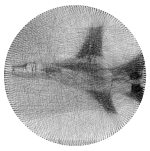
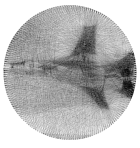
P: Píxeles totales | C: Nº de clavos | L: Long. máx. de línea | H: Hilos totales objetivo | T: Threads | S: Mejores hilos

LineaBresenham: O(L) **Carga imagen / Mapa gradiente:** O(P) **Pesos de clavos:** O(C) **Inicialización globa**l: O(P + C + P/C) **Selección clavo inicial:** O(log C) **Bucle greedy principal**: O((H/S)×C×L×log(S)/T)	**Base (sin heurísticas)**: O(P + H·C·L/S + H·L) **USE_WEIGHTED_NAIL**: O(P + P/C + H·C·L/S·log C) **USE_TOP_S_HEAP**: O(P + H·C·L/S) *(Sin cambio asintótico)* **USE_GRADIENT_MAP**: O(2P + H·C·L/S) **HEAP + MULTITHREADING**: O(P + H·C·L/(S·T) + H·T·log S) **v_all (todas activas)**: O(P/C + H·C·L/(S·T) + H·T·log S)

Análisis de resultados:

```
./comparativa.sh -c 190 -H 3000 -s 1
▸ Imagen: apollonian_gasket.ascii
    Sin heurísticas (base)...              877 ms | hilos:  480 | MSE:    384.613
    + Clavo Ponderado...                    939 ms | hilos:  498 | MSE:    385.094
    + Min-Heap Top-S...                     916 ms | hilos:  510 | MSE:    384.232
    + Gradiente...                         1113 ms | hilos:  631 | MSE:    400.274
    + Threads + Heap...                     560 ms | hilos:  510 | MSE:    384.232
    Todas las heurísticas...               680 ms | hilos:  617 | MSE:    401.109
```

Las heurísticas que informan y mejoran las soluciones aumentan el cómputo del programa, el que mas el gradiente al permitir aumentar las iteraciones mientras esté justificado al ser un contorno. Y los Threads mejoran de forma muy notable su velocidad.

Sin Heurísticas:	Con Heurísticas:	Comentarios:
		Como vemos la imagen gracias a las heurísticas anteriormente mencionadas permiten una mayor visualización de la imagen pasando de verse 3 circunferencias a 6 para la misma entrada. Con esta poca cantidad de hilos la diferencia es muy notable.
		Al igual que antes sucede parecido, las heurísticas mejoran el problema. Donde la que más ayuda es la de gradiente, que permite una mayor cantidad de fallos gracias a su priorización antes mencionada donde les da un menor error ficticio. Otro buen ejemplo debido a su cantidad de detalle.
		Para imágenes demasiado complejas o tonos muy oscuros con una gran cantidad de hilos la mejora es menos notable. Lo usaremos para comparar aproximaciones del mismo modelo en función a su entrada.