UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

REPRESENTACIÓN DE IMÁGENES MEDIANTE GRAFOS PLANARES

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

JOSÉ MIGUEL TRIVIÑO ÁLVAREZ

PROFESORA GUÍA:
NANCY HITSCHFELD KAHLER

PROFESOR CO-GUÍA:
SERGIO SALINAS FERNÁNDEZ

MIEMBROS DE LA COMISIÓN:
MAURICIO PALMA LIZANA
GONZALO NAVARRO BADINO

SANTIAGO DE CHILE
2023

Resumen

El presente trabajo se sitúa en el ámbito de la detección de bordes y la generación de mallas poligonales. En este contexto, se destaca el algoritmo Polylla, el cual permite transformar mallas de triángulos en mallas de polígonos. Estas mallas poligonales encuentran aplicaciones en diversos campos, especialmente en simulaciones que emplean métodos numéricos y que imponen restricciones geométricas específicas. En tales casos, herramientas como Polylla resultan de gran utilidad al permitir la adaptación de las mallas para cumplir con dichas restricciones.

El objetivo de este trabajo es automatizar la extracción de bordes a partir de imágenes de referencia para generar archivos de entrada compatibles con el software Polylla. Anteriormente, esta generación de bordes se realizaba de forma manual, lo que implicaba trazar el contorno de las imágenes punto por punto y luego transformarlos al formato aceptado por Polylla. Estos pasos requerían el uso de varias herramientas y la modificación de código para cada caso, lo cual resultaba en un proceso laborioso y que demandaba una considerable cantidad de tiempo. Con el fin de simplificar este proceso, se ha desarrollado una herramienta que automatiza la extracción de bordes utilizando dos métodos diferentes para explorar sus fortalezas y debilidades respectivas.

El primer método se basa en el reconocido algoritmo de Canny, ampliamente utilizado en la detección de bordes, para transformar los resultados entregados por este en listados de vértices y aristas. El segundo método se basa en el uso de mallas de triángulos y el desplazamiento de vértices, con el fin de lograr el posicionamiento óptimo de estos últimos y obtener aristas que reflejen la geometría de la imagen de referencia.

A lo largo del desarrollo de esta memoria se exploraron diversas variantes de cada algoritmo con el fin de encontrar las combinaciones óptimas de parámetros y las estrategias más eficaces en la generación de geometrías precisas. A partir de este proceso se logró identificar combinaciones de parámetros que generaran resultados ideales para cada caso, obteniéndose figuras con una alta similitud a las imágenes de referencia. Estas figuras resultan óptimas para la generación de mallas de triángulos, caracterizándose por una distribución uniforme de longitudes de aristas y una cantidad reducida de vértices, lo que permite su triangulación y posterior uso como entrada para Polylla.

Agradecimientos

En primer lugar, quiero agradecer a mi familia por su apoyo constante y su cariño. A pesar de la distancia, siempre me sentí acompañado por ustedes durante todos estos años.

También quiero agradecer a mis amigos por hacer de mi paso por la universidad una experiencia llena de lindos momentos. Un especial agradecimiento a Pamelita, por ser la mejor roomie y por la positividad que transmites constantemente.

Finalmente, extiendo mi gratitud a mis profesores guía, Nancy y Sergio, por compartir su conocimiento y orientarme durante este proceso. Sus constantes palabras de aliento fueron fundamentales para llevar a cabo este trabajo.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Contexto

El presente trabajo de memoria se enmarca en las áreas de geometría computacional y procesamiento de imágenes, específicamente en la detección de bordes y la generación de mallas de polígonos. Como contexto, las mallas poligonales tienen múltiples aplicaciones que trascienden el ámbito de la computación gráfica, entre las cuales se encuentran sistemas de tipo CAD (Computer-aided design), simulaciones de diversos tipos e incluso en la astronomía.

Las mallas compuestas de triángulos y cuadriláteros son comunes en simulaciones que emplean el método de elementos finitos (FEM), método numérico utilizado para aproximar soluciones a ecuaciones diferenciales parciales. Sin embargo, este método impone restricciones sobre la geometría de cada polígono, lo que requiere llevar a cabo modificaciones en las mallas para cumplir con dichos requisitos. Específicamente, se hace necesario agregar nuevos puntos para subdividir los polígonos, lo que conlleva un aumento en el tiempo utilizado para realizar las simulaciones. No obstante, nuevos enfoques como el VEM (Virtual Element Method) permiten la utilización de polígonos de forma arbitraria, abriendo así la posibilidad al uso de mallas basadas en polígonos diferentes a los triángulos.

Existen diversos algoritmos utilizados para generar de mallas de polígonos. Uno de ellos es el algoritmo Polylla (Polygonal meshing algorithm based on terminal-edge regions) [14], desarrollado por el estudiante de doctorado Sergio Salinas. Este algoritmo es capaz de generar una malla a partir de una triangulación inicial, permitiendo reducir el número de polígonos en una malla compuesta por triángulos, sin que esta reducción comprometa el desempeño de la malla al utilizarla para la resolución de ecuaciones mediante el VEM. Este algoritmo cuenta con una implementación en C++ denominada Polylla-Mesh, la cual se encuentra disponible en un repositorio público, lo que permite su utilización en diversos proyectos.

La implementación mencionada previamente es capaz de recibir una triangulación en varios formatos de archivo y generar una malla de polígonos a partir de ella. Sin embargo, la creación de estos archivos puede ser un proceso bastante largo, especialmente cuando se desea obtener una geometría basada en una imagen de referencia. En tales casos, se hace necesario utilizar diversas herramientas para trazar manualmente el contorno de las figuras y triangularlas antes de poder ingresarlas como entrada al algoritmo.

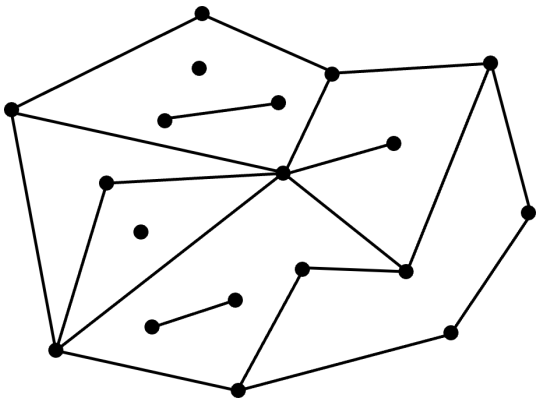

Figura 1.1: Ejemplo de Planar Straight Line Graph.

Para abordar este problema, se busca desarrollar una herramienta que automatice el paso inicial de este proceso, específicamente la extracción de los bordes presentes en imágenes de referencia para generar un PSLG (Planar Straight Line Graph), ilustrado en la figura 1.1, que pueda ser utilizado tanto para generar triangulaciones como para otros propósitos. Para lograr esta funcionalidad, es necesario emplear un algoritmo que detecte características relevantes en una imagen y genere un archivo con extensión .poly que represente el PSLG resultante, el cual al ser triangulado pueda utilizarse como entrada para el mallador Polylla. Dado que existen numerosos estudios en las áreas de detección de bordes y triangulación de imágenes, se propone adaptar uno de los algoritmos existentes para la generación de archivos .poly a partir de una imagen de referencia.

Uno de los aspectos más relevantes para obtener los resultados deseados es asegurar que la ubicación de los vértices en la geometría de la figura obtenida no sea arbitraria. Es decir, se espera que el grafo generado cumpla con ciertos requisitos, como minimizar la variación en el largo de las aristas y preservar información relevante sobre el contorno de la imagen. El propósito de esto es garantizar que, al proporcionar la figura obtenida como entrada a los diferentes softwares de generación de mallas, los polígonos generados tengan la geometría adecuada para su uso.

1.2. Solución Actual

Actualmente, la generación de archivos utilizados como entrada para el mallador Polylla implica múltiples pasos, los cuales se enumeran a continuación:

1. **Uso de software de edición de imágenes:** Si se desea utilizar una imagen de referencia para generar las geometrías, se requiere un software de edición de imágenes que permita la creación de archivos SVG. Un ejemplo de este tipo de software es GIMP, que cuenta con una herramienta de creación de caminos llamada Path [7]. Mediante esta herramienta, es posible trazar el contorno de una figura punto por punto para crear un archivo SVG que defina su borde. Sin embargo, este proceso implica un trabajo manual por parte del usuario, quien debe decidir qué partes del contorno son más relevantes y, por lo tanto, requieren la inserción de puntos.

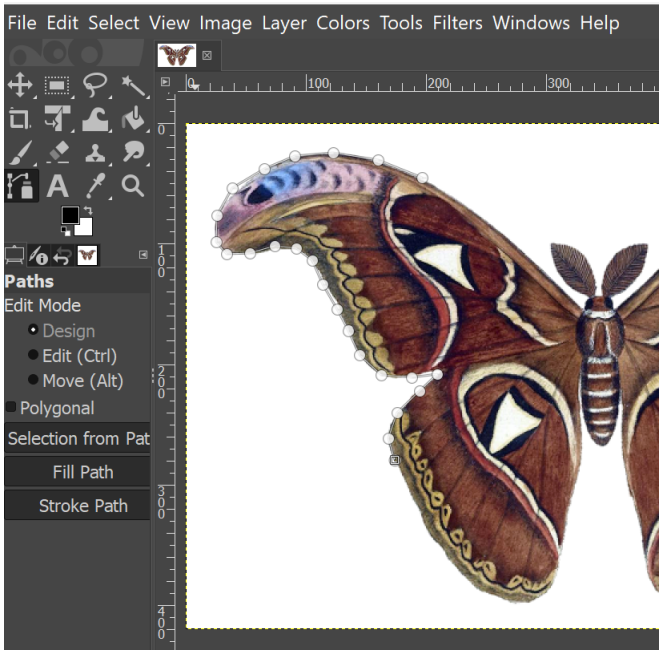

Figura 1.2: Captura de pantalla del software GIMP, utiilizado para la creación de caminos.

   En la figura 1.2 es posible visualizar el proceso de creación de caminos utilizando la herramienta GIMP. El contorno es manualmente trazado sobre la imagen de referencia, posicionando los puntos de forma manual, los cuales son automáticamente conectados mediante líneas rectas.

   Una limitación del proceso de creación de caminos es que cada camino generado debe guardarse en un archivo separado. Es decir, si una figura está compuesta por múltiples caminos cerrados, se debe crear un archivo SVG para cada camino trazado.

2. **Conversión de archivos SVG a formato .poly:** Una vez generados los archivos SVG que definen los bordes de la figura, es necesario ingresar todos estos puntos en un archivo con extensión .poly, que es el formato aceptado por Polylla. La estructura de datos de un archivo .poly consta de vértices, aristas y puntos interiores que definen agujeros en la figura.

Actualmente esta conversión se realiza mediante un script de Python, el cual recorre y analiza las figuras definidas en cada archivo SVG. Sin embargo, el script requiere modificaciones manuales para cada archivo que se desea crear, como reemplazar los nombres de los archivos que se desean leer. Además, se debe agregar manualmente las coordenadas los puntos interiores de cada polígono en caso de que estos representen agujeros dentro de la figura, siento insertadas en las líneas correspondientes del script de Python para luego ser incluidas en el archivo .poly.

3. **Generación de una triangulación de Delaunay:** Una vez que se tiene el archivo .poly que define los vértices, aristas y puntos interiores de los agujeros de la figura, es necesario generar una triangulación de Delaunay que utilice estos elementos como base. Para esto, se utiliza un software especializado como Triangle o Detri2qt, los cuales permiten generar una malla de triángulos que posteriormente puede ser utilizada como entrada para Polylla.

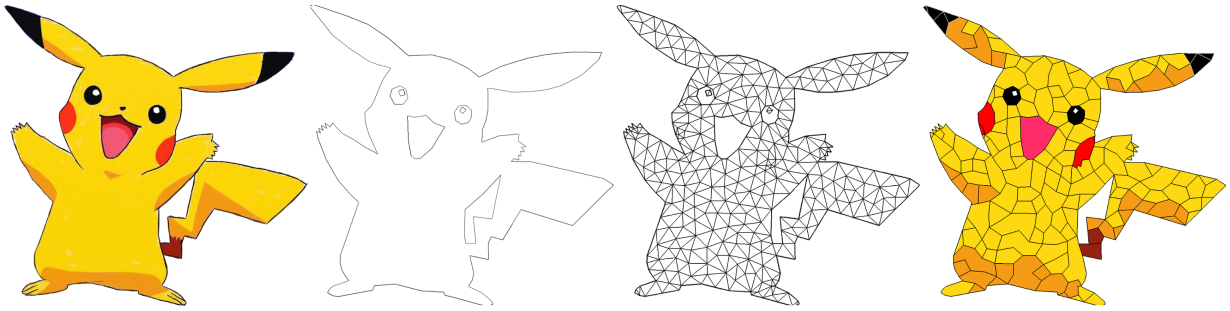

Figura 1.3: Distintas etapas del procesamiento de una imagen. Personaje propiedad de Nintendo Company, Ltd.

En la figura 1.3 se muestran las diferentes etapas descritas previamente para generar una malla de polígonos a partir de una imagen. El proceso comienza con la imagen original, a partir de la cual se realiza un trazado manual del PSLG que define los contornos de la figura. Luego, este PSLG es triangulado utilizando un software especializado en esta tarea. Una vez completado este proceso, es posible transformar la malla de triángulos resultante en una malla de polígonos utilizando Polylla. Es importante destacar que la coloración de la imagen final fue realizada manualmente, por lo que los colores de los polígonos pueden no coincidir con los de la imagen de referencia.

Como se puede observar, el proceso de generación de los archivos utilizados como entrada para Polylla involucra múltiples pasos y requiere una cantidad significativa de trabajo manual por parte del usuario, incluso siendo necesario realizar modificaciones de código para cada imagen que se desea procesar. Por lo tanto, contar con una herramienta que automatice este proceso resultaría de gran utilidad, ya que permitiría al usuario ahorrar una considerable cantidad de tiempo al generar los archivos de entrada requeridos por el software.

1.3. Objetivos

1.3.1. Objetivo general

Diseñar e implementar una herramienta capaz de detectar segmentos y regiones relevantes en una imagen, con el fin de crear un PSLG (Planar Straight Line Graph) que pueda ser utilizado posteriormente como entrada para su posterior triangulación. Se busca que este PSLG tenga vértices y aristas que reflejen con precisión los bordes de la imagen original, con un grado de detalle ajustable según las preferencias del usuario.

1.3.2. Objetivos específicos

1. Diseñar e implementar un método basado en el algoritmo de Canny para la obtención de los bordes de una imagen como un listado de aristas y vértices.
2. Diseñar e implementar un método basado en triangulaciones para la obtención de los bordes de una imagen como un listado de aristas y vértices.
3. Crear un script que transforme las geometrías generadas por ambos algoritmos en un archivo .poly que pueda ser triangulado mediante un software externo.
4. Comparar los resultados entregados por ambos algoritmos relativo a la aproximación con la imagen de referencia.
5. Evaluar el desempeño en tiempo de cálculo de las soluciones.

1.4. Evaluación

La evaluación de los resultados se realizó inicialmente mediante la comparación visual entre los gráficos generados por cada algoritmo y las imágenes de referencia. De esta manera, se determinó la correspondencia entre los bordes reales de la imagen y aquellos obtenidos de forma automática. Mediante la inspección visual se evaluó la robustez de ambos algoritmos frente al ruido presente en las imágenes originales, así como la capacidad de preservar los detalles más relevantes en estas.

Además de realizar una inspección visual de las mallas generadas, se llevó a cabo una evaluación utilizando distintas métricas para los archivos generados. Se analizó la cantidad de aristas y vértices presentes, así como la distribución de las longitudes de las aristas generadas.

Junto con la evaluación en términos de la eficacia de los resultados obtenidos, también se midió la eficiencia de ambos algoritmos, siendo el objetivo de esto determinar cuál de los algoritmos presenta un menor tiempo de cálculo.

Para todas las instancias de evaluación, se realizaron variaciones en los parámetros de ejecución de los algoritmos. Esto se llevó a cabo con el propósito de observar el efecto de estos cambios en los resultados finales y, a partir de ello, determinar las combinaciones de parámetros ideales para la generación grafos con resultados utilizables por el usuario.

1.5. Estructura del Documento

En los siguientes capítulos se abordan los conceptos previos, el diseño y la implementación de la solución propuesta, así como los resultados obtenidos y su evaluación. Los capítulos son los siguientes:

El capítulo 2, titulado "Marco Teórico", se enfoca en presentar conceptos relevantes para el lector, incluyendo tecnologías, herramientas y algoritmos de detección de borde utilizados en esta memoria. También se explorarán conceptos relacionados con las mallas poligonales, su refinamiento y optimización, y se explicarán algunos algoritmos que resultaron útiles para el desarrollo de este trabajo.

En el capítulo 3, titulado "Diseño e Implementación", se presentará la estructura general del trabajo realizado, seguido del análisis detallado de los dos métodos de detección de bordes utilizados. Se explicará el diseño y las decisiones tomadas para la implementación de cada uno, así como los detalles técnicos del código.

En el capítulo 4, titulado "Resultados y Análisis", se expondrán los resultados obtenidos tras la ejecución de cada método de detección de bordes. Primero se analizará su efectividad en cuanto a la calidad de los resultados obtenidos, y luego se discutirá la eficiencia en cuanto a tiempo de ejecución. Además, se discutirán las limitaciones de cada método en base a los resultados obtenidos.

Finalmente, el capítulo 5, titulado "Conclusión", resume todos los objetivos logrados en este trabajo y presenta una sección donde se proponen ideas para trabajo futuro con el fin de agregar mayor funcionalidad a la herramienta desarrollada.

Capítulo 2

Marco Teórico

El problema de la detección de bordes en imágenes es un tema que ha sido ampliamente investigado y existen numerosos trabajos al respecto [1]. Sin embargo, hasta el momento no se ha desarrollado una herramienta dedicada exclusivamente a la extracción de bordes de imágenes en un formato adecuado para la generación de mallas de polígonos.

En la actualidad, una solución sencilla a este problema es utilizar librerías de manejo de imágenes, como OpenCV [3] en C/C++, ImageJ [13] en Java, o PIL [5] en el caso de Python. Todas estas librerías ofrecen funciones para la extracción de bordes en imágenes y también proporcionan herramientas para representar estos bordes como un arreglo de aristas. No obstante, estas funciones generan listados con una cantidad significativa de puntos, ya que no llevan a cabo una reducción del número de vértices. Por lo tanto, su utilidad para la generación de mallas de polígonos es limitada si no se realiza un procesamiento previo.

2.1. Polylla-Mesh

Con respecto al algoritmo Polylla previamente mencionado, existe una implementación de este llamada Polylla-Mesh [12], escrita principalmente en C++ y con partes del código en C, utilizando la librería Detri2 como apoyo para su funcionamiento. Se emplea la herramienta CMake para permitir su compilación en múltiples plataformas, lo que facilita su uso en diversos sistemas operativos. En cuanto a la generación de las geometrías utilizadas como entrada, el repositorio de GitHub proporciona scripts auxiliares de Python encargados de la creación de estos.

En la actualidad, se puede utilizar Polylla-Mesh como un script, el cual recibe archivos binarios de distinto formato que representan puntos o triangulaciones como entrada. Entre los tipos de archivo admitidos se incluyen .poly, .ele y .node. A partir de estos archivos de entrada es posible generar una malla de polígonos, la que luego puede ser almacenada en diversos formatos de archivo, como .vem, .off, .gid o .svg.

2.2. Formato .poly y Triangle

Un archivo .poly representa un PSLG (Planar Straight Line Graph) mediante un listado de vértices y segmentos, siendo este tipo de archivo utilizado para definir geometrías tanto en 2D como en 3D. Además, es el principal tipo de entrada recibido por el software Triangle [16], el cual genera triangulaciones de Delaunay a partir de la geometría proporcionada, las cuales luego son entregadas como entrada al mallador Polylla.

Un archivo .poly se compone de las siguientes secciones:

1. La primera sección es un listado de vértices, donde cada vértice se define por un índice (comenzando desde el 0) y sus coordenadas espaciales.
2. La segunda sección es un listado de segmentos, los cuales también son indexados a partir de 0 y se definen por los vértices que los componen. Estos vértices se representan mediante los índices asignados en la sección anterior.
3. La tercera sección representa los agujeros en la triangulación, en caso de existir. Los agujeros se definen de manera similar a los vértices, con un índice correspondiente y coordenadas espaciales.

En el contexto de un archivo .poly, se considera un agujero como una región cerrada dentro de la triangulación en la cual no se insertarán vértices ni se generarán triángulos, y que a su vez se encuentra contenida dentro de una región que sí forma parte de la triangulación. En la figura 2.1 se ejemplifica este concepto, con las regiones azules representando los agujeros en la malla, mientras que las regiones blancas son parte de la triangulación.

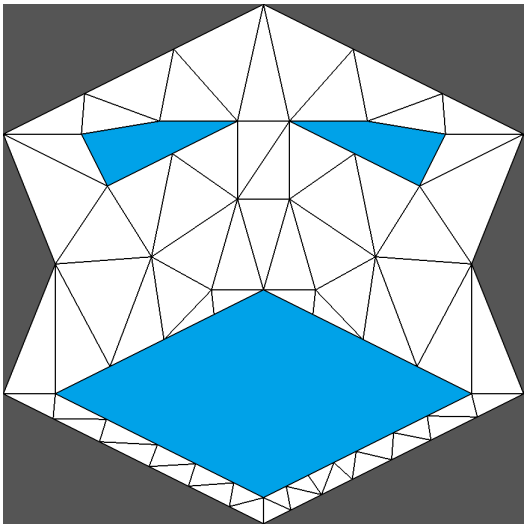

Figura 2.1: Ejemplo de triangulación con agujeros. Fuente: Triangle [15].

Al recibir el archivo .poly como entrada, Triangle generará una triangulación de Delaunay respetando los vértices y aristas proporcionados. Incluso si las aristas entregadas no forman un ciclo cerrado, el software realizará las triangulaciones correspondientes. Sin embargo, esto resultará en una malla que abarcará toda la envolvente convexa de los vértices presentes en el archivo. Es importante destacar que esta triangulación se realizará en todas las áreas interiores de la imagen, ya que la generación de agujeros se llevará a cabo en un paso posterior.

Para generar agujeros dentro de la malla, el software Triangle considera cada uno de los puntos presentes en el listado de agujeros del archivo .poly y elimina todas las aristas generadas a partir de ese punto hasta llegar a aquellas que no pueden ser eliminadas, es decir, aquellas ingresadas por el usuario. En consecuencia, si los segmentos que rodean a un punto que define un agujero no forman un ciclo cerrado, existe la posibilidad de que todos los triángulos generados sean eliminados.

Es importante tener en cuenta que en un archivo .poly el orden de definición de las aristas no es relevante. Por lo tanto, no es necesario definir aristas que formen un ciclo cerrado de manera consecutiva, ni es necesario que estas tengan el mismo sentido. Sin embargo, en este trabajo se sigue la convención de definir todos los contornos generados en sentido antihorario, mientras que los caminos que definen agujeros en la figura se establecen en sentido horario.

2.3. Algoritmos de detección de bordes

En relación a los algoritmos de detección de bordes, existe una amplia investigación en este campo [1]. Un algoritmo destacado es el Operador Sobel [17], el cual calcula el cambio en la intensidad de los pixeles de una imagen. A partir del Operador Sobel se deriva el Algoritmo de Canny [4], que se considera uno de los métodos más utilizados para la detección de bordes. Ambos algoritmos producen una imagen en formato de mapa de bits donde los bordes detectados son resaltados.

2.3.1. Operador Sobel

El Operador Sobel se aplica sobre imágenes en escala de grises. Su funcionamiento consiste en recorrer la imagen y calcular una aproximación del gradiente de la intensidad sobre cada pixel. Esto es, para cada pixel en la imagen se analizan los 8 pixeles adyacentes y se obtienen dos valores, los cuales representan el cambio en intensidad en dirección vertical y horizontal. Estos cálculos se realizan por separado.

Una vez obtenidos los valores del cambio de intensidad en ambas direcciones, se calcula la magnitud total del gradiente sobre cada pixel utilizando la distancia euclidiana. Esto produce una imagen en escala de grises en la que los pixeles más claros representan un mayor cambio de intensidad en la imagen original.

Este método de detección de bordes es computacionalmente simple, ya que sólo requiere recorrer los pixeles de la imagen en dos pasadas y realizar operaciones sencillas. Sin embargo, entre las principales desventajas del Operador Sobel se encuentran su sensibilidad al ruido

en las imágenes y la irregularidad de los grosores de los bordes entregados. Los resultados obtenidos al aplicar este operador a una imagen se muestran en la figura 2.2.

Figura 2.2: Resultado de aplicar el Operador Sobel sobre una imagen. Imagen original de dominio público.

2.3.2. Algoritmo de Canny

Como respuesta a las limitaciones del método anterior, surge el Algoritmo de Canny, el cual aplica el Operador Sobel a una imagen preprocesada. Este preprocesamiento implica aplicar un filtro gaussiano a la imagen inicial para difuminarla y reducir el ruido. Luego de esto, se aplica el Operador Sobel de la misma forma descrita previamente.

Una vez obtenido el resultado de aplicar el Operador Sobel sobre la imagen, se procede a reducir el ancho de los bordes detectados. Esto se logra buscando los máximos locales dentro de cada borde y reduciendo el ancho de estos a 1 pixel. Posteriormente, se lleva a cabo una reducción de la cantidad de bordes detectados para conservar solo aquellos de mayor relevancia.

Este último paso implica clasificar los pixeles de los bordes según su intensidad, utilizando dos umbrales definidos por el usuario: un umbral superior y un umbral inferior. Los bordes cuya intensidad supera el umbral superior son clasificados como pixeles de borde fuertes, y siempre son conservados. Los pixeles cuya intensidad no supere el umbral inferior son clasificados como pixeles de borde débiles y siempre son descartados. Finalmente, aquellos pixeles que se encuentren entre ambos umbrales se conservan únicamente si son adyacentes a un pixel de borde fuerte y se clasificarán de esta misma manera.

Aunque el algoritmo descrito anteriormente se desarrolló en las etapas tempranas de la computación gráfica, todavía es utilizado para la detección de bordes. Esto se debe a que los algoritmos que ofrecen mejores resultados en términos de imágenes y bordes detectados suelen requerir más tiempo de cálculo o una mayor cantidad de parámetros ingresados manualmente por el usuario.

Es importante destacar que tanto el Operador Sobel como el Algoritmo de Canny generan como resultado imágenes en formato de mapa de bits. Dado que la herramienta que se

desea desarrollar debe entregar PSLG en formato .poly, los resultados obtenidos mediante la ejecución de estos algoritmos representan simplemente un paso inicial hacia la obtención de estas geometrías.

Los resultados obtenidos de aplicar este algoritmo a una imagen se muestran en la figura 2.3.

Figura 2.3: Resultado de aplicar el algoritmo de Canny sobre una imagen. Imagen original de dominio público.

2.4. Triangulación de imágenes

En relación a la generación de mallas poligonales basadas en imágenes de referencia, destaca el trabajo de Kai Lawonn y Tobias Günther, quienes en el artículo "Stylized Image Triangulation" [10] desarrollaron un método que permite la representación de imágenes mediante triángulos, donde las aproximaciones resultantes son de una alta calidad con respecto a las imágenes utilizadas como referencia. Un algoritmo de este tipo puede resultar útil para el trabajo que se desea realizar, ya que a partir de estas triangulaciones óptimas la obtención de los contornos de la imagen en formato PSLG resulta directa.

El funcionamiento del algoritmo se puede resumir en los siguientes pasos:

1. Se proporciona una imagen de referencia y una triangulación inicial, que puede ser independiente del contenido de la imagen. La cantidad de triángulos en esta malla puede ser fija o aumentar mediante la inserción de más triángulos en iteraciones posteriores.
2. Se calcula el color de los triángulos de la malla como el promedio del color de los pixeles contenidos en cada triángulo.
3. Se calcula el error de aproximación para cada vértice, que se determina mediante la diferencia entre el color promedio calculado y el color original de la imagen.
4. Para cada vértice en la triangulación, se realiza un desplazamiento en una distancia definida, con la dirección determinada por el gradiente de este. De esta manera, el

vértice se desplaza en la dirección que minimiza el error de aproximación de manera más significativa.

5. Se llevan a cabo distintos tipos de refinamiento y optimización en la malla, como la incorporación de nuevos elementos (vértices y aristas) para mejorar la calidad de la aproximación, y también la eliminación o sustitución de otros elementos para preservar la integridad de la malla.
6. Se alternan las etapas de desplazamiento de vértices y optimización de la malla hasta alcanzar una triangulación satisfactoria, según los parámetros definidos por el usuario

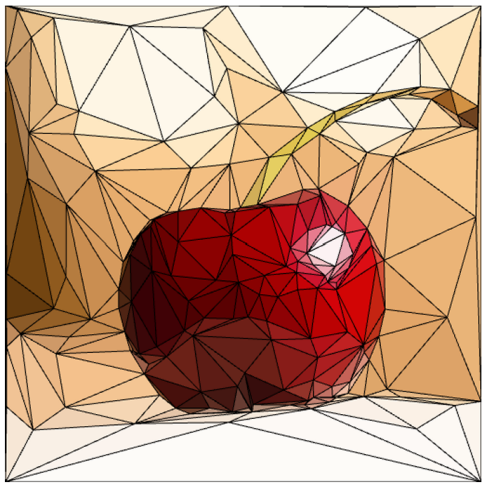

Figura 2.4: Resultado de aplicar el algoritmo de triangulación sobre una imagen. Fuente: Stylized Image Triangulation [10].

En la figura 2.4 se muestra un ejemplo de los resultados obtenidos al aplicar este algoritmo a una imagen. Se puede observar que cada triángulo está relleno con un color sólido, lo que permite representar las variaciones en intensidad de color mediante triángulos de diferentes tamaños. Se destaca la capacidad del algoritmo para capturar una gran cantidad de detalles finos presentes en la imagen de referencia, ya que de esta forma las mallas obtenidas pueden ser potencialmente utilizadas para la obtención de bordes de la imagen de manera más directa, sin pérdidas de detalle.

2.5. Mallas poligonales

En esta sección se tratarán conceptos relativos a las mallas poligonales, así como diferentes métodos para mejorar su estructura a través de la inserción de elementos como vértices y aristas, proceso al que se denomina refinamiento de mallas.

2.5.1. Estructura de datos half-edge

La estructura de datos *half-edge* [11] se utiliza para representar mallas de polígonos tanto en 2D como en 3D. Esta estructura contiene vértices, aristas y caras, donde cada arista se

divide en dos mitades llamadas *half-edges*. Cada *half-edge* tiene un punto de inicio y un punto final, siendo el punto de inicio de uno el punto final del otro. Estos dos *half-edges* que se definen por los mismos puntos se denominan *twins*, y cada *half-edge* tiene una referencia a su *twin* correspondiente.

La particularidad de esta estructura de datos radica en que es posible aprovechar la información sobre la orientación de los *half-edges* de distintas formas. Una de las principales aplicaciones es en la definición de las caras en la malla: como las aristas tienen orientación, es posible definir todas las caras utilizando *half-edges* con un mismo sentido. Por convención se utiliza el sentido antihorario para definir polígonos, por lo que cada triángulo en una malla estará definido por tres aristas en este sentido.

A raíz de lo anterior, se tiene que cada *half-edge* está asociado únicamente a un triángulo, a diferencia de otras estructuras de datos donde una arista suele tener un triángulo a cada lado. Además, la orientación de los *half-edges* en un triángulo permite establecer un ordenamiento, de modo que cada *half-edge* tiene una arista previa y una arista siguiente. En la figura 2.5 se ejemplifican los elementos relativos a una arista.

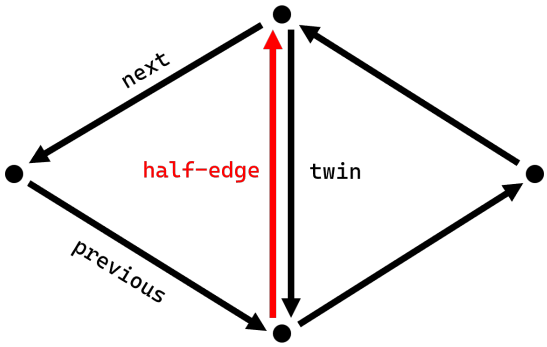

Figura 2.5: Representación de la estructura de datos *half-edge*.

De esta forma, se tiene que cada *half-edge* contiene información sobre su *twin*, así como sobre los *half-edges* que lo preceden y lo suceden dentro de un triángulo. Además, un *half-edge* puede hacer referencia a su triángulo asociado, mientras que cada vértice puede almacenar un listado de *half-edges* que se originan desde él. Teniendo toda esta información disponible es posible llevar a cabo distintos procesos de forma muy eficiente, como obtener todos los triángulos en torno a un vértice, incluso aunque en la malla los triángulos y vértices no se hagan referencia entre ellos.

Dependiendo de la implementación, los bordes de la malla pueden consistir en un *half-edge* sin un *twin* o un par de *half-edges* con su correspondiente *twin*, donde uno de estos no cuenta con un triángulo asociado. En este trabajo se optó por la primera implementación, lo que implica que la ausencia de un *twin* para un *half-edge* indica que este se encuentra en el borde de la malla. Esta decisión se tomó con el objetivo de reducir la cantidad de elementos presentes en la malla y, junto con esto, la cantidad de pasos necesarios para verificar si una arista específica está en el borde.

2.5.2. Triangulación de Delaunay

Una triangulación de Delaunay consiste en una malla de triángulos donde no existen circunferencias circunscritas que contengan vértices de otros triángulos. Es decir, para cada triángulo en la triangulación, la circunferencia circunscrita a dicho triángulo no debe contener ningún vértice de los triángulos adyacentes.

Esta propiedad implica que en una triangulación de Delaunay se maximiza el ángulo mínimo presente en la malla, lo que resulta en triángulos lo más equiláteros posible. Esto es beneficioso cuando se desea generar triangulaciones más uniformes, evitando así triángulos con ángulos extremadamente agudos y áreas muy reducidas.

Para dos triángulos que comparten una arista, es posible demostrar que ambos triángulos cumplen con la condición de Delaunay si y solo si la suma de los ángulos opuestos a la arista compartida es menor o igual a 180°. Por lo tanto, para verificar si una triangulación es de Delaunay basta con recorrer todas las aristas de la malla evaluando esta condición.

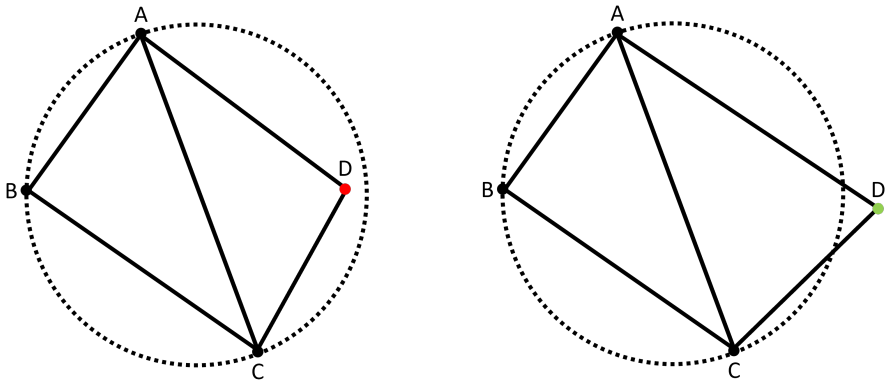

Figura 2.6: Representación de la condición de Delaunay.

Como se puede ver en la figura 2.6, para en el caso ilustrado en la izquierda se tiene que el triángulo ABC no cumple con la condición de Delaunay, ya que el punto D se encuentra en el interior de la circunferencia circunscrita de dicho triángulo. En contraste, en la figura de la derecha sí se cumple la condición, ya que el punto D se encuentra fuera de la circunferencia circunscrita.

2.5.3. Edge-flip

Un *edge-flip* [6] es una operación de optimización de mallas, la cual consiste en seleccionar una arista específica de la malla y reemplazarla por una nueva arista que conecte los dos vértices opuestos a la arista original, como se muestra en la figura 2.7.

Esta operación es comúnmente realizada cuando se desea generar una triangulación de Delaunay a partir de una triangulación inicial donde no se cumple esta condición. Para lograr esto, se lleva a cabo un *edge-flip* en cada arista donde no se cumpla la condición de Delaunay, lo que implica calcular la suma de los ángulos opuestos sobre cada arista.

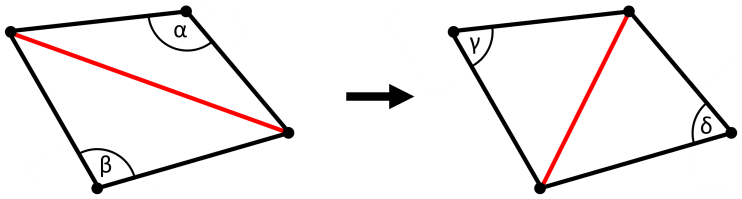

Figura 2.7: Resultado de aplicar un *edge-flip* sobre una arista.

En el ejemplo de la figura 2.7, la suma de α y β es mayor a 180°, lo que indica que la condición de Delaunay no se cumple. Por lo tanto, se realiza un *edge-flip* en esa arista, resultando en dos nuevos ángulos γ y δ, cuya suma es menor a 180°, cumpliendo así la condición de Delaunay.

Esta operación debe llevarse a cabo de forma recursiva cada vez que se ejecute un *edge-flip*. Por lo tanto, en cada iteración es necesario evaluar el cumplimiento de la condición de Delaunay para las cuatro aristas que rodean la arista recientemente modificada. Si no se cumple la condición, se realiza un nuevo *edge-flip* en la arista correspondiente. Este proceso continúa hasta que se cumpla la condición de Delaunay para todas las aristas de la malla.

2.5.4. Edge-collapse

Un colapso de arista o *edge-collapse* [2] es un proceso que consiste en eliminar una arista presente en la malla, fusionando los vértices que la definen en uno solo. Esta operación también conlleva la eliminación de los triángulos adyacentes a la arista eliminada, así como de otras dos aristas en la malla. Este proceso es utilizado para simplificar la malla, ya que se reduce el número de vértices, triángulos y aristas.

El colapso de aristas puede tener beneficios en términos de memoria y tiempo de ejecución, ya que se reduce la cantidad de elementos presentes en la malla. Sin embargo, realizar demasiados colapsos de aristas puede resultar en una pérdida de calidad en la geometría que se desea representar.

En relación al *edge-collapse* existe también el *half-edge-collapse*, que difiere del proceso original en cuanto a la posición final del nuevo vértice generado. Cabe destacar que esta variante no tiene relación con la estructura de datos *half-edge*; la similitud de los términos es simplemente un alcance de nombres.

En un *edge-collapse* regular, los vértices de inicio y fin de la arista eliminada se reposicionan y fusionan en un nuevo vértice ubicado en el punto medio entre ambos. Por otro lado, en un *half-edge-collapse*, uno de los dos vértices se desplaza a la posición del otro. Esto simplifica el proceso, ya que uno de los vértices conserva su posición original, reduciendo así la cantidad de aristas que deben ser desplazadas. Por lo tanto, para la implementación de la solución desarrollada se optó por este último método para llevar a cabo la eliminación de aristas.

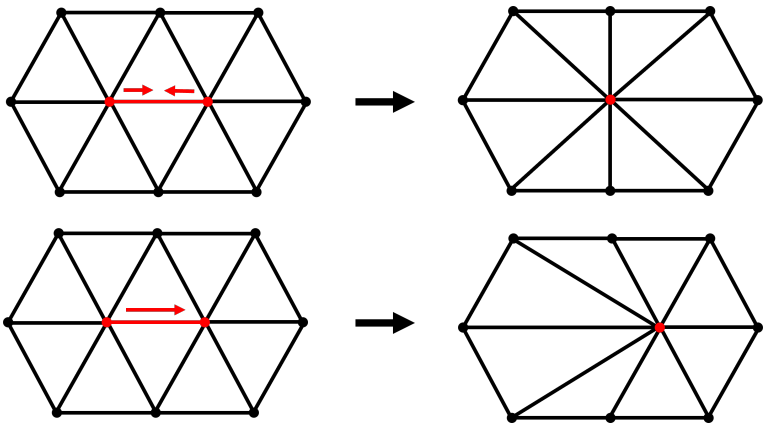

Figura 2.8: *Edge-collapse* y *half-edge-collapse*.

En la figura 2.8 se puede observar la geometría resultante después de realizar un *edge-collapse* regular (arriba) y un *half-edge-collapse* (abajo). Es importante tener en cuenta que en ambos casos se debe preservar la integridad de la malla, evitando la generación de triángulos con área nula o negativa.

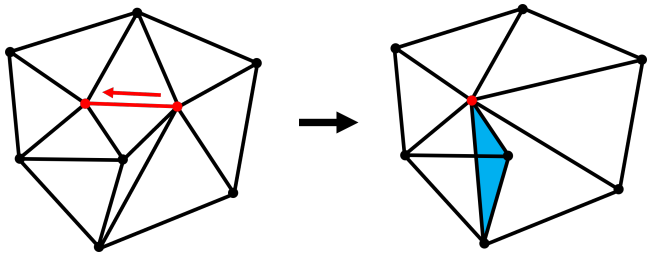

Figura 2.9: Triángulo con área negativa como resultado de un *half-edge-collapse*.

En la figura 2.9 se ilustra el escenario previamente descrito, resaltándose en azul el triángulo con área negativa que se genera al realizar un *half-edge-collapse*, una situación que también hubiese ocurrido con un *edge-collapse* regular. Además, se observa una intersección de dos aristas donde no hay un vértice presente en la intersección. Existen diversas soluciones para abordar estos casos, las cuales serán vistas en detalle en la etapa de la implementación.

Es crucial evitar la presencia de áreas negativas en una malla, ya que esto puede generar comportamientos indeseados durante los pasos del refinamiento y el cálculo de los colores promedio de cada triángulo. La principal consecuencia de las áreas negativas es la superposición de elementos en la malla, incluyendo vértices, aristas y triángulos. En el caso de estos últimos, su superposición resultaría en áreas de la imagen contenidas en múltiples triángulos. Este solapamiento puede generar resultados incoherentes y perjudicar el procesamiento de la imagen, por lo que esto debe ser siempre evitado.

2.5.5. Inserción de puntos

La inserción de puntos (también conocidos como puntos de Steiner [8]) consiste en agregar nuevos vértices dentro de la malla, lo cual implica también la creación de nuevas aristas que los conecten a los vértices ya existentes, preservando así la triangulación. Este proceso puede llevarse a cabo de dos formas: en una cara, o en una arista. A continuación se detallará el proceso llevado a cabo para cada caso.

En el caso de inserción en una cara, las coordenadas del nuevo vértice suelen corresponder al centroide del triángulo, es decir, el promedio de las coordenadas de sus 3 vértices. Al insertar un vértice, es necesario también crear 3 nuevas aristas que lo conecten a los vértices del triángulo en el que se realiza la inserción. Como resultado, el triángulo original es dividido en 3 nuevos triángulos. Un ejemplo de este tipo de inserción se muestra en la figura 2.10.

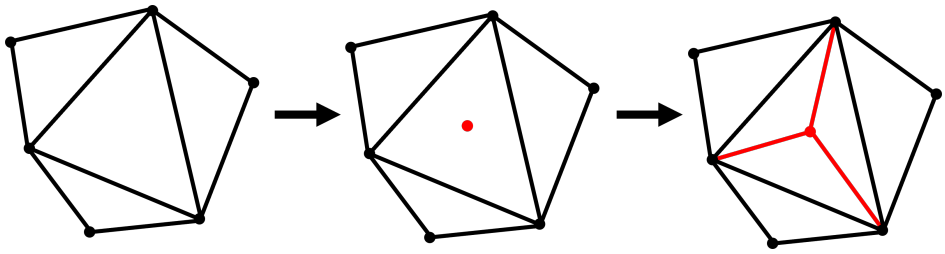

Figura 2.10: Ejemplo de inserción de puntos en una cara.

Para el caso de la inserción en aristas, las coordenadas del vértice insertado suelen corresponder al punto medio de la arista en la cual se realiza la inserción, es decir, el promedio entre el punto de inicio y el punto final de la arista. Además, la adición de un vértice implica la creación de dos nuevas aristas que lo conecten con los vértices opuestos a la arista original. Esto resulta en la subdivisión de los triángulos adyacentes a la arista en dos, añadiendo así dos nuevos triángulos a la malla. El resultado de realizar una inserción de este tipo se ilustra en la figura 2.11

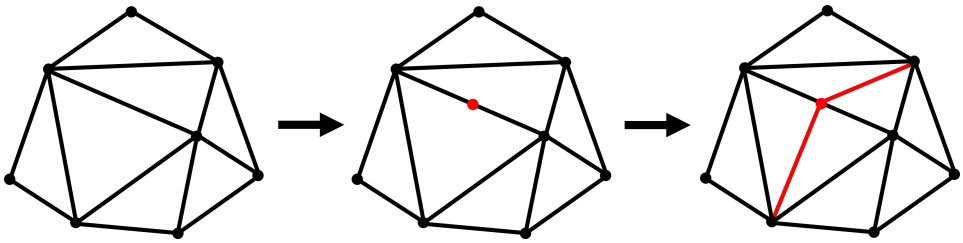

Figura 2.11: Ejemplo de inserción de puntos en una arista.

La inserción de puntos en la malla permite una aproximación más precisa de la imagen que se desea triangular, ya que al aumentar la cantidad de aristas y vértices se pueden capturar de mejor forma los detalles presentes, y se hace posible generar bordes con una mayor cantidad de puntos de inflexión. No obstante, es necesario tener precaución al realizar este proceso de forma excesiva, ya que esto puede generar una malla muy densa y afectar el rendimiento de los algoritmos en cuanto al tiempo de ejecución.

2.6. Algoritmos geométricos

A continuación, se profundizará en distintos algoritmos utilizados para obtener información relativa a los bordes detectados en una imagen:

2.6.1. Determinar la orientación de un polígono

Al utilizar la estructura *half-edge* para la representación de las mallas poligonales, se tiene que cada polígono, ya sea un triángulo individual o un polígono compuesto por varios triángulos, debe tener una orientación definida, que puede ser horaria o antihoraria. Esta característica se basa en el hecho de que los polígonos generados no presentan intersecciones consigo mismos, lo que implica que el sentido de recorrido del polígono es uniforme en toda la figura.

Para determinar la orientación de un polígono dado un conjunto de vértices que lo definen, se comienza seleccionando cualquier punto extremo del polígono [6], es decir, aquel con la mayor o menor coordenada en x o en y. Esto se debe a que para todo punto extremo se tiene que sus aristas adyacentes siempre formarán un ángulo menor a 180°. En este caso, se selecciona el punto más alto del polígono, al cual se denominará A.

Dado que la orientación se determina asumiendo que los vértices están ordenados, es posible obtener tanto el vértice anterior como el siguiente a A, los cuales se denominarán B y C, respectivamente. Al considerar que A es el punto más alto del polígono, se tiene que el ángulo formado por los vectores AB y AC no superará los 180 grados, ya que en caso contrario esto implicaría que B o C se encuentra en una posición más alta que A, lo cual contradice la suposición inicial.

Teniendo los vectores AB y BC, es posible calcular el producto cruz de estos. El signo del producto cruz obtenido otorgará información acerca de la orientación del polígono: si este es positivo, implica una orientación antihoraria, mientras que si es negativo implica una orientación horaria. Esto se debe a que el producto cruz no es conmutativo, lo que significa que la posición del vector AB con respecto a AC influirá en el valor resultante, ya que AB siempre es el primer término del producto calculado.

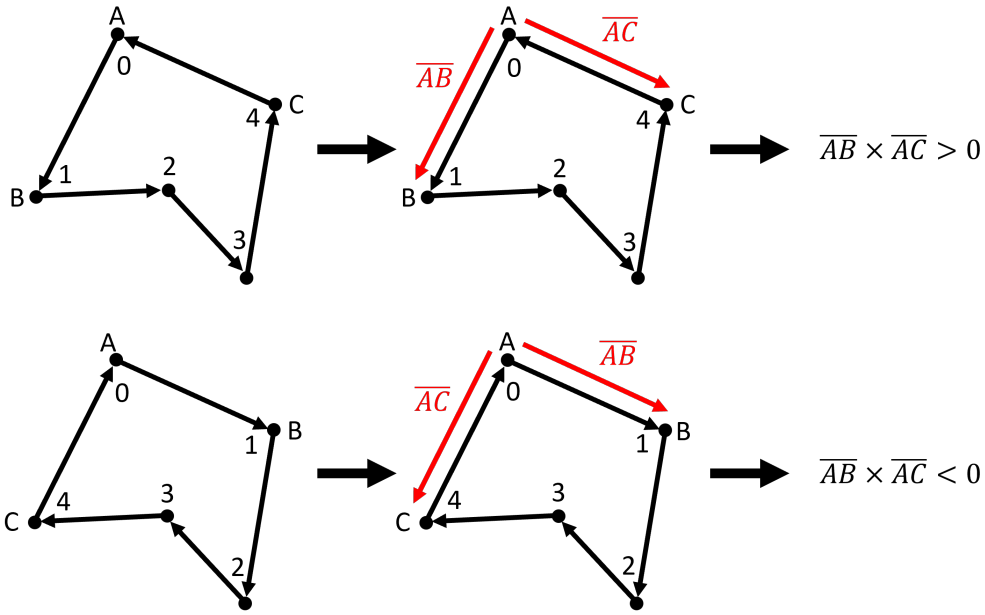

Figura 2.12: Ejemplo de cálculo de la orientación de un polígono.

En la figura 2.12 se muestra el cálculo de la orientación para un mismo polígono en dos casos distintos: primero para el caso antihorario y luego para el caso horario. Como se puede observar, en el segundo caso los vértices A y B intercambian posiciones, lo que implica que los vectores AB y AC también cambian. Como la fórmula utilizada para el producto cruz es la misma en ambos casos, lo único que cambiará será el signo del resultado, siendo mayor a 0 para el caso antihorario, y menor a 0 para el caso horario.

2.6.2. Determinar los puntos interiores de un triángulo

Para determinar los puntos contenidos en un triángulo sobre una imagen, se puede emplear un algoritmo basado en *scanlines* [6], cuyo tiempo de ejecución es proporcional a la diferencia entre la coordenada y del vértice más alto y la del vértice más bajo del triángulo. El algoritmo sigue los siguientes pasos:

Primero, se obtienen las ecuaciones de las tres rectas que definen el triángulo en cuestión. Utilizando estas ecuaciones, se calculará la intersección de estas rectas con cada horizontal que atraviesa al triángulo, es decir, la intersección con las rectas definidas por $y = y_i$, donde y_i toma valores dentro del intervalo entre la coordenada y más alta y la coordenada y más baja del triángulo.

Para cada altura de la línea horizontal $y = y_i$, se calcularán las intersecciones solo con dos de las tres rectas que definen el triángulo. Estos puntos de intersección representarán los límites de un intervalo en el que todos los puntos contenidos estarán dentro del triángulo. Para determinar qué rectas serán utilizadas para el cálculo de las intersecciones, es necesario etiquetar cada una de las rectas del triángulo según su posición relativa, denominándolas como `left`, `right` y `bottom`. Tanto la recta `left` como la recta `right` pasan por el vértice más alto del triángulo, mientras que la recta `bottom` está definida por los otros dos vértices.

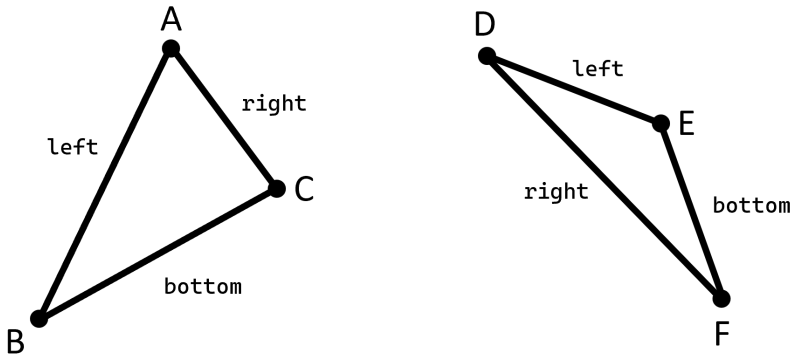

Figura 2.13: Ejemplo de etiquetado de las aristas de un triángulo.

En la figura 2.13 se ejemplifica el proceso de etiquetado de las aristas del triángulo. En todos los casos, la recta `bottom` no pasa por el punto más alto del triángulo. Sin embargo, nótese que para el triángulo DEF los segmentos `right` y `left` ocupan posiciones contrarias a lo esperado, caso borde que fue considerado al momento de la implementación.

Para determinar qué recta recibirá la etiqueta `left` y cuál la etiqueta `right`, es necesario realizar una evaluación de condiciones. Esto consiste en buscar el vértice con la posición en y más baja para ambas rectas. Luego, se obtiene la coordenada x de ambos vértices y se determina cuál arista está asociada al vértice con el menor valor en x. Esta arista se etiqueta como `left`, mientras que la otra arista se etiqueta como `right`. Con esto, se puede iniciar el cálculo de las intersecciones con las rectas.

Para cada iteración del algoritmo, se sustituye el valor actual de y_i en dos de las tres ecuaciones de rectas calculadas, las cuales dependen de la iteración en la que se encuentre el algoritmo. Durante las primeras iteraciones, se calculan las intersecciones con la horizontal utilizando las rectas `left` y `right`, siendo ambos puntos de intersección almacenados en un arreglo.

Una vez que el valor de y_i es igual o mayor a la posición en y del vértice medio del triángulo (aquel que no es el más alto ni el más bajo), se reemplaza una de las dos rectas utilizadas actualmente por la recta `right`. Esto se determina en función de la posición del vértice medio: si este se encuentra a la izquierda del punto más bajo, se reemplaza la recta `left` por la recta `bottom`. El caso es análogo para la recta `right`.

En la figura 2.14 se ejemplifica el cálculo de las intersecciones, donde las líneas rojas representan un intervalo de pixeles contenidos en el interior del triángulo. Como se puede notar, las rectas con las que se calculan los puntos de intersección varían dependiendo de la altura de la horizontal.

Una vez que se han calculado todas las intersecciones posibles con la horizontal, se obtiene un arreglo de pares de puntos, donde cada par representa un intervalo donde todos los pixeles se encuentran contenidos en el triángulo.

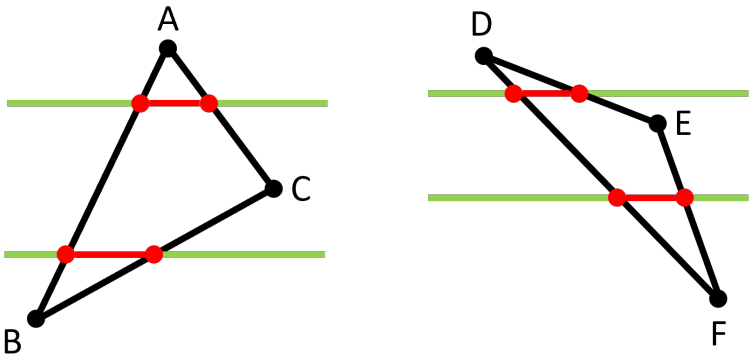

Figura 2.14: Ejemplo de cálculo de intersecciones con horizontales.

En resumen, el algoritmo utilizado consta de los siguientes pasos:

1. Se obtienen las ecuaciones que definen las tres rectas que conforman el triángulo.
2. Se etiquetan las rectas del triángulo como `left`, `right` y `bottom`, en función de su posición relativa
3. Para cada recta definida por $y = y_i$, donde y_i toma valores entre la coordenada y más alta y la coordenada y más baja del triángulo, se calculan las coordenadas de las intersecciones con dos de las tres rectas que conforman el triángulo. Estas coordenadas de intersección se agregan a un arreglo.

Los pasos descritos anteriormente no abordan los casos en los que el triángulo tiene dos puntos a la misma altura. Sin embargo, durante la implementación se establecieron las condiciones necesarias para manejar estos casos, llevando a cabo los cálculos sólo con las rectas `left` y `right`, sin considerar la existencia de una recta `bottom`. Estos casos borde son similares tanto si estos dos puntos se encuentran en la parte superior del triángulo como cuando se encuentran en la base de este.

2.6.3. Determinar si un polígono está contenido en otro

Para determinar si un polígono A está contenido dentro de un polígono B, basta con verificar si alguno de los vértices de A se encuentra dentro de B [6]. Esto es posible debido a las características de las geometrías generadas por ambos algoritmos de detección de bordes, los cuales impiden que existan intersecciones entre distintos polígonos. Por lo tanto, si un vértice de A se encuentra dentro de B, se puede concluir que todo el polígono A se encuentra dentro de B.

Para determinar si un punto P con coordenadas (x_0, y_0) se encuentra dentro de un polígono, se utilizará un algoritmo basado en ray-casting. Este método consiste en proyectar un rayo desde un punto en una dirección específica y buscar intersecciones entre esta proyección y los bordes del polígono. En este caso, se utiliza una proyección horizontal en dirección derecha.

El proceso para determinar las intersecciones de la proyección con los bordes del polígono se realiza en varios pasos. En primer lugar, se recorre el listado de las aristas que definen el polígono, conservando aquellas cuyo intervalo de coordenadas en el eje y contiene la coordenada y_0 del punto P. A continuación, se calcula la ecuación que define todas estas rectas y se obtienen los puntos de intersección para $y = y_0$, generando así una lista de puntos.

Luego, se seleccionan únicamente aquellos puntos que se encuentran a la derecha de P, es decir, aquellos cuya coordenada x es mayor a x_0. Una vez obtenidos todos los puntos que intersectan con la proyección, es posible determinar la posición del punto con respecto al polígono según la cantidad de intersecciones encontradas: si la cantidad de intersecciones es impar, se concluye que el punto está contenido en el polígono, mientras que si la cantidad de intersecciones es par, el punto se encuentra fuera de este.

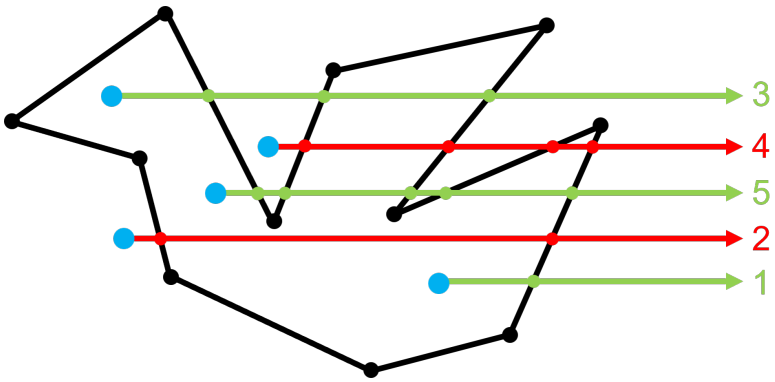

Figura 2.15: Ray-casting para determinar si un punto se encuentra contenido en un polígono.

En la figura 2.15 se ilustra este método y se muestran los resultados obtenidos para distintos puntos. Cabe destacar que, debido a las características de la geometría generada, nunca se dará el caso de que un punto cuya contención se desea determinar esté sobre las aristas del polígono. Por lo tanto, para la implementación de este algoritmo no se aborda este caso.

Esta información de contención es utilizada para determinar si un polígono dentro de una figura define un agujero dentro de esta, ya que es necesario tener esta información al momento de generar un archivo .poly que represente la geometría de manera adecuada. Se asume que un agujero siempre estará contenido en la figura principal, es decir, estará contenido dentro de otro polígono. Sin embargo, estas relaciones de contención pueden anidarse de forma infinita, lo que significa que es posible que un agujero contenga otro polígono que es parte de la figura principal, y así sucesivamente. Por lo tanto, es necesario nuevamente evaluar condiciones sobre cada polígono, con el fin de determinar su rol en la figura.

Para determinar si un polígono es parte de la figura o si define un agujero, se realiza una comparación entre cada par de polígonos utilizando el algoritmo mencionado anteriormente para determinar si uno está contenido dentro del otro. Para cada polígono se lleva una cuenta de todos los polígonos que lo contienen, y una vez obtenido el valor final, se establece que cada polígono contenido en un número impar de polígonos es un agujero, mientras que aquellos contenidos en un numero par son parte de la figura principal.

2.6.4. Punto aleatorio dentro de un polígono

Para obtener un punto aleatorio dentro de un polígono, se utiliza la propiedad de que todo polígono generado siempre tiene un ángulo interior menor a 180 grados. Dado esto, basta con recorrer la lista de vértices del polígono hasta encontrar un ángulo que cumpla con esta condición. El cálculo del ángulo se realiza utilizando los vectores formados por el vértice actual con su antecesor y el vértice actual con su sucesor en el polígono. Si el ángulo formado por estos dos vectores es menor a 180°, se concluye que el triángulo definido por estos tres puntos se encuentra dentro del polígono. Por lo tanto, se genera un punto en el centroide de este triángulo, el cual también estará contenido en el polígono.

Para obtener puntos óptimos que se alejen lo más posible de los bordes del polígono es posible implementar ciertas restricciones en la selección del ángulo. Por ejemplo, se puede dar mayor peso a los ángulos más cercanos a 60° o a aquellos formados por vectores de longitudes similares. Esto se debe a que los ángulos formados por vectores que no cumplen estas condiciones tienden a poseer centroides muy cercanos a los bordes del polígono.

Capítulo 3

Diseño e Implementación

3.1. Estructura general

La solución desarrollada consiste en un programa en Python que permite generar archivos .poly a partir de imágenes proporcionadas por el usuario. El programa fue desarrollado utilizando Python 3.11.2, e incorpora varias librerías para diferentes tareas. Numpy en su versión 1.24.2 se emplea para realizar cálculos complejos de manera eficiente. OpenCV en su versión 4.7.0 se utiliza para llevar a cabo el manejo de imágenes, incluyendo la inicialización de estas como arreglos, su procesamiento y la visualización de los bordes obtenidos. Además, se emplea la librería Imageio en su versión 2.27.0 para generar animaciones que ilustren el avance de los algoritmos utilizados.

El programa principal es capaz de utilizar dos métodos diferentes para la detección de bordes y generación de geometrías, donde cada método tiene fortalezas y debilidades distintas según el tipo de imagen, siendo el método utilizado a elección del usuario. El primer método de detección de bordes se basa en el algoritmo de Canny, generando PSLGs a partir de las imágenes entregadas por este algoritmo. El segundo método se basa en el trabajo de Kai Lawonn y Tobias Günther, utilizando mallas de triángulos que aproximan el color de la imagen de referencia. Estas mallas se refinan iterativamente para reducir el error de aproximación, proceso tras el cual se extraen aquellos componentes que representan el borde de la imagen.

En cuanto al alcance de la solución desarrollada, se requiere que las imágenes entregadas cuenten un fondo de color blanco, de forma tal que sea posible distinguir la figura del fondo, y así poder determinar de forma automática las áreas a conservar y eliminar.

A continuación se describirán en detalle ambos métodos y los aspectos de su implementación:

3.2. Método basado en el algoritmo de Canny

3.2.1. Diseño

3.2.1.1. Generación de aristas

Para obtener un listado de aristas a partir de los pixeles que indican bordes en las imágenes generadas por el algoritmo de Canny, se desarrolló un método específico. Una implementación directa de este tipo de algoritmo podría consistir en recorrer el arreglo de pixeles en la imagen y crear conexiones entre pixeles de borde adyacentes. Sin embargo, una implementación de este tipo podría presentar dos problemas:

1. **Generación de aristas duplicadas:** Cada arista sería creada dos veces, una vez en cada dirección. Esto se debe a que para cada pixel de borde donde se genere una arista saliente, también habría una arista entrante definida por los mismos puntos.
2. **Formación de triángulos en bordes:** Los bordes entregados por el algoritmo de Canny pueden presentar diagonales de uno o dos pixeles de ancho. En el segundo caso, es posible que tres pixeles se encuentren en una formación similar a una "L" (en cualquiera de sus posibles rotaciones), lo que resultaría en que cada pixel genere una conexión con los dos pixeles vecinos. Esto daría lugar a la formación de un triángulo, lo cual no reflejaría adecuadamente la topología de la figura, ya que se generaría un polígono en una zona de la imagen donde sólo debería haber vértices consecutivos unidos por aristas.

Para evitar estos problemas, se establecen las siguientes condiciones para la generación de aristas:

- Si el pixel a la derecha del pixel actual es blanco, se genera la arista de forma inmediata.
- Si el pixel debajo del pixel actual es blanco, se genera la arista de forma inmediata.
- Si el pixel ubicado en la esquina inferior derecha del pixel actual es blanco, se genera una nueva arista solo si los pixeles debajo y a la derecha de este no son blancos (es decir, se creará una arista diagonal solo si no existen aristas verticales u horizontales).
- Si el pixel ubicado en la esquina inferior izquierda del pixel actual es blanco, se genera una nueva arista solo si los pixeles de abajo y de la izquierda no son blancos

De esta manera, se priorizan las conexiones horizontales y verticales por sobre las diagonales, donde estas últimas solo se realizan si no existe otro tipo de conexión posible. Además, para cada pixel sólo es necesario consultar 4 pixeles vecinos, en lugar de los 8 pixeles inmediatamente adyacentes a este. En la figura 3.1 se ilustra el funcionamiento de este algoritmo, donde las flechas rojas representan las conexiones que finalmente se llevarán a cabo.

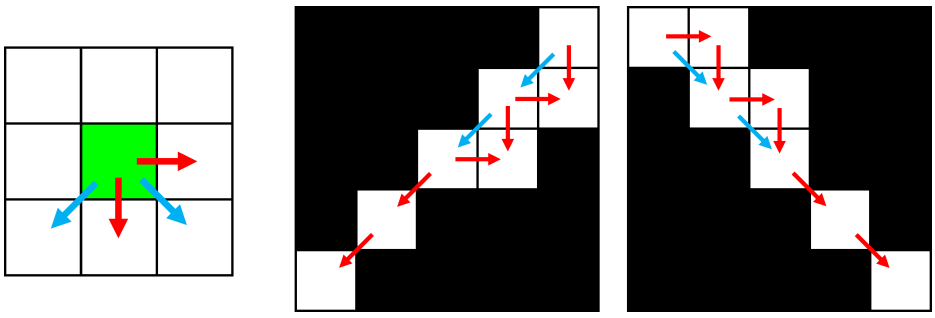

Figura 3.1: Ejemplo de funcionamiento del algoritmo de vectorización.

3.2.1.2. Reducción de cantidad de aristas

El uso de un algoritmo como el anteriormente descrito resultará en un listado de aristas muy extenso, y donde cada arista tendrá una longitud muy reducida, ya que cada vértice representaría un pixel en la imagen procesada con Canny. Dado que estas aristas serán utilizadas para crear mallas de polígonos, es necesario llevar a cabo una reducción del número de vértices para generar aristas de una mayor longitud. Para lograr esto, se diseñaron 3 algoritmos distintos:

Eliminación de vértices a intervalos constantes: Esta solución se enfoca en generar aristas con longitudes lo más uniformes posible, lo cual es deseable para la malla de polígonos que se generará posteriormente. En este enfoque, se recorre un listado de vértices que define los bordes en una imagen, conservando vértices solo cada ciertos intervalos. Por ejemplo, si se busca generar aristas con una longitud cercana a 10 pixeles, se recorre la lista de vértices, se conserva el primer vértice, se eliminan los 9 vértices siguientes y se repite el proceso hasta recorrer todo el contorno.

Este método produce caminos con aristas de longitud bastante uniforme. Sin embargo, la longitud de las aristas no será exactamente igual al valor del parámetro ingresado, debido a la existencia de aristas diagonales que tienen una longitud mayor a 1 pixel. Además, este algoritmo puede entregar resultados donde se pierdan detalles finos en la imagen, especialmente en curvas con ángulos muy cerrados.

Eliminación de vértices a intervalos variables: Para contrarrestar las desventajas de la primera solución, se propone un nuevo algoritmo que conserva los detalles finos de la imagen. Esto se logra mediante la evaluación de una condición de distancia, la que se explicará a continuación:

El proceso de reducción del número de vértices comienza seleccionando un vértice como punto de inicio, al cual se denominará v_0. A continuación, se recorren los demás vértices de la lista en orden, y cada vértice recorrido se almacena en una lista llamada `to_delete`. Para cada vértice v_i recorrido, se genera una arista candidata con v_0 como punto de inicio y v_i como punto final. Cada vez que se genera una arista candidata, se calcula la distancia mínima entre esta y cada vértice en la lista `to_delete`. Si ninguna de las distancias calculadas supera un valor previamente establecido por el usuario, se continúa con el proceso. Sin embargo, si alguna de las distancias calculadas supera este valor, se descarta el vértice actual y se toma el vértice candidato recorrido anteriormente v_{i-1} como punto final, generando una arista desde

v_0 a v_{i-1}. Luego, se reinician los valores de las variables y se toma v_{i-1} como nuevo punto de inicio. Este proceso se repite hasta recorrer todos los vértices en el contorno.

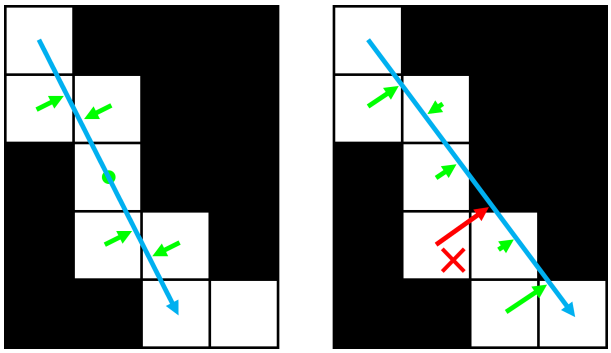

Figura 3.2: Cálculo de distancia a la arista candidata.

La figura 3.2 ilustra el funcionamiento de este algoritmo. Las flechas verdes representan los vértices que cumplen con la condición de distancia con respecto a la arista azul que se busca generar. Sin embargo, en la segunda imagen se rechaza la arista candidata de la segunda imagen es rechazada, ya que la condición de distancia no se cumple para uno de los vértices, como se muestra en color rojo.

Una solución de este tipo proporciona resultados más precisos que el algoritmo anterior, preservando los detalles más finos y generando una vectorización que se asemeja más a la imagen original. Sin embargo, debido a la inexistencia de una cota superior para la longitud de las aristas generadas, es posible que exista una gran variación en su longitud, especialmente si la figura de referencia contiene tanto detalles finos como líneas rectas prolongadas. En el primer caso, se generan aristas para cada punto de inflexión encontrado en la imagen, mientras que en el segundo caso se elimina una gran cantidad de vértices debido a la mínima variación en el ángulo de las aristas generadas.

Esta amplia variabilidad en las longitudes de las aristas puede ser no deseable para la generación de mallas de polígonos, por lo que es necesario desarrollar un algoritmo capaz de regularizar esta métrica y, al mismo tiempo, conservar un grado adecuado de detalle de la imagen de referencia.

Algoritmo híbrido de eliminación de vértices: Con el objetivo de aprovechar las fortalezas de los dos algoritmos previamente desarrollados, se diseñó un enfoque híbrido que combina el algoritmo de eliminación a intervalos fijos y el de intervalos variables. Se utilizó el algoritmo de eliminación de vértices a intervalos variables como base, lo cual permite capturar los detalles finos de la imagen. Además, se incorporó una reducción de vértices a intervalos constantes en casos donde una arista excede una cierta longitud, preservando así vértices intermedios en aristas largas. De esta manera, se obtiene un algoritmo que conserva los detalles finos de la imagen y establece un límite superior para la longitud de las aristas generadas, reduciendo así la variabilidad en longitudes y posibilitando el uso de las geometrías resultantes para la generación de mallas de polígonos.

3.2.2. Implementación

La implementación del método basado en el algoritmo de Canny se realizó utilizando programación orientada a objetos, creándose 3 clases que representan los elementos en los que se opera durante el procesamiento de la imagen. En la figura 3.3 se ilustra el diagrama de las clases utilizadas para este método.

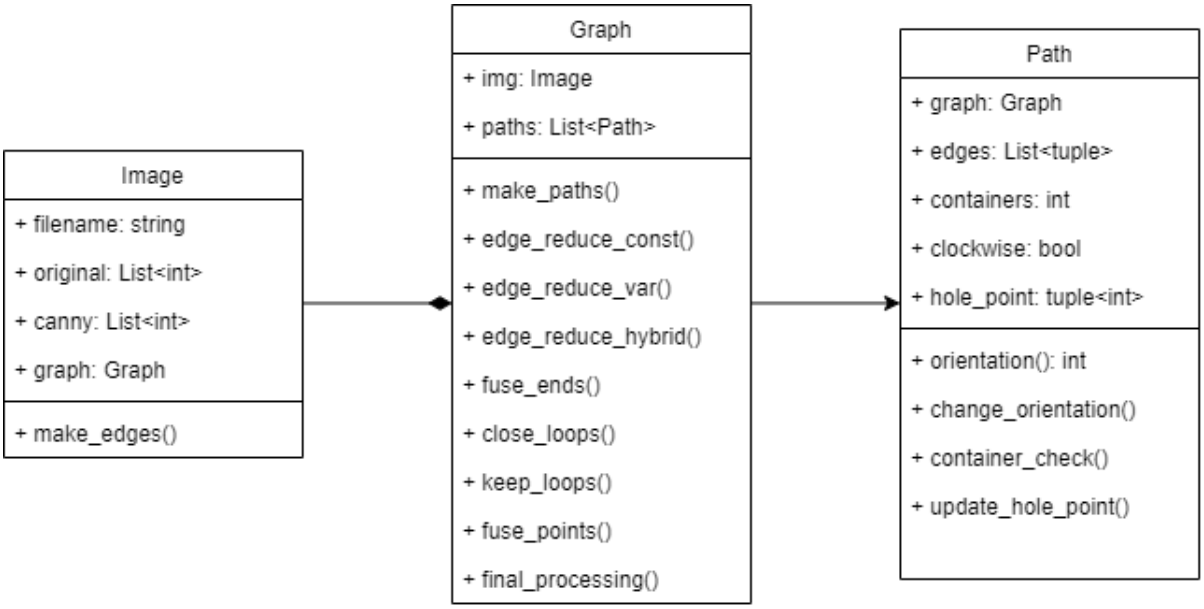

Figura 3.3: Diagrama de clases para el método basado en el algoritmo de Canny.

3.2.2.1. Clase Image

La primera clase utilizada es la clase `Image`. Esta clase se encarga de inicializar la imagen en la cual se desean detectar los bordes, y asocia a esta todos los elementos relacionados con la vectorización generada

La inicialización de la imagen implica generar un arreglo que la represente utilizando la librería OpenCV. Junto con esto, también se genera una copia de la imagen transformada a una representación en blanco y negro. Es importante destacar que esta transformación no equivale a una imagen en escala de grises. En la imagen transformada, cada pixel puede tener un valor de 0 o 255, efectivamente limitando los colores a solo blanco o negro, sin tonos intermedios.

Para lograr esto, se utiliza la función `threshold` de OpenCV. Esta función asigna el color negro a todos los pixeles cuyo valor sea menor o igual a 254. De esta manera, los pixeles que no son blancos en la imagen original se convierten en negro en la nueva imagen. El umbral del valor puede ser ajustado mediante un parámetro ingresado por el usuario al utilizar la herramienta. El resultado de aplicar esta función sobre una imagen se ilustra en la figura 3.4

Figura 3.4: Resultado de aplicar la función de thresholding en una imagen. Imagen original de dominio público.

Con este procesamiento, se considera que todos los pixeles no blancos son parte de la figura en la cual se desean detectar los bordes. A partir de esta imagen procesada se realiza la detección de bordes utilizando el algoritmo de Canny, siendo el resultado de esta detección almacenado en uno de los campos del objeto Image.

En resumen, un objeto de la clase `Image` contiene los siguientes campos:

- **filename:** Almacena el nombre del archivo como una cadena de texto.
- **original:** Representa la imagen original como un objeto de OpenCV. Esta imagen no se modifica y es utilizada como referencia.
- **canny:** Almacena la imagen resultante después de aplicar el algoritmo de Canny sobre la imagen transformada a blanco y negro.
- **graph:** Contiene una referencia al objeto `Graph` asociado a la imagen, el cual se encarga de la generación y el procesamiento de los bordes en formato vectorial.

Con respecto a los métodos presentes en la clase, se incluyen los getters y setters correspondientes para cada campo, junto con métodos a modo de wrapper para llamar de manera más directa a diferentes métodos de la clase `Graph`. Además, se proporcionan métodos para mostrar las imágenes finales en pantalla, dibujando los bordes detectados sobre la imagen original.

Una característica importante de los objetos esta clase es que siempre deben estar asociados a un objeto de la clase `Graph`, la cual será descrita en detalle más adelante. Esta asociación se establece mediante el método `make_edges()`, presente en la clase `Image`. Dicho método crea un objeto de la clase Graph y lo asigna al campo del mismo nombre en el objeto `Image`.

3.2.2.2. Clase Graph

La clase `Graph` contiene toda la información relacionada a la vectorización de la imagen y almacena la lista de caminos obtenidos tras el procesamiento de la imagen entregada por el algoritmo de Canny.

Esta clase cuenta con dos campos: el primero es el campo `img`, que almacena una referencia al objeto `Image` asociado, y el segundo es el campo `path`, que guarda una lista de caminos generados, donde cada camino es un objeto de la clase `Path`.

El primer método utilizado para la generación de los bordes es el método `make_edges()`. Este método recorre todos los pixeles de la imagen con los bordes detectados por el algoritmo de Canny, y genera a partir de esto una lista de aristas utilizando el algoritmo de generación de aristas descrito anteriormente en la sección 3.2.1.1.

Sin embargo, la lista de aristas generada por el método `make_edges()` consiste en múltiples aristas de longitud muy reducida que no están conectadas entre sí, lo que significa que no se dispone de información sobre qué aristas son adyacentes. Para abordar esto, se utiliza el método `make_paths()`, el cual recorre esta lista de aristas y conecta aquellas que comparten puntos de inicio o de fin, formando así caminos continuos.

El método `make_paths()` funciona de la siguiente manera: se inicializa un arreglo vacío que representa un camino de aristas. Se agrega una arista inicial al camino, a partir de la cual se conectarán las aristas adyacentes. Luego, se recorre el listado de aristas y se agregan aquellas cuyos puntos coincidan con los puntos de inicio o fin del camino actual. Una vez que no haya más coincidencias y no sea posible agregar más aristas al camino actual, se da inicio a la creación de un nuevo camino. Este proceso se repite hasta que todas las aristas pertenezcan a un camino.

Puede darse el caso de que una arista tenga múltiples aristas adyacentes. En estas situaciones, se prioriza la arista que forme un ángulo lo más cercano posible a 180 grados con la arista anterior, con el objetivo de dar mayor continuidad a los caminos generados. Dado que las aristas en la lista sólo pueden tener ángulos que sean múltiplos de 45 grados (ya que se originan a partir de conexiones entre pixeles), se consideran las últimas 5 aristas del camino actual para calcular el ángulo con la arista candidata.

3.2.2.3. Clase Path

La clase `Path` representa los caminos formados a partir de las aristas encontradas. Un camino se define como una secuencia de aristas en la que el punto final de una arista es el punto de inicio de la siguiente. Estos caminos pueden ser abiertos o formar polígonos cerrados, siendo solo estos últimos conservados en la vectorización final. Por otra parte, los caminos que no son cerrados son eliminados durante el procesamiento realizado al listado de caminos.

Un objeto de la clase `Path` tiene los siguientes campos:

- **graph:** Almacena una referencia al objeto `Graph` asociado.
- **edges:** Contiene un listado de aristas definidas por su punto de inicio y punto final.
- **containers:** Almacena un entero que representa la cantidad de polígonos que contienen al camino.
- **clockwise:** Campo booleano con valor True si el polígono está definido en el sentido de las agujas del reloj.

- **hole_point:** En caso de que el polígono defina un agujero, este campo almacena un punto dentro de él que se utiliza para la generación de agujeros. Si no es un agujero, el valor almacenado es nulo.

Es importante mencionar que los campos `containers`, `clockwise` y `hole_point` conservan sus valores por defecto durante la mayor parte del procesamiento y no son modificados hasta que todos los caminos abiertos sean eliminados. Una vez que solo quedan caminos cerrados (polígonos), es posible actualizar sus valores mediante el uso de los siguientes métodos:

- **orientation():** Determina la orientación de un polígono utilizando el algoritmo descrito en la sección 2.6.1.
- **change_orientation():** Se utiliza para alternar entre un polígono definido en sentido horario o antihorario.
- **container_check():** Recibe otro `Path` como argumento y devuelve True si el camino actual está contenido en el otro, utilizando el algoritmo descrito anteriormente en la sección 2.6.3.
- **update_hole_point():** Calcula un punto aleatorio dentro de un polígono utilizando el algoritmo descrito en la sección 2.6.4. Se llevan a cabo todas las verificaciones previas para asegurarse de que el polígono esté definido en sentido antihorario.

Además de los métodos previamente descritos, también se incluyen los getters y setters correspondientes para cada campo, así como todos los métodos utilizados para reducir la cantidad de aristas en un camino, aplicando los 3 diferentes algoritmos descritos en la sección 3.2.1.

Finalmente, se dispone de varios métodos para el procesamiento final de los caminos, incluyendo la generación de caminos cerrados y la fusión de vértices cercanos.

3.2.2.4. Procesamiento de caminos

Las tres clases mencionadas anteriormente son utilizadas para llevar a cabo el procesamiento general de la imagen mediante el programa contenido en `border_canny.py`. Este proceso se realiza a través de los siguientes pasos:

En primer lugar, se crea una nueva instancia de la clase `Image`, al cual se asocia inmediatamente un objeto de la clase `Graph`. Se realiza la detección de caminos y se genera un listado de objetos de la clase `Path` que están contenidos en el objeto `Graph` previamente inicializado. Una vez obtenida la lista de aristas sin reducir, se procede a realizar las etapas de procesamiento en el siguiente orden:

El primer paso consiste en la fusión de caminos. Esto es necesario debido a que puede haber discontinuidades en los bordes detectados por el algoritmo de Canny, lo que se traduce a caminos largos divididos en subcaminos de menor longitud.

La fusión de caminos se lleva a cabo recorriendo la lista de caminos generados y buscando otros caminos en la lista cuyos puntos de inicio o fin estén a una distancia menor que un umbral establecido a los puntos de inicio o fin del camino actual. De ser necesario, se ajusta la orientación del camino para que coincida con el camino objetivo, y luego estos son unidos modificando sus puntos de inicio y fin para que sean iguales al promedio de los dos puntos a conectar.

El segundo paso es la generación de ciclos cerrados, que servirán como base para la posterior creación de mallas. Esto se realiza simplemente recorriendo la lista de caminos y buscando aquellos cuyos puntos de inicio y fin estén a una distancia menor que un valor especificado por el usuario. De cumplirse esta condición, se ajustan los puntos de inicio y fin para que sean iguales al promedio de estos dos puntos.

Después de la generación de ciclos cerrados, se conservan solo aquellos caminos cuyo punto de inicio sea igual al punto final, descartando aquellos que no cumplan con esta condición. Este paso se realiza por separado de la generación de caminos, por lo que es posible prescindir de este y así mantener aquellos caminos que no formen ciclos, en caso de que se desee que estos formen parte de la malla final.

Por último, con el objetivo de eliminar posibles irregularidades causadas durante los pasos anteriores, se recorren todos los caminos para reducir la proximidad entre vértices que se encuentren demasiado cerca, según un parámetro establecido por el usuario. Esto se logra fusionando los vértices, eliminando uno de ellos y desplazando el restante al punto medio entre los dos vértices fusionados.

Por lo tanto, el procesamiento de los caminos se puede representar mediante el siguiente pseudocódigo:

Algoritmo 1 Algoritmo basado en Canny

```
function CANNY_MAIN()(caminos, umbral)
    fusionar_caminos()
    ciclos_cerrados()
    conservar_ciclos()
    fusionar_puntos_cercanos()
    Retornar caminos generados
end function
```

3.2.2.5. Procesamiento final

El procesamiento final consiste en modificar la estructura de los caminos generados para que se ajusten al formato utilizado por el script encargado generar archivos .poly. El primer paso es determinar si un camino en particular representa un área que será un agujero en la imagen final.

Para lograr esto, se calculan las relaciones de contención entre los diferentes polígonos utilizando la función `container_check()` en cada par de caminos. Esto proporciona información sobre la cantidad de polígonos que contienen a cada polígono utilizando el algoritmo

descrito en la sección 2.6.3. Con esta información, es posible determinar si un polígono define o no un agujero dentro de la figura, y en caso afirmativo, se calcula un punto interior utilizando la función `update_hole_point()`, la cual genera un punto dentro del agujero detectado mediante el algoritmo descrito en la sección 2.6.4.

Una vez que se tiene toda la información necesaria acerca de la figura, se retorna cada camino como una tupla. El primer elemento de la tupla es una lista de puntos que define el polígono, y el segundo elemento es un punto interior en caso de que el polígono defina un agujero, o un valor nulo en caso contrario. Esta lista de caminos es entregada al archivo `make_poly.py`, el cual transformará esta lista de puntos en un archivo .poly.

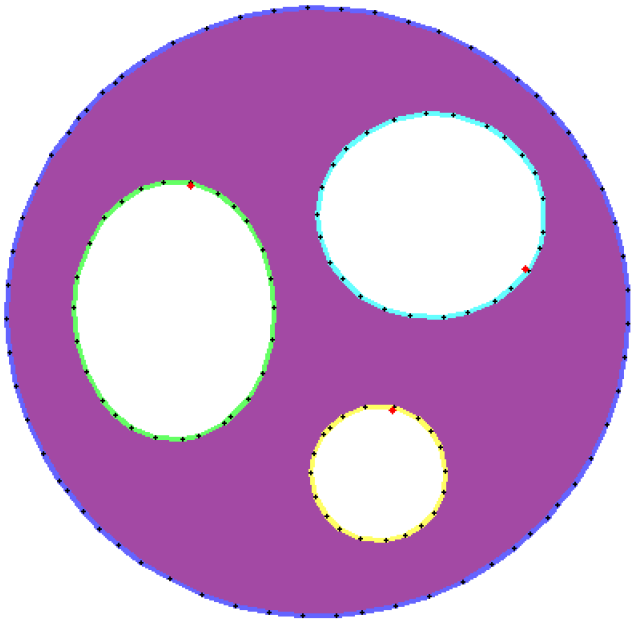

Figura 3.5: Resultado de aplicar el método basado en Canny en una figura simple.

En la figura 3.5 se muestra el resultado de aplicar el método basado en Canny sobre una figura simple. Los caminos generados se superponen a la figura de referencia, y cada camino es representado con un color diferente. También es posible observar los puntos interiores detectados, donde cada uno se encuentra marcado con un punto rojo.

3.3. Método basado en triangulaciones

3.3.1. Diseño

La solución desarrollada se basa en una versión simplificada del algoritmo de triangulación de imágenes desarrollada por Kai Lawonn y Tobias Günther en su artículo "Stylized Image Triangulation". En la sección 2.4 se resumió el funcionamiento general de dicho algoritmo. Sin embargo, se realizaron modificaciones con el objetivo principal de enfocar la funcionalidad de la herramienta a la detección de bordes, dejando de lado la captura de los detalles internos

de la imagen.

3.3.1.1. Generación de una malla inicial

El algoritmo original ofrece 3 métodos distintos para la generación de una malla de triángulos inicial.

- **Grilla regular:** Este método implica crear una malla en forma de grilla, donde cada celda se divide mediante una diagonal, generando así una triangulación donde cada cara tiene la misma área y las aristas tienen longitudes regulares.
- **Distribución de probabilidad:** Este método consiste en guiar la posición inicial de los vértices, ya manualmente mediante input del usuario, o bien mediante mapas de saliencia, es decir, imágenes que destaquen ciertas regiones importantes de la imagen. Esto resulta en una malla donde los vértices están más densamente distribuidos en las áreas más importantes de la imagen, lo que puede dar lugar a triángulos y aristas de tamaños muy variables.
- **Refinamiento adaptativo:** El tercer método aprovecha las estrategias de refinamiento de mallas disponibles en la herramienta. Este comienza con una triangulación muy gruesa (dos triángulos que cubren toda el área de la imagen) y mejora la aproximación a la imagen mediante la inserción de vértices en áreas con un alto error de aproximación.

Para este trabajo, se eligió usar el método de la grilla inicial, ya que este permite comenzar con un conjunto de aristas de longitudes similares, evitando disparidades significativas en las longitudes después de haber transcurrido todas las iteraciones del proceso. Sin embargo, se decidió realizar una modificación en esta malla inicial con el objetivo de mejorar la aproximación a la imagen desde el inicio del proceso.

La modificación consistió en ajustar los ángulos de las diagonales para de esta forma obtener las orientaciones que minimizaran el error de aproximación dentro de cada triángulo. Esto se hizo para evitar tener que reorientar los triángulos a lo largo del proceso o insertar elementos a la malla en posiciones subóptimas. Con este fin, antes de generar la malla inicial se calcula del error de aproximación para los triángulos generados con ambas orientaciones de la diagonal, y se conserva aquella orientación que minimice este valor.

En la figura 3.6 se muestra el resultado obtenido al superponer las mallas iniciales sobre la imagen de referencia. En la visualización, cada triángulo está coloreado de acuerdo al color promedio de los pixeles contenidos en él. A la izquierda se encuentra la imagen utilizada, mientras que en el centro se muestra la malla inicial implementada de la misma forma que en el trabajo usado de referencia. Por último, a la derecha se muestra la imagen con una nueva malla inicial, en la cual las diagonales que dividen cada celda varían su dirección con el objetivo de obtener la mejor aproximación posible.

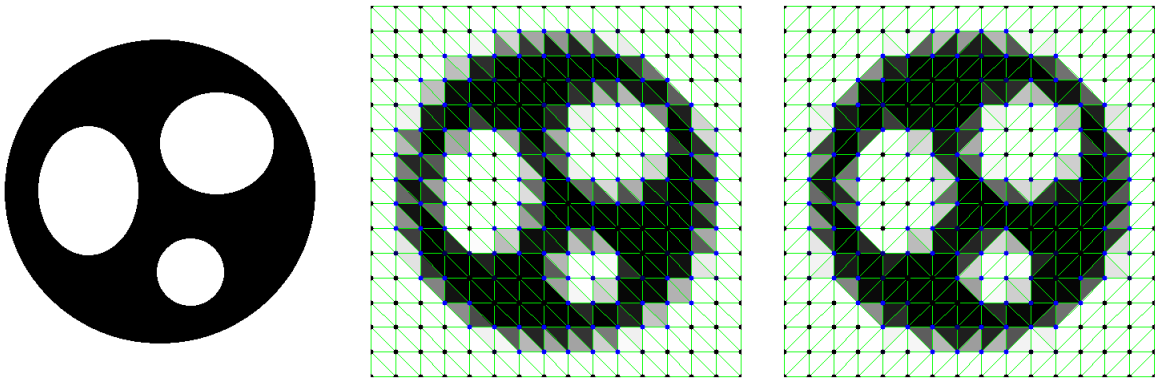

Figura 3.6: Aproximacion resultante tras la generacion de distintas mallas inciales.

3.3.1.2. Cálculo del error de aproximación en un triángulo

El cálculo del error de aproximación en un triángulo es una etapa crucial en este proceso, ya que este valor determinará tanto el desplazamiento de los vértices como la ejecución de distintos pasos para el refinamiento de la malla. La suma de los errores de aproximación de todos los triángulos de la malla se conoce como el error de aproximación de la imagen, valor que se busca minimizar.

El primer paso para calcular el error de aproximación en un triángulo es determinar el color promedio de los pixeles contenidos en él, lo cual implica conocer las coordenadas de estos pixeles. Esto último se logra utilizando el algoritmo descrito en la sección 2.6.2, obteniendo así las coordenadas de cada pixel ubicado en el interior de un triángulo determinado. Teniendo las coordenadas de los pixeles y sus colores correspondientes, se calcula el color promedio que se utilizará para rellenar el triángulo y lograr la aproximación de la imagen.

El siguiente paso en el cálculo del error de aproximación consiste en determinar la diferencia entre el color de cada pixel en el triángulo con respecto al color promedio calculado previamente. Para esto, se recorre nuevamente el listado de pixeles del triángulo y se aplica la fórmula de la distancia euclideana para cada pixel. Estas distancias se utilizan para calcular la distancia promedio, que representa el error de aproximación.

Según la fórmula anteriormente descrita, si todos los pixeles en un triángulo tienen el mismo color, el color promedio será igual al color de cada pixel, lo que resultará en una distancia de cada pixel al promedio igual a 0 y, por lo tanto, un error de aproximación de 0. En consecuencia, se busca minimizar el error de aproximación para que de esta forma cada triángulo contenga pixeles con colores lo más similares posible.

3.3.1.3. Cálculo del error de aproximación en un vértice

El error de aproximación de una imagen se puede obtener sumando los errores de aproximación calculados sobre cada triángulo. Sin embargo, también es posible calcular este valor tomando en cuenta los vértices en lugar de los triángulos. Bajo este enfoque, el error de aproximación en un vértice se define como la suma de los errores de todos los triángulos ad-

yacentes a ese vértice, dividida en 3. De esta manera, la suma de los errores de aproximación en todos los vértices es igual a la suma de los errores en los triángulos.

Este método de cálculo implica que aquellos vértices con más triángulos adyacentes generalmente tendrán errores de aproximación más altos. Sin embargo, esto no representa un problema, ya que el error de aproximación sobre un vértice se compara solo consigo mismo.

Es posible reducir el error de aproximación en una imagen al disminuir el error de aproximación en los vértices de forma individual, desplazándolos en la dirección que minimice dicho valor. Dado que el desplazamiento de un vértice solo afecta a los triángulos adyacentes a él, sólo los vértices a 1 arista de distancia se ven afectados, ya que comparten triángulos adyacentes. Debido a la localidad de este impacto, es posible despreciar el efecto de este desplazamiento en los demás vértices, lo que permite realizar este procesamiento para todos los vértices en la malla de forma simultánea.

3.3.1.4. Desplazamiento de vértices

Una vez establecida una metodología para calcular el error de aproximación en un vértice, es posible determinar cómo los desplazamientos de este vértice afectan el error de aproximación y, por lo tanto, en el error de aproximación de la imagen en su totalidad.

Para reducir el error de aproximación en un vértice específico, es necesario determinar la dirección de desplazamiento que disminuirá el error de manera más significativa. Para lograr esto, se pueden realizar desplazamientos temporales del vértice en diferentes direcciones, calcular el error de aproximación con las nuevas coordenadas, y luego regresar el vértice a su posición original. Para esta implementación, se optó por realizar movimientos tentativos en cuatro direcciones: arriba, abajo, izquierda y derecha, cada uno a una distancia de un pixel.

Para cada desplazamiento tentativo, es necesario recalcular los errores de aproximación de cada triángulo adyacente. Esto implica recorrer nuevamente los pixeles interiores de los triángulos para determinar el color promedio y calcular el nuevo error de aproximación. Una vez que obtenido el error de aproximación para cada triángulo, se calcula el error sobre el vértice y se registra en una lista que contiene los errores obtenidos para cada dirección de desplazamiento. Luego, se selecciona la dirección de desplazamiento que resulte en la disminución más significativa del error sobre el vértice. Si ninguno de los desplazamientos conlleva una reducción del error del vértice, este conserva su posición original durante la iteración actual del algoritmo.

En la figura 3.7 se puede apreciar el proceso de desplazamiento de los vértices en una malla, donde los triángulos se encuentran coloreados con el color promedio de los pixeles contenidos en estos. Es posible notar que al transcurrir las iteraciones las celdas de la malla pierden su regularidad, adoptando distintas formas que se adaptan a la imagen de referencia. Además, también es posible notar la mejora en la calidad de aproximación al transcurrir las iteraciones, ya que la cantidad de triángulos grises disminuye, evidenciando que estos pasan a ser completamente negros o blancos a medida que avanza el algoritmo.

Cabe destacar que el desplazamiento de los vértices se realiza después de calcular las di-

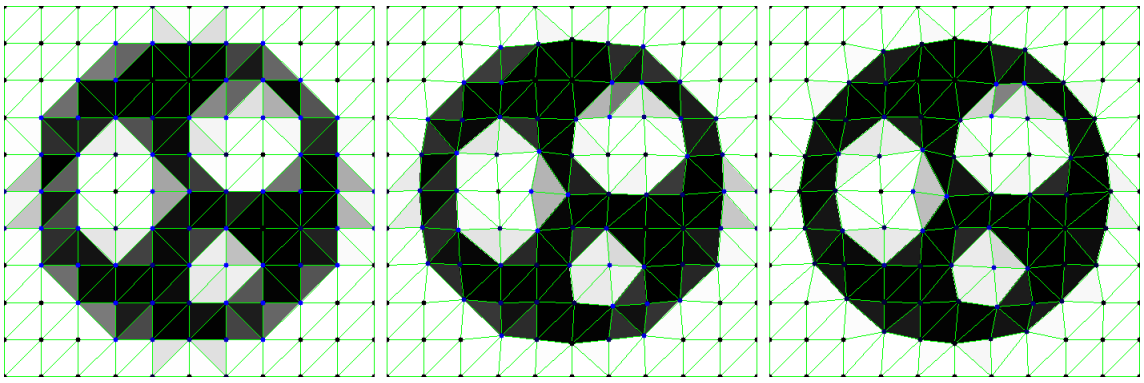

Figura 3.7: Malla inicial y malla resultante tras 5 y 10 iteraciones de desplazamiento de vértices.

recciones de desplazamiento de cada vértice. En otras palabras, para cada vértice se calcula una dirección de desplazamiento considerando el estado de la malla al inicio de la iteración. Una vez que se hayan calculado todas las direcciones, se lleva a cabo el desplazamiento de los vértices de manera simultánea. Sin embargo, este enfoque puede generar desplazamientos oscilantes, donde un vértice alterna entre dos posiciones durante varias iteraciones del algoritmo. Esto ocurre porque los desplazamientos tentativos de los vértices solo consideran la variación en si mismos, sin tener en cuenta los desplazamientos en los vértices que definen los triángulos adyacentes.

Para resolver este problema, que es especialmente prominente en las iteraciones finales del algoritmo, se decide realizar el desplazamiento de los vértices de forma secuencial en las últimas iteraciones. Esto significa que, para cada vértice, se calcula su dirección de desplazamiento y se lleva a cabo inmediatamente, de modo que el cálculo de la dirección para los siguientes vértices tome en cuenta el desplazamiento ya realizado. El orden de estos desplazamientos no es aleatorio, ya que se priorizan aquellos vértices con el mayor error de aproximación, recorriéndose la lista de vértices en orden descendente.

3.3.1.5. Refinamiento y optimización para mejorar la aproximación

Una vez que se ha calculado el error de aproximación en los triángulos y vértices, y se ha realizado el desplazamiento de estos últimos, es necesario llevar a cabo diferentes etapas de refinamiento y optimización de la malla para acelerar el proceso de aproximación.

En el trabajo "Stylized Image Triangulation", se describen varias etapas de refinamiento y optimización posteriores al desplazamiento de los vértices, como los *edge-flips*, *edge-collapses* e inserción de puntos. Sin embargo, no se especifica el orden en el que se deben realizar estas etapas, por lo que en la implementación actual fue necesario determinar este orden de manera experimental.

Después de realizar varias pruebas, se observó que las etapas de refinamiento y optimización de la malla se pueden realizar con dos objetivos: mejorar la aproximación de la triangulación sacrificando la calidad de la malla, o bien preservar la calidad de la malla sacrificando la aproximación.

Se decidió llevar a cabo estos pasos de modificación de la estructura de la malla en el orden descrito anteriormente, comenzando por aquellos pasos orientados a mejorar la calidad de la aproximación. De esta manera, una vez que se han realizado estos pasos, se procede a realizar los pasos orientados a restablecer la calidad de la malla. Esto garantiza que al final de cada iteración se obtendrá una triangulación de buena calidad sobre la cual se podrán llevar a cabo posteriores desplazamientos de vértices, reduciendo la posibilidad de tener triángulos con área nula o negativa.

Para mejorar la calidad de la aproximación, se emplean dos tipos de operaciones: *edge-flips* e inserción de puntos.

En el caso de los *edge-flips*, se recorren todas las aristas de la malla y se generan *edge-flips* tentativos. Esto implica crear las aristas y triángulos temporales que se generarían si el *edge-flip* se llevara a cabo. Se calcula el error de aproximación para estos triángulos temporales y, si el error es menor al error de aproximación de los triángulos que se desea eliminar, se realiza el *edge-flip*. En caso contrario, se descartan los elementos temporales y se continúa recorriendo el listado de aristas. En la figura 3.8 se ilustra este caso de *edge-flip*.

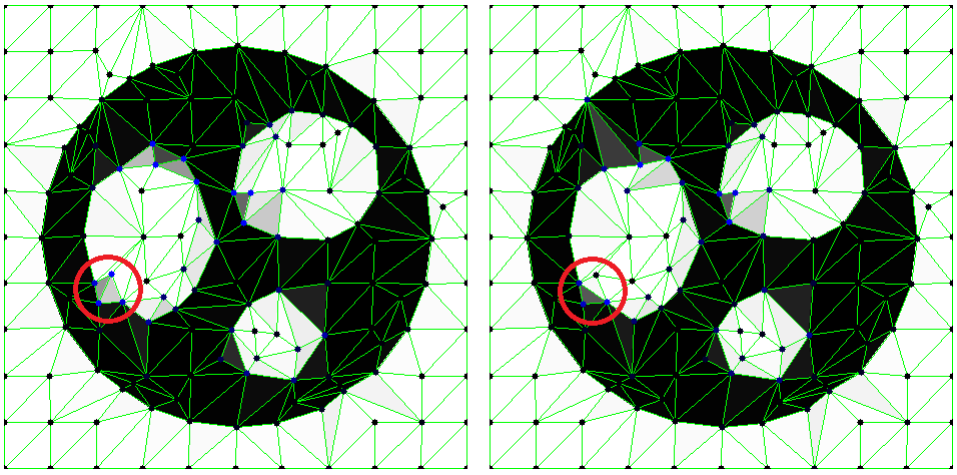

Figura 3.8: Ejemplo de *edge-flip* realizado para mejorar la aproximación.

Aunque este paso está orientado a mejorar la calidad de la aproximación, también se aplican restricciones para evitar la generación de elementos que empeoren la calidad de la triangulación. Se establecen restricciones tanto para la longitud de las aristas que se pretende crear como para el tamaño máximo de los ángulos opuestos a ellas, con el objetivo de evitar la formación de triángulos con ángulos muy obtusos.

Una vez realizados los *edge-flips*, se procede al paso de inserción de puntos en la malla. Esta inserción de puntos puede ser de los dos tipos descritos anteriormente: inserción en el punto medio de una arista o inserción en el punto medio de un triángulo.

Las condiciones para la inserción de puntos, ya sea en una arista o en un triángulo, siempre se evaluarán en función de los triángulos. La primera condición para realizar una inserción de puntos es cuando el error de aproximación en un triángulo supere un umbral establecido. La segunda condición es cuando el área de un triángulo supera un tamaño predefinido. Si alguna de estas condiciones se cumple, se realizará una inserción de puntos, y el elemento donde se

realizará la inserción (arista o triángulo) dependerá de las características del triángulo en el que se cumpla la condición evaluada:

- Si el triángulo es obtuso (tiene un ángulo mayor a 90 grados) o su ángulo más pequeño mide menos de 45 grados, la inserción del punto se realizará en el punto medio de la arista más larga del triángulo.
- En caso contrario, la inserción de puntos se llevará a cabo en el centroide del triángulo en cuestión.

Mediante estos criterios se busca evitar la inserción de puntos en los centroides de triángulos con áreas muy pequeñas, lo cual podría generar triángulos con ángulos aún más extremos. En la figura 3.9 se muestra el resultado de insertar puntos cuando se cumple la primera condición, generando un vértice nuevo en medio de una arista.

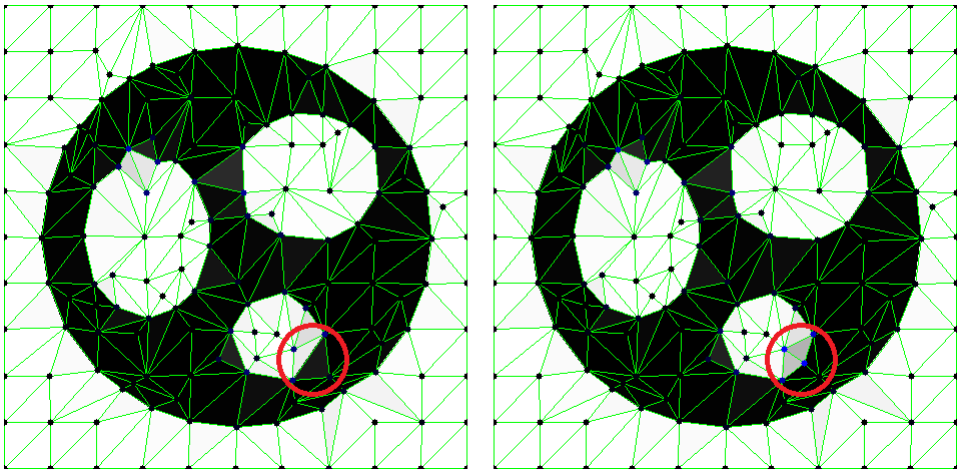

Figura 3.9: Ejemplo de inserción de puntos en medio de una arista.

Cabe destacar que las inserciones de puntos no se llevarán a cabo en cada iteración, sino que se optó por realizar este proceso cada 4 iteraciones. El objetivo de esto es evitar insertar puntos de forma excesiva, y de esta forma dar tiempo al algoritmo de lograr la mejor aproximación posible con los puntos que se encuentran actualmente disponibles antes de insertar nuevos elementos en la malla.

Existe un tercer caso de inserción de puntos que se realiza en las iteraciones finales del proceso. En este caso, en lugar de evaluar condiciones sobre los triángulos, se recorre la lista de vértices de la malla. Para cada vértice, se determina su dirección de desplazamiento en la última iteración. Si un vértice no se desplazó en la última iteración, y su error de aproximación supera un valor previamente definido, se realiza una inserción de puntos en el triángulo adyacente que tenga el mayor error de aproximación. Esta inserción provoca cambios en el conjunto de triángulos adyacentes al vértice, lo que modifica los errores de aproximación calculados para dicho vértice. El objetivo de este proceso es evitar que los vértices se queden estancados en mínimos locales y permitir su desplazamiento en direcciones que continúen disminuyendo el error de aproximación.

Una vez que se han realizado todos estos pasos, se obtiene una malla que aproxima la imagen de referencia de manera más precisa. Sin embargo, a pesar de lograr una buena aproximación, la malla resultante no cumple con las propiedades deseables de una triangulación de Delaunay. Por el contrario, los desplazamientos de los vértices y los pasos de refinamiento a menudo dan lugar a triángulos con ángulos no óptimos. Por lo tanto, es necesario llevar a cabo más pasos de refinamiento y optimización para preservar la integridad de la malla y regularizar los triángulos, no con el objetivo de generar una triangulación de Delaunay, pero si para acercarlo un poco a esto.

3.3.1.6. Simplificación y optimización para preservar la integridad de la malla

Los siguientes pasos tienen como objetivo la eliminación y sustitución de elementos de la malla, en concreto aquellos que puedan dar lugar a errores después de realizar el desplazamiento de vértices al comienzo de la siguiente iteración.

Uno de los problemas a abordar es la generación de triángulos con áreas nulas o negativas. Este problema puede ocurrir debido a dos factores: ángulos excesivamente obtusos y aristas de longitud muy reducida. En el primer caso, un triángulo puede tener un área nula en una iteración si un desplazamiento en el vértice adyacente a su ángulo más grande hace que dicho ángulo alcance los 180 grados, lo que resulta en que los tres vértices del triángulo sean colineales y, por lo tanto, el área sea nula. En el segundo caso, una arista puede reducir su longitud en iteraciones sucesivas, y cuando esta longitud se anula, también se anula el área de los dos triángulos adyacentes a esta arista.

Para evitar estos problemas causados por triángulos con dimensiones irregulares, se comienza realizando *edge-flips* con el objetivo de regularizar los ángulos de los triángulos, independientemente de si esto mejora o empeora el error de aproximación. Se utiliza un criterio similar al de las triangulaciones de Delaunay, evaluando la suma de los ángulos opuestos a una arista. Sin embargo, se busca evitar deshacer el refinamiento realizado en las iteraciones anteriores y no comprometer significativamente la aproximación de la imagen. Por lo tanto, se relaja la condición de Delaunay y se realizan *edge-flips* solo en los casos más extremos.

De este modo, se modifica la condición sobre la suma de los ángulos opuestos a la arista. En lugar de realizar un *edge-flip* cuando la suma supera los 180 grados, ahora se realiza cuando la suma supera los 240 grados. Esto permite la existencia de triángulos con ángulos más abiertos, pero aún se realiza el *edge-flip* si los ángulos exceden un valor determinado.

En la figura 3.10 se muestra un ejemplo de un *edge-flip* realizado entre dos iteraciones. Este *edge-flip* se llevó a cabo con el objetivo de preservar la integridad de la malla, transformando dos triángulos con ángulos obtusos en dos triángulos que se aproximan más a un triángulo equilátero.

Una particularidad de la implementación de los *edge-flips* en este trabajo es que estos no se llevan a cabo de forma recursiva. Esto significa que una vez que se modifica la posición de una arista, no se evalúa el impacto de este cambio en las aristas adyacentes. Esta decisión se tomó para evitar un aumento significativo en el tiempo de procesamiento requerido para el refinamiento de la malla. Además, como el refinamiento se realiza en varias iteraciones, los

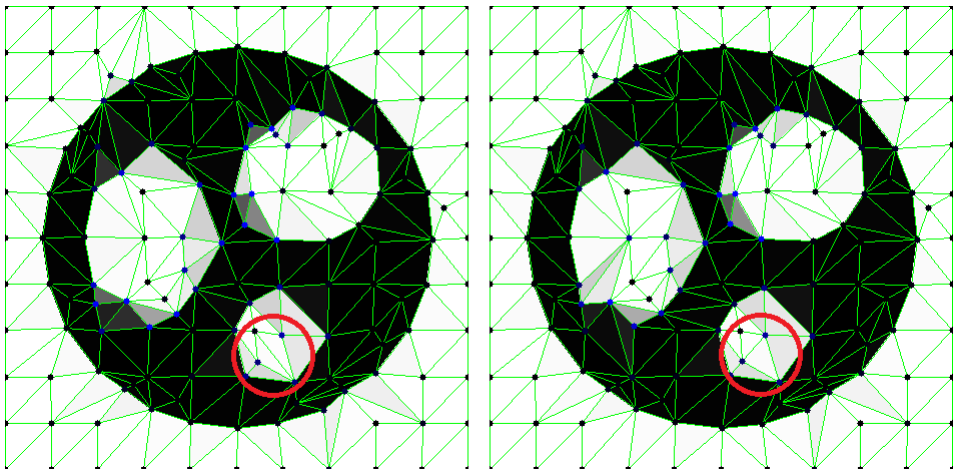

Figura 3.10: Ejemplo de *edge-flip* realizado para preservar la integridad de la malla.

edge-flips que se hubieran realizado de forma inmediata en una implementación recursiva se realizarán de todos modos, pero en la siguiente iteración.

El segundo paso llevado a cabo para preservar la integridad de la malla implica la realización de *half-edge-collapses*. Esto tiene como objetivo eliminar aristas y triángulos de dimensiones muy reducidas que hayan sido generados durante los pasos anteriores.

Un *half-edge-collapse* se realiza en dos casos diferentes, cada uno sobre distintos elementos de la malla. En el primer caso, se recorre el listado de aristas de la malla en busca de aquellas con una longitud menor a `min_e_len`, definido como el valor mínimo aceptado para las aristas en la malla. Para cada arista que cumpla con esta condición, se efectúa un *half-edge-collapse* en dirección a su punto final. Este caso de *half-edge-collapse* se ilustra en la figura 3.11.

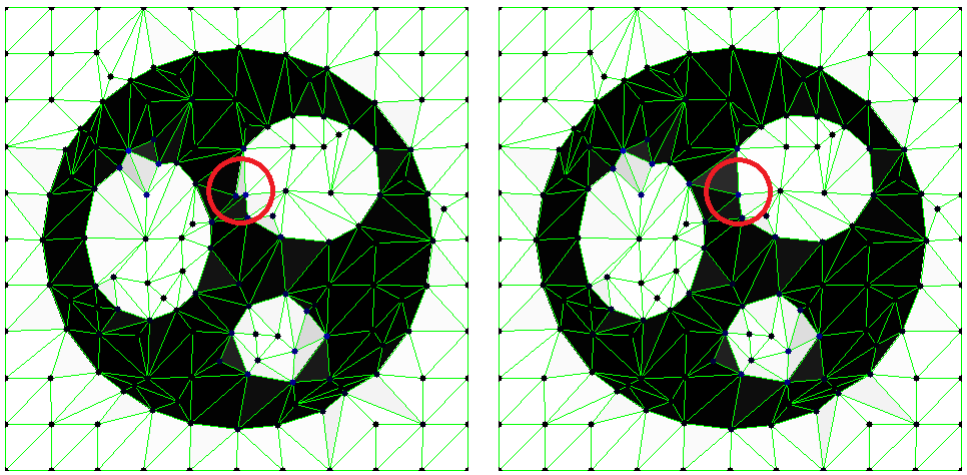

Figura 3.11: Ejemplo de *half-edge-collapse* realizado sobre una arista de longitud muy reducida.

En el segundo caso, se recorre la lista de triángulos de la malla en busca de aquellos triángulos cuya área sea menor a una quinta parte del área inicial de los triángulos de la malla. El objetivo de este paso es eliminar aquellos triángulos cuya área se haya reducido significativamente durante las iteraciones, siendo un valor de 1/5 obtenido experimentalmente

como un buen umbral para llevar a cabo esta eliminación. Dado que un *half-edge-collapse* debe llevarse a cabo sobre una arista, en caso de cumplirse la condición para el colapso, este se lleva a cabo en la arista más corta del triángulo correspondiente.

Como se mencionó anteriormente, durante la ejecución del *half-edge-collapse* pueden surgir situaciones de borde que resulten en triángulos con área negativa, situación que se desea evitar. Por lo tanto, se establecieron múltiples restricciones al momento de llevar a cabo este paso.

Una de estas restricciones es limitar la capacidad de realizar un *half-edge-collapse* en aristas que se encuentren en el borde de la malla, ya sea como aristas que definen un borde o aquellas con solo un vértice en el borde. Dado que las imágenes son preprocesadas para incluir un borde blanco alrededor del contenido de la figura que se desea aproximar, se puede asumir que los vértices en el borde tendran un desplazamiento mínimo, por lo que no será necesario llevar a cabo estas operaciones de eliminación de bordes.

Para todas las demás aristas, también se consideran los casos en los que un *half-edge-collapse* resultaría en triángulos de área nula. Para abordar este problema, se realiza una verificación basada en una condición de ángulos, la cual se explicará en detalle a continuación.

En el caso de un *half-edge-collapse*, todas las aristas que serán modificadas se encuentran contenidas dentro de un polígono que es la unión de todos los triángulos adyacentes al vértice que será desplazado. Las nuevas aristas generadas también estarán dentro de este polígono y tendrán al vértice de destino como uno de sus puntos. Por lo tanto, es necesario evitar que estas nuevas aristas intersecten los bordes del polígono mismo, lo cual solo ocurre si el polígono presenta concavidades. Luego, para determinar si habrá intersecciones entre las aristas generadas y el polígono, es necesario evaluar los ángulos interiores de este último.

Dadas las características de las triangulaciones generadas, se asume que las concavidades generadas no serán muy pronunciadas. Por lo tanto, se decidió evaluar únicamente dos ángulos específicos, los cuales son adyacentes al vértice de destino. Si alguno de estos dos ángulos es mayor a 180 grados, se concluye que esto resultaría en un triángulo con área nula o negativa. En tal caso, se realiza un *half-edge-collapse* en la dirección opuesta, y se lleva a cabo la misma evaluación.

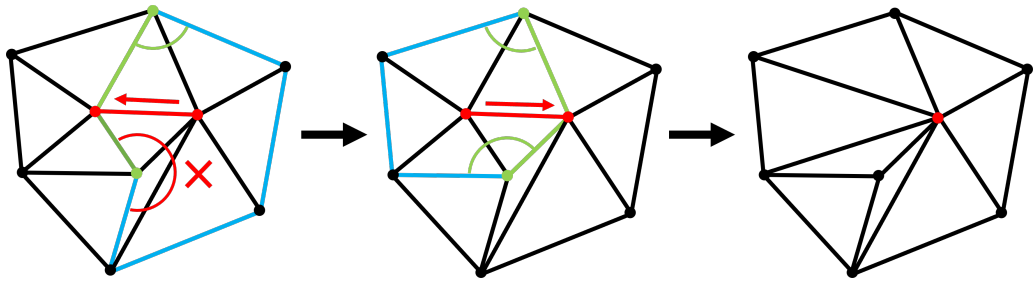

Figura 3.12: Caso en el que un *half-edge-collapse* se lleva a cabo en sentido contrario.

Como se puede observar en la figura 3.12, en el caso inicial se intenta realizar un *half-edge-collapse* moviendo el punto de la derecha hacia la izquierda, lo cual resultaría en un triángulo con área negativa. Sin embargo, antes de llevar a cabo esta operación, se realiza una revisión midiendo los dos ángulos inmediatamente adyacentes al vértice de destino. Se observa que

uno de estos ángulos es mayor a 180 grados, lo que impide la realización del *half-edge-collapse* en esta dirección. Luego de esta comprobación, se realiza la misma verificación considerando un *half-edge-collapse* en sentido contrario, y dado que ambos ángulos son menores a 180 grados, esta operación puede llevarse a cabo.

Una vez que se han completado todos los pasos del refinamiento, es posible proceder nuevamente al desplazamiento de los vértices. Gracias a la regularización de la malla durante el refinamiento, la probabilidad de generar triángulos con áreas nulas o negativas se reduce significativamente.

3.3.1.7. Detección de bordes

Para el proceso de detección de bordes, se realiza un último recorrido por el listado de aristas de la malla. Para cada arista, se calcula el color promedio de sus dos triángulos adyacentes. Utilizando estos colores promedio, se calcula la distancia euclidiana hacia el color blanco. Si esta distancia es menor que un umbral predefinido, se considera que el triángulo forma parte del fondo de la imagen. En caso contrario, se considera que pertenece a la figura. Luego, si una arista tiene su triángulo correspondiente etiquetado como fondo, y el triángulo de su *twin* está etiquetado como figura (o viceversa), se clasifica dicha arista como una arista de borde, la cual formará parte del PSLG generado.

Una vez realizado el etiquetado de las aristas, se procede a la generación de caminos, es decir, la búsqueda de una secuencia de aristas en la que el punto final de una arista coincida con el punto de inicio de la siguiente. Para lograr esto, se aprovecha la estructura de datos de la malla, lo que hace posible recorrer el listado de aristas y, cuando se encuentra una arista de borde, se recorren las aristas adyacentes a esta, permitiendo encontrar aristas de borde de forma consecutiva. El recorrido continúa hasta que el punto de inicio y el punto final del camino de aristas sean iguales, y luego se continúa con las aristas no visitadas.

Para determinar si un camino define un agujero, se puede aprovechar la información de orientación de las aristas en la estructura de datos *half-edge*. Esto permite aplicar una condición más estricta para encontrar bordes: en lugar de etiquetar una arista como borde solo en base a si su triángulo y el de su *twin* estén etiquetados de manera diferente, se considera la posición relativa de los triángulos. Es decir, una arista se etiqueta como borde solo si el triángulo asociado a este arista pertenece a la figura, y el triangulo asociado a su *twin* pertenece al fondo. Como resultado, las aristas de borde tendrán un *twin* que no se considerará como borde, y los caminos generados estarán siempre definidos en sentido antihorario para representar la figura y en sentido horario para definir agujeros.

Para generar puntos dentro de los agujeros, basta con seleccionar el centroide de uno de los triángulos adyacentes a un camino que define un agujero, estando este punto siempre contenido en el agujero en cuestión. De esta manera, se obtiene la lista de vértices, aristas y puntos interiores necesarios para definir un archivo .poly, los cuales serán proporcionados al script `make_poly.py` para su posterior generación.

3.3.1.8. Implementación basada en imágenes en escala de grises

Una desventaja del algoritmo descrito anteriormente es que la información de color puede no ser relevante para detectar los bordes deseados. Esto es especialmente problemático en imágenes 2D con contornos negros, ya que el algoritmo intenta aproximar estos contornos mediante triángulos, pero estos se eliminan continuamente debido a que su tamaño tiende a reducirse hasta no cumplir con las dimensiones mínimas aceptadas por la malla.

Como resultado de los problemas encontrados con esta implementación, se decidió limitar el alcance del método únicamente a la detección de bordes de la imagen con respecto a un fondo blanco, sin considerar la información de color contenida en esta. Esta modificación, además de reducir significativamente los tiempos de cálculo, produjo resultados mucho más cercanos a la imagen de referencia. Por lo tanto, la versión del algoritmo incluida en la herramienta desarrollada se basa en esta implementación.

Para este método alternativo, en lugar de realizar todos los cálculos en la imagen original, se realiza un procesamiento similar al utilizado en el método basado en Canny durante la inicialización de la imagen. Para esto, se genera una copia en la cual todos los pixeles que no son blancos se transforman en negro. Esto efectivamente transforma el formato de la imagen a escala de grises, de modo que cada pixel ahora se representa con un valor de 0 a 255 en lugar de los 3 canales BGR utilizados previamente.

Con esta imagen preprocesada, todo el proceso de generación y refinamiento de la malla se lleva a cabo de la misma manera, mientras que la modificación más relevante se encuentra en el cálculo del error de aproximación sobre los triángulos. Originalmente, para cada triángulo era necesario recorrer sus puntos interiores para calcular el color promedio de los pixeles, y luego recorrer la lista nuevamente para calcular la distancia de cada pixel al color promedio calculado.

Con la modificación realizada, solo es necesario recorrer la lista una vez para calcular el color promedio dentro de un triángulo. Dado que se tienen valores en escala de grises, el color promedio obtenido es un valor entero entre 0 y 255. Por lo tanto, es posible reinterpretar el error de aproximación como la distancia del promedio a uno de los dos extremos: blanco o negro. De esta manera, se puede calcular el error de aproximación como la distancia mínima entre el promedio y los valores de blanco y negro.

Así, el error de aproximación tomará valores entre 0 y 127, siendo un valor más alto indicativo de una cantidad similar de pixeles blancos y negros dentro de un triángulo, mientras que un valor de 0 implica que todos los pixeles en el triángulo son del mismo color. De esta forma, se obtiene una métrica que conserva el comportamiento de los vértices al ser desplazados, ya que tenderán a disminuir el valor del error de aproximación, generando triángulos que tienden a contener pixeles del mismo color.

Como se puede notar, el cálculo del color promedio y el error de aproximación se simplifica considerablemente. En lugar de recorrer el listado de pixeles en dos ocasiones, ahora solo es necesario hacerlo una vez para calcular el valor promedio. Posteriormente, se puede obtener el error de aproximación mediante una simple diferencia entre dos valores. Además, todo este proceso se realiza en pixeles con información de un solo canal de color, a diferencia del caso

anterior donde se requería calcular la distancia euclidiana para una lista de pixeles con tres canales BGR.

Como resultado de estos cambios, el algoritmo muestra una mejora en eficiencia al reducir la cantidad de operaciones necesarias. Además, la calidad de los resultados también se ve beneficiada, especialmente en casos donde las imágenes de referencia contienen bordes delgados. En tales situaciones, el nuevo enfoque descarta información que no puede ser aproximada por el algoritmo anterior, evitando así la generación de elementos en la malla que posteriormente serán eliminados.

3.3.2. Implementación

La implementación de este método de detección de bordes se llevó a cabo utilizando programación orientada a objetos, siguiendo un enfoque similar al método basado en el algoritmo de Canny. Para esto, se crearon varias clases destinadas a representar los diferentes elementos de una malla de triángulos, empleando una estructura de datos basada en la estructura *half-edge* descrita anteriormente, con leves modificaciones con el fin de facilitar los cálculos que se llevarán a cabo sobre la malla. En la figura 3.13 se ilustra el diagrama de las clases utilizadas para este método.

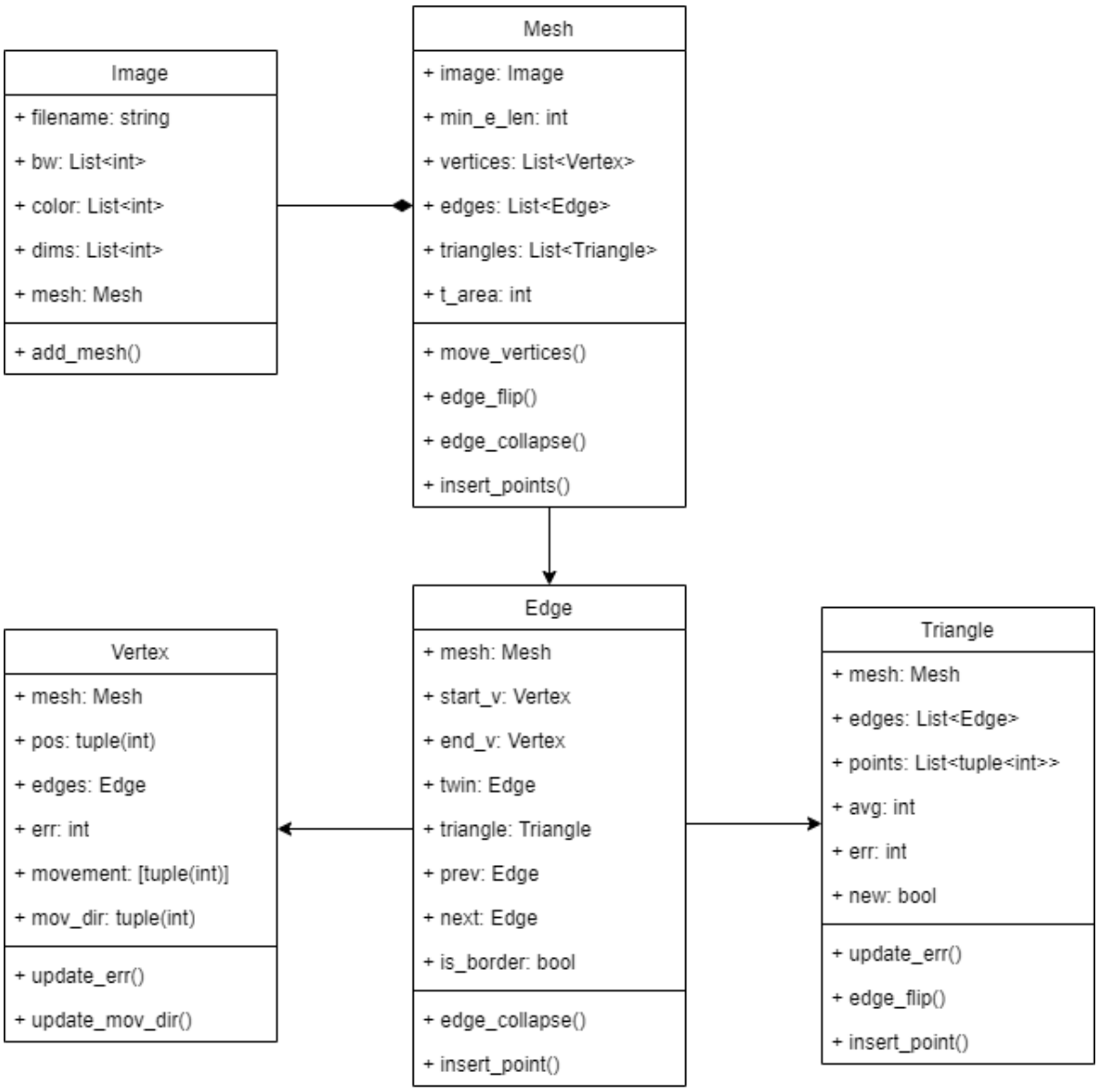

Figura 3.13: Diagrama de clases para el método basado en triangulación.

3.3.2.1. Clase Image

De forma similar al método basado en Canny, se crea una clase Image que servirá como base para el procesamiento posterior. Al igual que en el método mencionado anteriormente, esta clase almacena una copia de la imagen de referencia, así como una versión de la imagen en blanco y negro que se utilizará para el cálculo de los bordes. Los objetos de esta clase siempre deben estar asociados a una malla, representada por un objeto de la clase `Mesh`.

Aunque existen similitudes con la clase `Image` utilizada para el método basado en Canny, esta clase tiene sus propios campos y métodos, diseñados para el posterior procesamiento que se realizará con la triangulación de la imagen. Los campos de la clase son los siguientes:

- **filename:** Almacena el nombre del archivo como una cadena de texto.
- **color** Almacena una copia de la imagen original, que se puede utilizar para generar visualizaciones del proceso de triangulación y refinamiento de la malla.
- **bw:** Almacena una copia de la imagen tras aplicar la función `threshold`, convirtiendo a negro todos los pixeles que no sean blancos en la imagen original.
- **shape:** Almacena las dimensiones de la imagen asociada, información que se utilizará para la generación de la malla.
- **mesh:** Contiene una referencia al objeto `Mesh` asociado.

Cabe destacar que para la inicialización de las imágenes almacenadas en los campos `color` y `bw`, se realiza un paso adicional que no se lleva a cabo en el método basado en Canny. Este paso consiste en copiar el contenido principal de la imagen en un lienzo cuadrado en blanco, con dimensiones correspondientes al 120 % de la dimensión más grande de la imagen. De esta manera, se genera un borde blanco alrededor de la figura cuyos bordes se desean detectar, evitando así que entre en contacto con los límites de la malla y permitiendo una mejor triangulación.

La asociación de un objeto `image` con un objeto de la clase `Mesh` se realiza a través del método `add_mesh()`. Este método recibe diferentes parámetros, incluyendo el número de celdas en la malla inicial (tanto horizontal como vertical) y la longitud mínima de las aristas en la malla creada. Esta restricción se utilizará en múltiples etapas del refinamiento y determinará los casos en los que es posible insertar puntos y aquellos en los que se debe llevar a cabo un colapso de aristas.

Además del método `add_mesh()`, la clase `Image` también cuenta con métodos a modo de wrapper para llamar a distintas funciones de la clase `Mesh`. Además, incluye funciones para mostrar las mallas generadas en pantalla, superponiendo todos los vértices y aristas sobre la imagen de referencia.

3.3.2.2. Clase Mesh

La clase `Mesh` contiene las referencias a todos los elementos de la malla. Esta posee una referencia a su objeto `Image` asociado, así como listas para cada tipo de elemento: aristas, triángulos y vértices. Esta clase se encarga del manejo de los pasos para el refinamiento de la malla, así como la creación y eliminación de nuevos vértices, aristas y triángulos que surgen durante estos procesos.

Además de las referencias a los distintos elementos de la malla, la clase también tiene dos campos que almacenan valores de referencia para realizar comparaciones durante el refinamiento de la malla. Estos campos reciben el nombre de `min_e_len` y `t_area`. El primero representa la longitud mínima admitida para las aristas, mientras que el segundo representa las áreas iniciales de los triángulos en la malla, ya que debido a la estructura de grilla utilizada se tiene que todos los triángulos poseen la misma área al comienzo.

La inicialización de una malla comienza con la ubicación de los vértices, los cuales son posicionados a intervalos regulares para abarcar toda la imagen de referencia. Luego, se establecen las conexiones entre los vértices para formar celdas cuadrados separadas por diagonales. Para determinar las diagonales, se generan conexiones tentativas y se evalúa cuál de ellas produce el menor error de aproximación. La diagonal seleccionada se agrega definitivamente al listado de aristas almacenado en el objeto. Finalmente, se generan los triángulos necesarios para que la malla esté bien formada, estableciendo todas las referencias entre los elementos presentes en esta.

La clase `Mesh` incluye también métodos para la inserción y eliminación de elementos dentro de la malla, como vértices, triángulos y aristas. Cada uno de estos métodos se encarga de mantener la consistencia de las referencias y preservar la integridad de la malla después de cada operación de inserción o eliminación.

Además, esta clase proporciona métodos cuya ejecución implica recorrer la lista de elementos de la malla, como el cálculo de los errores de aproximación de los triángulos, el desplazamiento de los vértices y los pasos del refinamiento.

3.3.2.3. Clase Triangle

Un objeto de la clase `Triangle` representa un triángulo dentro de la malla, y contiene los siguientes campos:

- **mesh:** Contiene una referencia al objeto `Mesh` asociado.
- **edges:** Contiene una lista ordenada de las aristas que forman el triángulo, dispuestas en sentido antihorario.
- **points:** Contiene las coordenadas de los pixeles en el interior del triángulo, representados por las coordenadas de inicio y fin de *scanlines* horizontales. Estos puntos se recalculan constantemente después de cada desplazamiento de vértices llevado a cabo.
- **avg:** Representa el promedio del color de todos los pixeles contenidos en el triángulo.
- **err:** Indica el error de aproximación del color promedio.
- **new:** Flag que determina si el triángulo fue creado recientemente. Esto evitar realizar pasos de refinamiento en triángulos nuevos, ya que esto podría provocar bucles infinitos.

Es importante destacar que un triángulo no almacena de forma directa referencias a los vértices que lo definen, ya que es posible acceder a estos a través del listado de aristas asociadas.

Además de los getters y setters correspondientes, la clase `Triangle` cuenta con funciones asociadas al cálculo del error de aproximación dentro del triángulo. El primer método utilizado para esto es `update_points()`, el cual actualiza la lista de puntos contenidos en el triángulo. Esto se realiza mediante el algoritmo basado en *scanlines* descrito en la sección 2.6.2, obteniendo así los puntos necesarios para calcular el color promedio y el error de aproximación de los pixeles en el triángulo.

Una vez calculados los puntos interiores del triángulo, se determina el error de aproximación utilizando el método `update_err()`. Este método obtiene el color de todos los pixeles presentes en la lista de puntos del triángulo y calcula su promedio. A partir de este valor se calcula el error de aproximación, el cual se almacena en el campo correspondiente. Todo este procesamiento se realiza utilizando la librería Numpy, con el fin de mejorar la eficiencia de los cálculos llevados a cabo.

Además de los métodos asociados al cálculo del error, la clase `Triangle` también incluye métodos relacionados con el refinamiento de la malla, específicamente aquellos que se aplican a los triángulos. El primero de ellos es el método `edge_flip()`, el cual recibe como parámetro un triángulo adyacente y realiza una reestructuración de las conexiones en la malla para cambiar la dirección de la arista compartida. También se encuentra el método `insert_point()`, el cual se encarga de insertar puntos en el centroide del triángulo y crear todos los elementos asociados para mantener la integridad de la malla.

Por último, la clase `Triangle` proporciona varias funciones auxiliares destinadas a la obtención de valores necesarios para otros procesos. Estas funciones incluyen el cálculo del borde más corto o más largo del triángulo, la obtención de la lista de vértices del triángulo, la obtención del ángulo más grande, y el cálculo del *bounding box* del triángulo como una forma sencilla de estimar su área.

3.3.2.4. Clase Vertex

Un objeto de la clase `Vertex` representa un vértice dentro de la malla, y posee los siguientes campos:

- **mesh:** Contiene una referencia al objeto `Mesh` asociado.
- **pos:** Almacena las coordenadas en los ejes x e y que representan la posición del vértice con respecto a la imagen.
- **edges:** Almacena un listado de aristas asociadas al vértice. Dado que la estructura *half-edge* proporciona dirección a las aristas, todas las aristas en esta lista son aquellas que tienen al vértice como punto de inicio.
- **err:** Error de aproximación del vértice, calculado como la suma del error de aproximación de los triángulos adyacentes a este, dividida en 3.

Como se puede observar, no hay un campo que almacene una lista de triángulos adyacentes al vértice. Esto se debe a que se puede aprovechar la estructura de datos *half-edge* utilizada, por lo que basta con obtener los triángulos asociados a las aristas almacenadas en el campo `edges`.

Junto con los campos mencionados anteriormente, también se tienen campos que almacenan información relacionada al desplazamiento de los vértices. El primero de ellos es el campo `movement`, que almacena las direcciones permitidas de movimiento para un vértice específico. Este atributo es una lista de tuplas, donde cada tupla representa un desplazamiento

del vértice en una dirección determinada. Las direcciones de desplazamiento se establecen durante la creación del vértice y dependen de su ubicación en la malla. Para la mayoría de los vértices, las direcciones de desplazamiento disponibles son: arriba, abajo, izquierda y derecha. Sin embargo, para los vértices ubicados en los bordes de la malla, el desplazamiento se restringe a un solo eje, ya sea horizontal o vertical. Por último, para los vértices ubicados en las esquinas de la malla, el desplazamiento está deshabilitado por completo, por lo que la lista de direcciones de desplazamiento posibles estará vacía.

El segundo campo asociado al desplazamiento de los vértices es `mov_dir`, que almacenará la dirección de desplazamiento elegida para la próxima iteración. Es necesario almacenar esta información en un campo dedicado, ya que el cálculo de la dirección del desplazamiento y el desplazamiento en sí se realizan en dos pasos diferentes, por lo que esta información debe ser almacenada. Además, el valor de este campo también es consultado en otros puntos del refinamiento de la malla, ya que en ocasiones es necesario determinar qué vértices no se han desplazado durante la última iteración.

Dentro de los métodos de esta clase se tienen todos los getters y setters correspondientes, así como métodos para calcular el error de aproximación en el vértice y determinar la dirección de desplazamiento que minimiza este error. También se incluyen métodos auxiliares que proporcionan información sobre la geometría cercana al vértice, como obtener la lista de triángulos adyacentes o la lista de vértices conectados con las aristas asociadas.

3.3.2.5. Clase Edge

Un objeto de la clase `Edge` representa una arista dentro de la malla y cuenta con los siguientes campos:

- **mesh:** Contiene una referencia al objeto `Mesh` asociado.
- **start_v:** Referencia al objeto `Vertex` que representa el punto de inicio de la arista.
- **end_v:** Referencia al objeto `Vertex` que representa el punto final de la arista.
- **twin:** Referencia al objeto `Edge` que representa la arista gemela a la arista actual. Esta arista se encuentra definida por los mismos vértices, pero tiene sentido contrario.
- **triangle:** Referencia al objeto `Triangle` que representa el triángulo adyacente a la arista. Dado que se utiliza una estructura de datos basada en la estructura *half-edge*, cada arista tiene un solo triángulo asociado.
- **prev:** Referencia a la arista anterior en relación al triángulo asociado.
- **next:** Referencia a la arista siguiente en relación al triángulo asociado.
- **is_border:** Flag que determina si la arista define un borde dentro de la imagen. Si es así, los puntos de la arista serán utilizados para la generación del archivo .poly.

Es importante destacar que la clase `Edge` es la única que almacena referencias a los otros dos elementos en la malla, esto es, a los vértices y triángulos. Por lo tanto, esta clase actúa

como un intermediario entre ambas clases, facilitando la comunicación y el intercambio de información cuando es necesario.

Con respecto a los métodos de la clase, se tienen todos los getters y setters correspondientes. Además, se tienen métodos para el refinamiento de la malla. Uno de estos métodos es el *edge-collapse*, cuya implementación se detalló en las secciones anteriores.

Otro método asociado al refinamiento de la malla es la inserción de puntos en el medio de una arista. Este método determina el punto medio de la arista en cuestión y realiza las conexiones necesarias con los vértices no adyacentes. Esto implica reconstruir las conexiones en la malla, generando nuevos triángulos y aristas para preservar la integridad de la estructura.

Además, se tienen múltiples métodos que proporcionan información sobre la geometría en la vecindad de una arista, entre los cuales se encuentran el cálculo de ángulos opuestos y adyacentes a esta. Esta información se utiliza en el refinamiento de la malla para determinar las condiciones necesarias para realizar un *edge-flip* o un *edge-collapse*. También se incluyen funciones para calcular el punto medio y la longitud de una arista, datos que son utilizados para la inserción de puntos y los *edge-collapse*.

3.4. Generación de archivos .poly

Tanto la implementación del método basado en Canny como la del método basado en triangulaciones entregan como resultado una lista de vértices que definen un contorno cerrado. Además, para cada contorno, se proporciona información sobre si el área contenida dentro de él es sólida o si debe considerarse como un agujero dentro de la triangulación posterior, en cuyo caso se entrega un punto contenido dentro del contorno.

Estos contornos son procesados por una función final encargada de generar los archivos .poly, tomando estos puntos y estructurándolos según el formato específico de dicho tipo de archivo.

La principal forma de interactuar con la herramienta es a través del programa make_poly.py. Este recibe como parámetro principal la imagen en la que se desea llevar a cabo la detección de bordes. Además, es capaz de recibir una variedad de parámetros ingresados por el usuario, los cuales se detallan en el capítulo 4.

En la figura 3.14 se muestra la iteración final del proceso de triangulación de una imagen. Las aristas de la malla están representadas con el color verde, mientras que las aristas que forman parte de los bordes detectados se resaltan en rojo.

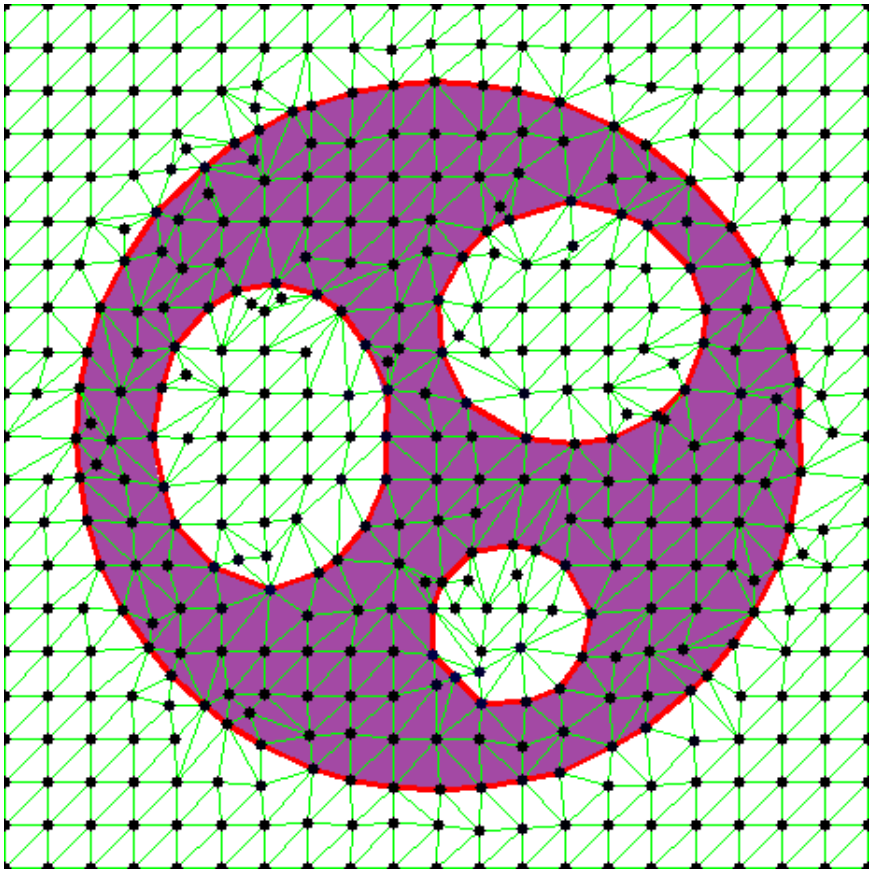

Figura 3.14: Resultado de aplicar el método basado en triangulación en una figura simple.

Capítulo 4

Resultados y Análisis

La calidad de los resultados obtenidos puede ser verificada mediante la inspección visual, ya que resulta evidente a simple vista si la aproximación realizada es congruente con la imagen de referencia. Sin embargo, además de evaluar la proximidad de los bordes generados con la imagen original, también es posible evaluar su idoneidad para una posterior triangulación mediante el análisis de las medidas de tendencia central de las aristas. Al analizar las distribuciones de las longitudes de las aristas, se observa que aquellas distribuciones más uniformes representarán bordes que, al ser triangulados, generarán triángulos de dimensiones más regulares.

La calidad de la aproximación para una imagen en particular dependerá del método utilizado para la generación de los bordes, los parámetros utilizados en este proceso y, sobre todo, de las características particulares de la imagen utilizada. Por lo tanto, ningún método utilizado garantizará resultados satisfactorios en todas las circunstancias, ya que esto dependerá de cada caso específico. No obstante, a través de la experimentación con diferentes tipos de imágenes, se identificaron dos combinaciones de parámetros por defecto, una para cada herramienta, que ofrecen aproximaciones de buena calidad para la gran mayoría de las imágenes.

La herramienta desarrollada es capaz de recibir distintos parámetros a través de la consola, y los parámetros modificables varían dependiendo del método de detección de bordes utilizado. Para el método basado en Canny, se pueden ajustar los siguientes parámetros:

- **reduction:** Se refiere al algoritmo de eliminación de vértices a utilizar. Por defecto, se utiliza el algoritmo de eliminación híbrido descrito en la sección 3.2.1.2. Sin embargo, el usuario también puede optar por utilizar el método de eliminación a intervalos fijos o a intervalos variables.
- **len:** Si se utiliza el método de eliminación híbrido o a intervalos fijos, es posible establecer la longitud deseada para las aristas generadas. En el caso del algoritmo híbrido, esta longitud representa una cota superior para la longitud de las aristas.

  Experimentalmente, se observó que una longitud de arista de 10 pixeles produce buenos resultados para imágenes con una dimensión aproximada de 500×500 pixeles.

- **maxdist:** En el caso del método de eliminación híbrido o a intervalos variables, se puede fijar la distancia máxima entre una arista y los bordes de la imagen de referencia.

  Experimentalmente, se observó que el valor más bajo para este parámetro, equivalente a 1 pixel de distancia, ofrece buenos resultados sin generar una cantidad excesiva de puntos.

- **fusedist:** Es posible establecer una distancia, medida en pixeles, en la cual todos los vértices que se encuentren a una distancia menor a este valor serán fusionados en uno solo. Esto se realiza con el objetivo de evitar la existencia de aristas de longitud muy reducida. Por defecto, los vértices que estén a una distancia menor a 5 pixeles serán fusionados.

Por otro lado, en el método basado en triangulaciones es posible ajustar los siguientes parámetros:

- **xy:** Representa la cantidad de celdas en la malla inicial para ambas dimensiones. Este valor se puede proporcionar como una tupla o un número entero si ambas dimensiones tienen la misma cantidad de celdas. Por defecto, se considera una malla inicial de 20×20 celdas.

- **it:** Numero de iteraciones para el desplazamiento de vértices. Experimentalmente se determinó que 40 iteraciones proporcionan una disminución máxima del error de aproximación en la mayoría de las imágenes, por lo que este es el valor predeterminado.

- **minlen:** Longitud mínima de una arista. Por defecto, se establece un valor de 3 pixeles, el cual puede aumentarse si se desea una aproximación más gruesa. Se desaconseja disminuir este valor, ya que se podrían generar desplazamientos de vértices que resulten en triángulos con áreas nulas o negativas.

4.1. Efectividad

A continuación se presenta una comparación de los resultados obtenidos mediante los dos métodos utilizados, con diferentes variaciones en los parámetros. Se utilizó la misma imagen de referencia en todas las pruebas mostradas, con el fin de facilitar la comparación entre los resultados de los distintos métodos.

La imagen de referencia utilizada corresponde a la figura 4.1, seleccionada debido a que presenta bordes con alta curvatura y también secciones que pueden ser aproximadas con segmentos rectos. Además, la imagen contiene secciones disjuntas que forman parte de una misma figura, así como agujeros que permiten probar los algoritmos encargados de determinar la contención de un polígono dentro de otro.

Figura 4.1: Imagen utilizada como referencia, correspondiente a una ilustración de un oso panda [9].

4.1.1. Efectividad del método basado en Canny

Para evaluar la efectividad del método basado en el algoritmo de Canny, se analizarán los resultados obtenidos mediante la ejecución de este con los tres algoritmos distintos utilizados para reducir la cantidad de vértices.

4.1.1.1. Eliminación de vértices a intervalos constantes

Para el algoritmo de eliminación de vértices a intervalos constantes, el usuario tiene la opción de modificar el parámetro `len`, que representa la longitud de las aristas generadas. El valor por defecto es 20, lo que equivale a una longitud de aproximadamente 20 pixeles. Se decidió variar este parámetro aumentándolo y disminuyéndolo, realizando pruebas con un `len` de 10 y 40 pixeles.

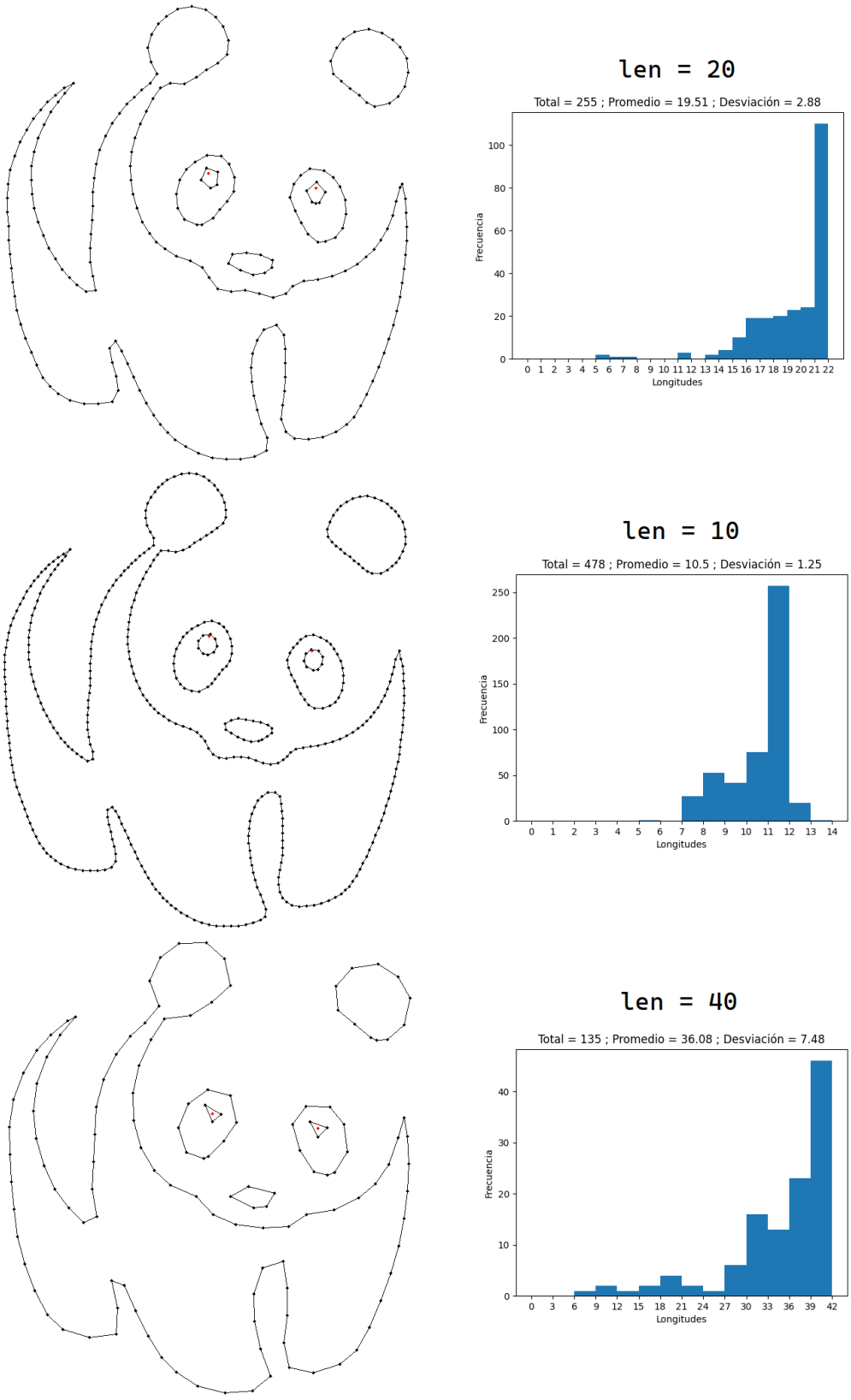

Figura 4.2: Resultado de aplicar el algoritmo de eliminación a intervalos fijos, modificando el parámetro `len`.

Como se muestra en la figura 4.2, cuando `len` es igual a 20 pixeles, no se observa una pérdida significativa de detalle en los polígonos más grandes de la imagen. Sin embargo, se puede apreciar que la aproximación de las pupilas en la figura difiere considerablemente de una circunferencia. Una situación similar ocurre cuando el valor de `len` aumenta a 40, notándose una pérdida casi total de detalle en áreas más finas de la figura, como las pupilas o la nariz del animal.

En contraste, para el caso en el que `len` = 10, se observa una mayor densidad de vértices en comparación con los otros dos casos, obteniéndose una aproximación bastante fiel a la imagen de referencia. Sin embargo, la cantidad de puntos generados y la longitud reducida de las aristas hacen que este método no sea ideal para la generación de triangulaciones.

Con respecto a las distribuciones de longitudes obtenidas, en los tres histogramas generados se observan sesgos hacia la derecha, ya que en cada prueba realizada, los valores que se alejan de la media y la moda son menores a estas. Por lo tanto, es posible concluir que el valor ingresado por el usuario actúa como una cota superior para la longitud de las aristas generadas.

En la tabla 4.1 se presenta un resumen de los resultados obtenidos con esta variación del algoritmo:

Valor parámetro	Aristas generadas	Longitud promedio	Desv. estándar
len = 20	255	19.51	2.88
len = 10	478	10.5	1.25
len = 30	135	36.08	7.48

Tabla 4.1: Resumen de resultados de aplicar el algoritmo de eliminación a intervalos fijos, modificando el parámetro `len`.

4.1.1.2. Eliminación de vértices a intervalos variables

En el algoritmo de aproximación utilizado anteriormente, se observó que el parámetro `len` ingresado por el usuario determina tanto la cantidad de puntos a utilizar como la longitud de las aristas generadas, sin depender de las características de la imagen. En contraste, en el segundo algoritmo, conocido como el algoritmo de eliminación de vértices a intervalos variables, el input del usuario establece la calidad de la aproximación final, pero el número de puntos utilizados depende únicamente de las características de la imagen de referencia. Esto implica que el usuario tiene un menor control sobre la cantidad de aristas creadas.

El parámetro a ajustar al utilizar este algoritmo de aproximación es `maxdist`, que representa la distancia máxima permitida entre las aristas generadas y los bordes de la imagen original. Se comenzará utilizando el valor predeterminado asignado a este parámetro, que es de 1 pixel. Esto implica que, para cada arista, todos los puntos en ella se encuentran a una distancia máxima de 1 pixel de los bordes de la imagen de referencia. Luego, se aumentará el valor de este parámetro, realizando pruebas con un `maxdist` de 2 y 5.

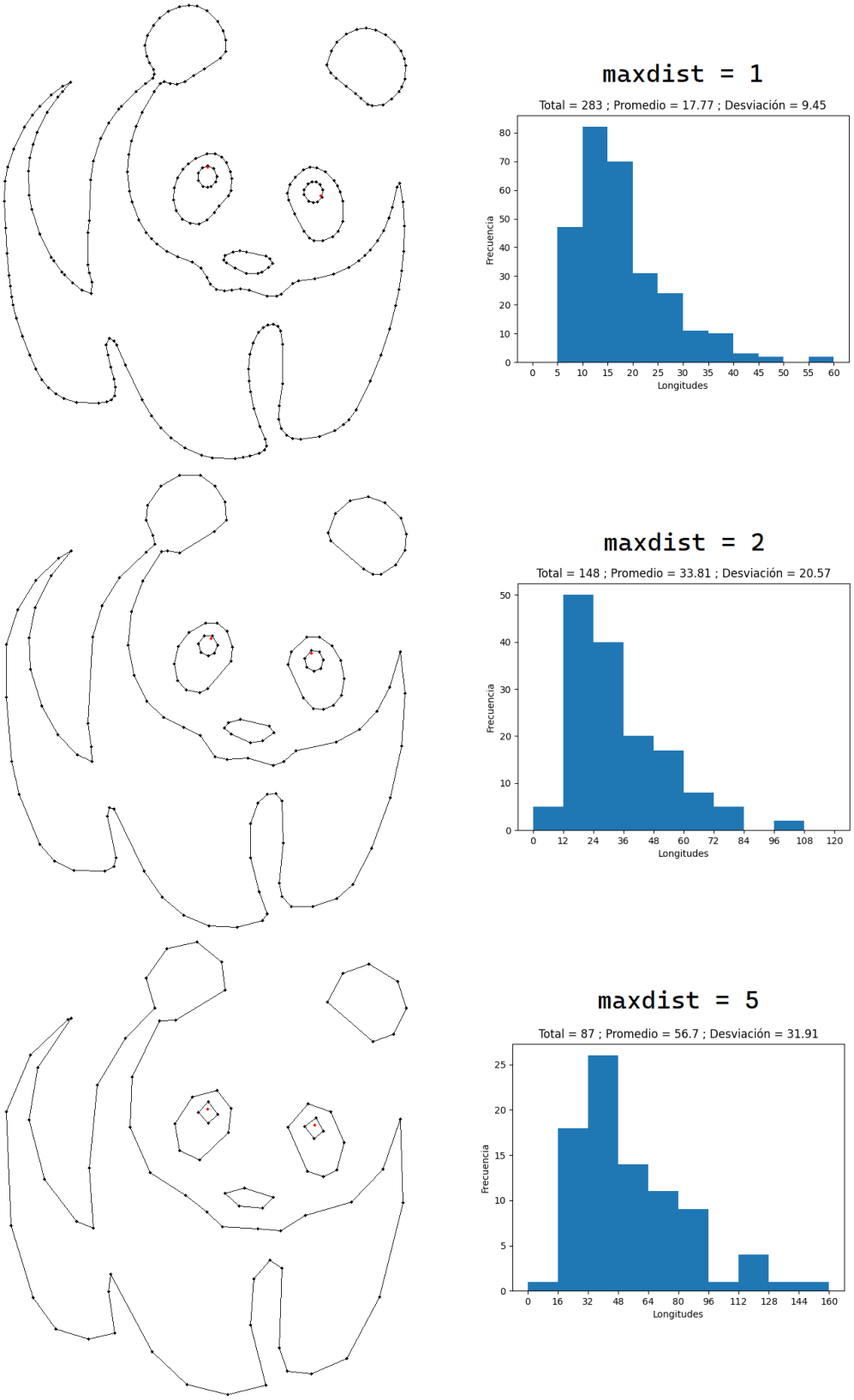

Figura 4.3: Resultado de aplicar el algoritmo de eliminación a intervalos variables, modificando el parámetro `maxdist`.

Como se puede apreciar en la figura 4.3, la aproximación se realiza sin pérdida de detalles, independiente del valor de `maxdist`. Incluso en el caso de una aproximación más gruesa, cuando maxdist = 5, se obtiene una representación adecuada de las pupilas y la nariz del animal, a pesar de las curvas cerradas presentes en estas áreas de la imagen.

A pesar de las ventajas descritas anteriormente, se observa que la variación en las longitudes de las aristas es mucho mayor que en los casos anteriores. Por ejemplo, en el caso donde `maxdist` = 1, estas longitudes oscilan entre 5 y 60 pixeles, con una desviación estándar de 9.45. Este valor es mucho mayor que todas las desviaciones estándar obtenidas con el método anterior y aumenta aún más a medida que aumenta el valor de `maxdist`. Esta gran variabilidad en las longitudes puede resultar inconveniente para la posterior triangulación que se desea realizar, dado que en todos los casos el valor máximo es más de 10 veces superior al mínimo.

Es importante destacar que la distribución de valores es opuesta a las distribuciones generadas por el método anterior, ya que en este caso se observa un sesgo hacia la izquierda. Esto implica que los valores tienden a estar cerca del valor mínimo, mientras que los *outliers* son valores mayores que este.

En la tabla 4.2 se presenta un resumen de los resultados obtenidos con esta variación del algoritmo:

Valor parámetro	Aristas generadas	Longitud promedio	Desv. estándar
maxdist = 1	283	17.77	9.45
maxdist = 2	148	33.81	20.57
maxdist = 5	87	56.7	31.91

Tabla 4.2: Resumen de resultados de aplicar el algoritmo de eliminación a intervalos variables, modificando el parámetro `maxdist`.

4.1.1.3. Algoritmo híbrido de eliminación de vértices

A continuación, se presentarán los resultados obtenidos al utilizar el método híbrido de eliminación de vértices. En este caso, se ajustarán los parámetros `len` y `maxdist`, que fueron utilizados en los dos casos anteriores. En primer lugar, se emplearán los valores por defecto para ambos parámetros, que corresponden a una longitud de arista de 20 pixeles y una distancia máxima de 1 pixel. Posteriormente, se duplicarán los valores de ambos parámetros, utilizando `len` = 40 y `maxdist` = 2, con el objetivo de generar una aproximación más gruesa que aún conserve los detalles importantes de la imagen.

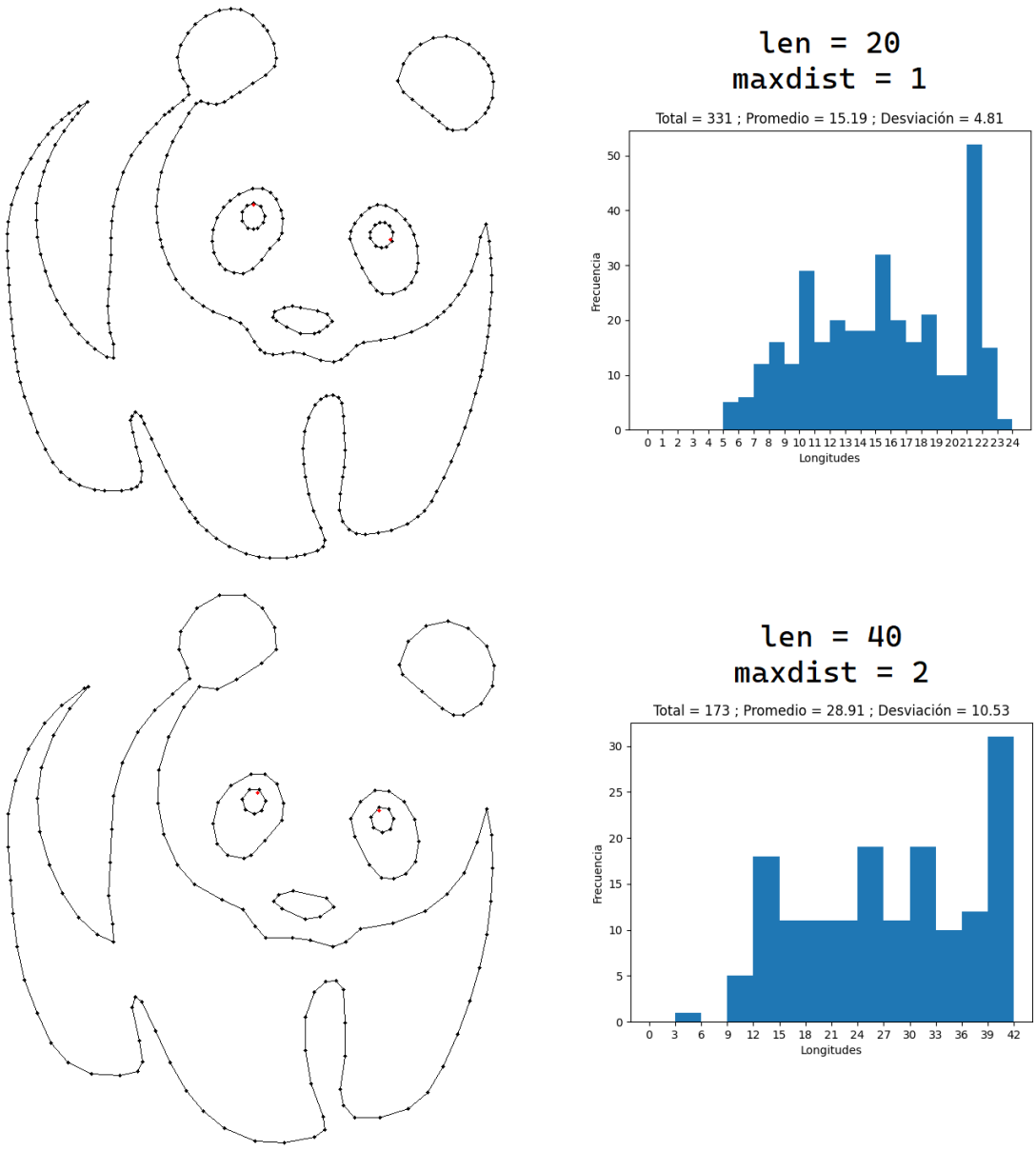

Figura 4.4: Resultado de aplicar el algoritmo de eliminación híbrido, modificando tanto `len` como `maxdist`.

Como se puede apreciar en la figura 4.4, no se pierden detalles importantes de la imagen en ninguno de los dos casos. Además, se obtienen distribuciones de longitudes mucho más uniformes en comparación con los casos anteriores. Para este algoritmo de eliminación de vértices, los valores ingresados por el usuario para la longitud de arista no tienen un impacto tan significativo en los resultados finales como en los algoritmos anteriores, ya que las longitudes promedio obtenidas se alejan en todos los casos al valor de `len` ingresado, correspondiendo aproximadamente al 75 % de este valor. Sin embargo, la moda sigue siendo cercana al valor ingresado por el usuario.

También es posible notar que, al duplicar los valores de ambos parámetros, fue posible reducir la cantidad de puntos a un poco más de la mitad en comparación con los resultados obtenidos del primer caso.

Finalmente, se puede apreciar que en ambos casos la desviación estándar y el rango de valores son mucho más reducidos en comparación con los otros dos algoritmos. Por lo tanto, se puede concluir que el uso de un algoritmo de reducción de vértices con las características de los dos primeros métodos es capaz de producir resultados ideales para la realización de triangulaciones.

En la tabla 4.3 se presenta un resumen de los resultados obtenidos con esta variación del algoritmo:

Valor parámetro	Aristas generadas	Longitud promedio	Desv. estándar
len = 20 ; maxdist = 1	331	15.19	4.81
len = 40 ; maxdist = 2	173	28.91	10.53

Tabla 4.3: Resumen de resultados de aplicar el algoritmo de eliminación híbrido, modificando tanto `len` como `maxdist`.

4.1.2. Efectividad del método basado en triangulación

A diferencia del método basado en Canny, el método basado en triangulaciones no ofrece una combinación de parámetros que garantice una aproximación completamente precisa de la imagen de referencia. Mientras que con el algoritmo basado en Canny es posible generar aristas de longitud muy reducida para capturar todos los detalles finos (aunque estos resultados no sean óptimos para la posterior triangulación), los resultados obtenidos con el método basado en triangulación dependen en gran medida de las características de la imagen de referencia. Por lo tanto, es necesario experimentar con diferentes combinaciones de parámetros hasta lograr un resultado satisfactorio.

4.1.2.1. Variación de las dimensiones de la malla inicial

En la primera prueba realizada con este método, se decidió variar la cantidad de celdas presentes en la malla inicial. Para esto, se comenzó utilizando los valores por defecto establecidos para todos los parámetros. Esto implica utilizar una malla de 20×20 celdas (parámetro `xy`), y establecer un largo mínimo de arista de 3 pixeles (parámetro `minlen`). El proceso de desplazamiento de vértices y optimización de la malla se llevó a cabo durante 40 iteraciones (parámetro `it`). Posteriormente, se modificó el valor de `xy`, aumentando el número de celdas iniciales a 25×25 y finalmente a 30×30 celdas. Los resultados mostrados en la figura 4.5 representan los bordes obtenidos a partir de la malla final.

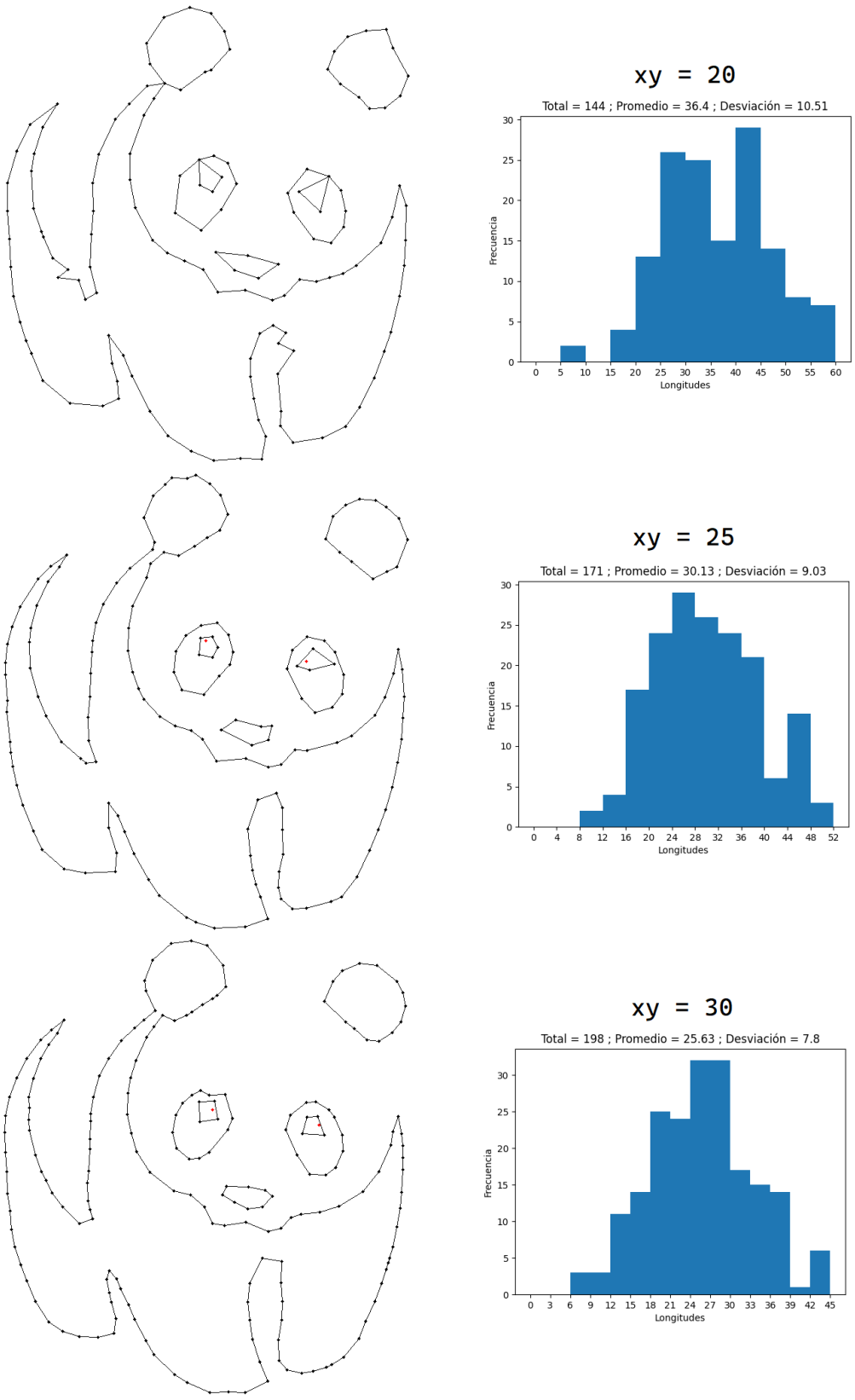

Figura 4.5: Resultado de aplicar el método basado en triangulaciones, variando la cantidad de celdas en la malla inicial.

Como se puede observar en la figura 4.5, la calidad de la aproximación inicial es significativamente inferior en comparación con los resultados obtenidos mediante el método anterior. No obstante, la cantidad de errores de aproximación disminuye a medida que aumenta la cantidad de celdas iniciales. Estos errores de aproximación se presentan especialmente en áreas donde hay curvas cerradas, como entre las piernas o en las pupilas del animal. Además, para el caso inicial con una malla de 20×20, también se evidencia una pérdida de información con respecto a los agujeros presentes en la figura, ya que las pupilas se fusionan con el resto del polígono que define los ojos, formando un solo polígono con concavidades.

En cuanto a la distribución de longitudes de arista, se observa que en los tres casos las distribuciones se asemejan a una distribución normal. Además, en los tres casos, las desviaciones estándar son menores a un tercio de la longitud promedio, lo cual es similar a las distribuciones de longitudes obtenidas mediante el método basado en Canny utilizando el algoritmo de eliminación de vértices híbrido.

En la tabla 4.4 se presenta un resumen de los resultados obtenidos variando el parámetro `xy`:

Valor parámetro	Aristas generadas	Longitud promedio	Desv. estándar
xy = 20	144	36.4	10.51
xy = 25	171	30.13	9.03
xy = 30	198	25.63	7.8

Tabla 4.4: Resumen de resultados de aplicar el método basado en triangulaciones, variando la cantidad de celdas en la malla inicial..

4.1.2.2. Variación del número de iteraciones

A continuación se presentan los resultados obtenidos al modificar el número de iteraciones del proceso para una misma imagen, utilizando una malla de 30×30 celdas y un largo mínimo de arista de 3 pixeles.

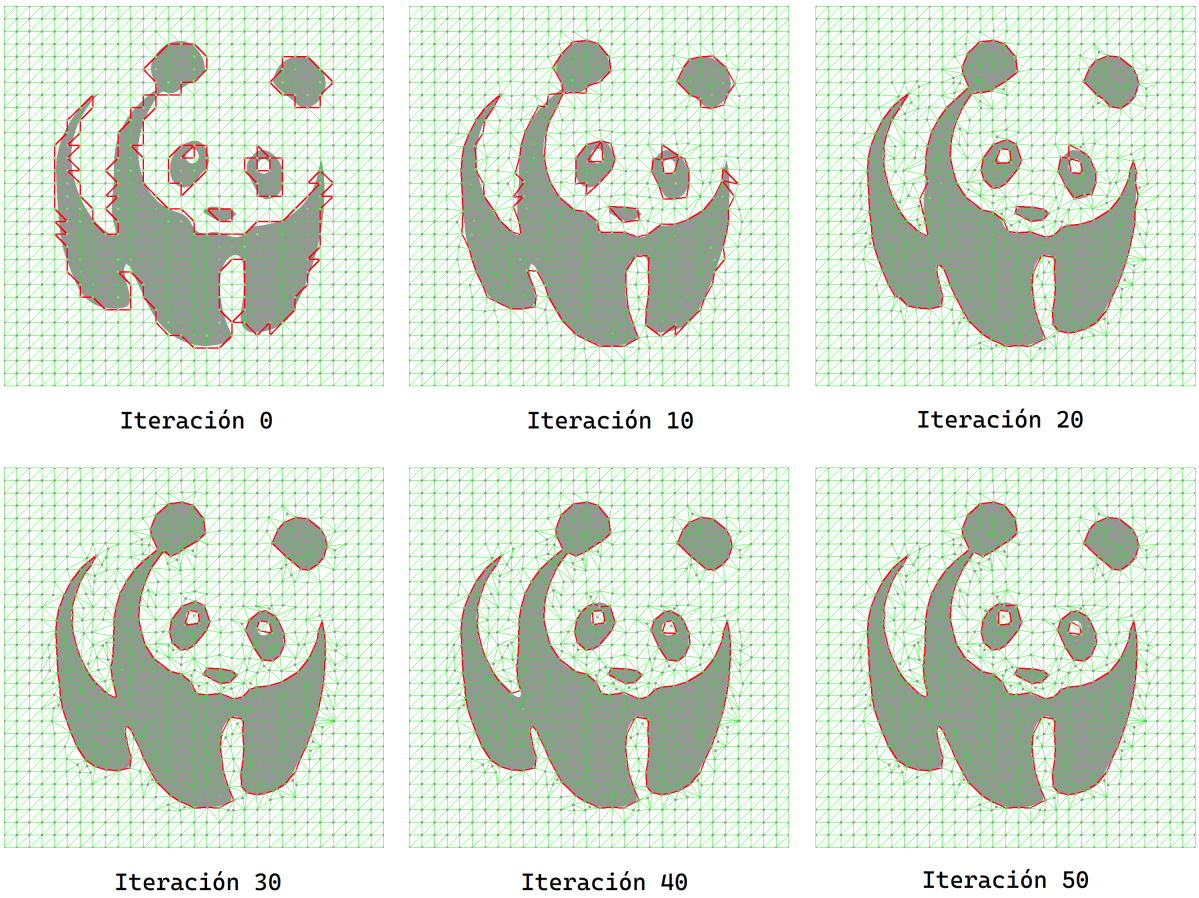

Figura 4.6: Resultados entregados para distintos valores de iteraciones totales.

En la figura 4.6 se puede observar que a partir de la iteración 30 en adelante las mejoras en la calidad de la aproximación son mínimas. Por lo tanto, se decidió establecer 40 iteraciones como el valor por defecto, ya que se verificó que, en la gran mayoría de los casos, para diferentes imágenes se obtuvieron las mejores aproximaciones posibles (para el conjunto de parámetros entregados) al alcanzar esta iteración.

4.1.3. Comparación de resultados para distintas imágenes

A continuación, se presenta un resumen de los resultados obtenidos al aplicar los algoritmos con las variaciones previamente mencionadas a cuatro imágenes diferentes, con el propósito de comprarar la cantidad de aristas generadas.

La primera imagen corresponde a la figura 4.1, utilizada en todos los ejemplos anteriores. Esta contiene una ilustración estilizada de un oso panda con dimensiones de 642×716 pixeles, y fue seleccionada por presentar bordes con distintos tipos de curvas.

Figura 4.7: Segunda imagen de referencia a utilizar.

La segunda imagen a utilizar corresponde a la figura 4.7, que muestra las tres primeras letras del abecedario. Esta imagen tiene dimensiones de 600×213 pixeles, y fue escogida debido a la presencia tanto de bordes rectos como curvos, además de contar con 3 agujeros que deben ser representados en la geometría generada.

Figura 4.8: Tercera imagen de referencia a utilizar. Imagen de dominio público.

La tercera imagen a utilizar, que se muestra en la figura 4.8, es la imagen de una polilla. Esta imagen es de 796×522 pixeles y fue seleccionada por presentar detalles finos en las antenas y en el cuerpo, los cuales pueden ser desafiantes de capturar al generar una cantidad reducida de aristas.

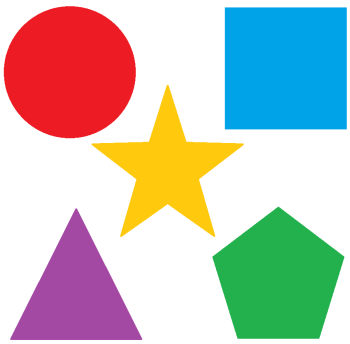

Figura 4.9: Cuarta imagen de referencia a utilizar.

Por último, la cuarta imagen a utilizar, mostrada en la figura 4.9, consta de una variedad de distintas figuras geométricas. Esta imagen tiene dimensiones de 900×900 pixeles y fue seleccionada para determinar si el tamaño de la imagen original afecta de alguna manera en los resultados obtenidos.

A continuación, se presentan los resultados obtenidos al ejecutar los algoritmos de detección de bordes en cada imagen, mostrando la cantidad de aristas generadas:

Método	Parám. modificado	Panda	ABC	Polilla	Figuras
Canny int. fijos	len = 10	477	241	289	539
	len = 20	254	128	154	283
	len = 40	131	68	80	146
Canny int. variables	maxdist = 1	283	98	187	106
	maxdist = 2	148	66	85	50
	maxdist = 5	87	47	45	38
Canny híbrido	len = 20 ; maxdist = 1	330	156	209	307
	len = 40 ; maxdist = 4	173	91	99	156
Triangulación	xy = 20	144	96	75	130
	xy = 25	171	108	99	158
	xy = 30	198	130	108	180

Tabla 4.5: Cantidad de aristas generadas para distintas variaciones de los algoritmos.

Como se puede observar, la cantidad de aristas generadas tiende a ser proporcional al tamaño de las imágenes de entrada, con valores más altos para la imagen del panda y las figuras geométricas. Sin embargo, se produce una excepción a esta regla en las variantes del método basado en Canny que utilizan eliminación de vértices a intervalos variables. En estos casos, las imágenes con mayor cantidad de bordes suelen ser aquellas con detalles más finos y menos líneas rectas, como las imágenes del panda y la polilla. Por lo tanto, se puede concluir que en estos casos, la eliminación de vértices en las líneas rectas conduce a reducciones significativas en la cantidad de bordes generados.

En cuanto a los resultados cualitativos, es importante destacar que se obtuvieron resultados óptimos para todas las imágenes de referencia al aplicar el método basado en el algoritmo de Canny, utilizando el algoritmo híbrido de eliminación de vértices. En estos casos, los bordes generados presentaban una alta similitud con las imágenes de referencia, capturando los detalles finos de manera precisa y sin generar una cantidad excesiva de vértices y aristas.

En el anexo se incluyen las imágenes correspondientes a los resultados obtenidos al aplicar la detección de bordes a las demás imágenes de referencia, aplicando todas las variaciones en parámetros previamente mencionadas.

4.1.4. Efectividad con respecto a la elaboración manual

A continuación, se compararán los resultados obtenidos con el uso de la herramienta con respecto a los bordes generados de forma manual, utilizando los pasos descritos en la sección 1.2. Para esto, se utilizará la imagen de referencia mostrada en la figura 4.10.

Figura 4.10: Figura utilizada como referencia, correspondiente al emblema del club Universidad de Chile. Imagen de dominio público.

En la figura 4.11 se pueden ver los bordes obtenidos mediante trazado manual y los obtenidos mediante el uso de la herramienta, utilizando los bordes obtenidos mediante el método basado en Canny con los parámetros por defecto. Es posible notar que en ambos casos la aproximación obtenida es visualmente similar a la imagen de referencia, sin una pérdida de detalle significativa en ninguno de los dos métodos. Sin embargo, se puede apreciar que para los bordes generados manualmente existe un gran rango de longitudes en las aristas generadas, donde la arista más larga es 10 veces mayor que la más corta. En contraste, para las aristas generadas con el uso de la herramienta, el rango de valores es mucho más acotado, con longitudes que van desde 5 a 24 pixeles.

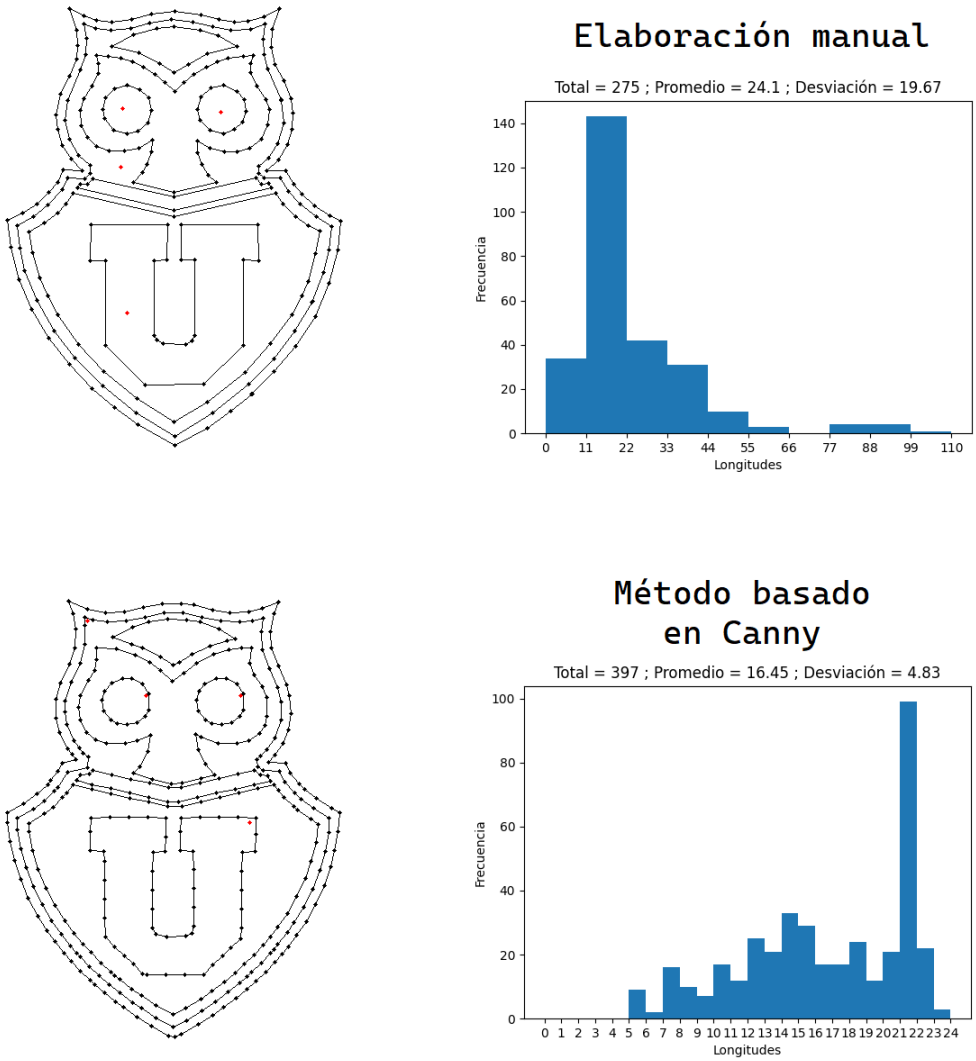

Figura 4.11: Bordes obtenidos mediante trazado manual, y mediante la utilización de la herramienta desarrollada.

Por lo tanto, es posible concluir que en términos de similitud visual con la imagen de referencia, la herramienta desarrollada logra resultados tan precisos como los obtenidos de forma manual, y además optimiza la distribución de las longitudes de las aristas, algo que es más difícil de lograr cuando se realiza el trazado de forma manual.

4.2. Eficiencia

4.2.1. Comparación de tiempos de ejecución

Ambos métodos de detección de bordes presentan diferencias significativas en términos de tiempo de ejecución. Mientras que el método basado en Canny tiene tiempos de ejecución del orden de segundos, el método basado en triangulaciones tiene tiempos de ejecución del orden de minutos, los cuales pueden variar según la cantidad de iteraciones del proceso y las dimensiones de la malla inicial. Sin embargo, es importante destacar que ambos métodos representan una mejora significativa con respecto a la elaboración manual, la cual, según los testimonios del profesor co-guía, podía llevar entre 6 a 8 horas.

Se llevó a cabo una comparación de los tiempos de ejecución de ambos métodos con las variaciones mencionadas en la sección anterior. Para ello, se emplearon las mismas 4 imágenes de referencia utilizadas en la sección 4.1.3, obteniéndose los resultados mostrados en la siguiente tabla:

Método	Parám. modificado	Panda	ABC	Polilla	Figuras
Canny int. fijos	len = 10	3.38s	0.91s	1.80s	2.60s
	len = 20	3.48s	0.91s	1.81s	2.60s
	len = 40	3.40s	0.90s	1.82s	2.60s
Canny int. variables	maxdist = 1	3.45s	0.94s	1.76s	2.73s
	maxdist = 2	3.51s	0.96s	1.81s	2.84s
	maxdist = 5	3.49s	0.99s	1.81s	2.94s
Canny híbrido	len = 20 ; maxdist = 1	3.41s	0.92s	1.72s	2.60s
	len = 40 ; maxdist = 4	3.44s	0.93s	1.80s	2.63s
Triangulación	xy = 20	57.09s	33.45s	44.18s	48.61s
	xy = 25	63.82s	40.07s	50.36s	53.52s
	xy = 30	73.38s	50.19s	61.85s	56.23s

Tabla 4.6: Tiempos de ejecución en segundos para distintas variaciones de los algoritmos.

Como se puede ver en la tabla 4.6, los tiempos de ejecución tienden a ser proporcionales a las dimensiones de las imágenes, sin importar el método utilizado. Sin embargo, se tiene que la imagen del panda siempre mostró tiempos de ejecución mayores que la imagen con figuras geométricas, a pesar del mayor tamaño de esta última. Por lo tanto, se puede concluir que las dimensiones de la imagen no son el único factor determinante para los tiempos de ejecución, ya que los contenidos de la imagen también influyen en los tiempos de procesamieto, siendo las imágenes con una mayor cantidad de curvas y detalles finos más complejas de procesar.

Por otro lado, es relevante destacar que el desempeño para las diferentes variantes del método basado en Canny es bastante constante, independientemente de las modificaciones en los parámetros. Por otro lado, en el algoritmo de triangulación, los tiempos de ejecución son siempre proporcionales a las dimensiones de la malla inicial. Es importante mencionar que, para este último algoritmo, en todos los casos se utilizaron 40 iteraciones, siendo el único parámetro modificado la cantidad de celdas en la malla inicial.

4.2.2. Complejidad del método basado en Canny

En cuanto al rendimiento de los algoritmos, el método basado en Canny inicia realizando el procesamiento de las imágenes usando el algoritmo de Canny. Esta operación tiene una complejidad $O(n)$, siendo n la cantidad de pixeles en la imagen ya que su tiempo de ejecución es proporcional a este valor. La generación de aristas sigue un patrón similar, ya que es necesario recorrer cada pixel en la imagen para generar las conexiones entre los puntos.

Una vez que se ha obtenido una lista de caminos generados, el tiempo de ejecución se vuelve dependiente de las características de la imagen procesada, ya que la cantidad de aristas generadas dependerá de dicha imagen. El procesamiento de las aristas y la generación de los caminos tiene una complejidad cuadrática, ya que para cada arista generada se realiza una comparación con todas las demás en busca de posibles conexiones.

Después de la generación de caminos, se aplican los algoritmos de eliminación de vértices, cuyo tiempo de ejecución depende de la cantidad de aristas en el camino, por lo que su complejidad es $O(n)$. Posteriormente, se realizan los pasos finales del procesamiento, que incluyen la fusión de caminos, el cierre de ciclos y la fusión de puntos cercanos. Los primeros dos algoritmos se aplican al listado de caminos, y tienen una complejidad cuadrática $O(n^2)$ y lineal $O(n)$ respectivamente, mientras que el último se aplica a cada camino individualmente y su complejidad depende de la cantidad de aristas en cada camino, lo que lo hace lineal $O(n)$.

En resumen, dado que la mayoría de los pasos del procesamiento se aplican al listado de aristas generado después del procesamiento de la imagen, y considerando que el procesamiento sobre los pixeles se lleva a cabo principalmente en los pasos iniciales, se concluye que el tiempo de ejecución del algoritmo depende más de las características de la imagen misma y de los bordes detectados en ella que de su tamaño.

4.2.3. Complejidad del método basado en triangulaciones

En cuanto al tiempo de procesamiento del algoritmo basado en triangulaciones, todos los pasos del proceso de refinamiento, optimización y simplificación de la malla presentan una complejidad lineal. Esto se debe a que cada uno de estos pasos implica recorrer una lista de elementos presentes en la malla para evaluar el cumplimiento de diferentes condiciones, sin la necesidad de realizar comparaciones con los demás elementos de la lista. Por lo tanto, el tiempo de ejecución dependerá únicamente de la cantidad de elementos contenidos en los arreglos de vértices, aristas y triángulos.

Por otro lado, para los triángulos, el cálculo de los puntos contenidos en estos es proporcional a la diferencia entre su punto más alto y el punto más bajo, según se describe en el algoritmo de la sección 2.6.2. Asimismo, el cálculo del color promedio de los pixeles contenidos en los triángulos también es lineal, ya que depende de la cantidad de pixeles en su interior.

En resumen, se puede observar que no hay ningún paso durante este proceso que tenga una complejidad cuadrática. La mayoría de los pasos dependen de la cantidad de elementos

presentes en la malla generada, donde la inserción y eliminación de elementos en esta dependen de las características de la imagen de referencia. Sin embargo, el usuario tiene cierto grado de control sobre estos valores mediante los parámetros ingresados, ya que esto hace posible reducir la cantidad de elementos iniciales y disminuir las inserciones de nuevos elementos en la malla. Por lo tanto, se puede concluir que una triangulación más gruesa potencialmente requerirá menos recursos que una triangulación generada a partir de una malla más densa.

4.3. Limitaciones

Una de las principales limitaciones de ambos métodos de obtención de bordes es la necesidad de que las imágenes tengan un fondo blanco, ya que esto reduce el conjunto de imágenes en las que se puede usar la herramienta. Sin embargo, es posible superar esta limitación mediante el uso de software de edición de imágenes. No obstante, esto agregaría un paso adicional al procesamiento necesario para generar los bordes.

En cuanto al método basado en el algoritmo de Canny, se han observado casos en los que las imágenes en formato .jpg no pueden ser procesadas de manera óptima. A pesar del procesamiento previo realizado, los algoritmos de generación y reducción de aristas captan el ruido presente en estas imágenes, lo que resulta en pérdidas de continuidad en los bordes y errores en el procesamiento. Por otro lado, esto es una de las fortalezas del método basado en triangulaciones, ya que es robusto al ruido en las imágenes debido a que los pixeles individuales tienen un impacto mínimo en el cálculo del color promedio de los triángulos.

Una limitación del método basado en triangulación es que la restricción del tamaño mínimo de las aristas afecta la calidad de las aproximaciones logradas. Esto se debe a que cuando una arista alcanza una longitud menor a este límite, se realiza un *half-edge-collapse*, lo que implica eliminar aristas y vértices en la malla. Esta eliminación de elementos disminuye la calidad de las aproximaciones realizadas, y cualquier intento posterior de insertar puntos para contrarrestar estos cambios se deshace inmediatamente debido a las restricciones impuestas a la longitud de las aristas.

Capítulo 5

Conclusión

En este trabajo de memoria se logró desarrollar una herramienta eficiente para la detección de bordes en imágenes y la generación de resultados en un formato adecuado para la creación de mallas de polígonos. Mediante dos métodos distintos para la detección de estos bordes, se brindaron alternativas al usuario para adaptarse a las necesidades específicas de cada imagen.

El primer método, basado en el algoritmo de Canny, permitió la obtención de los bordes de una imagen como un listado de aristas y vértices, presentando resultados altamente similares a las imágenes de referencia. Su diseño final proporciona al usuario un gran control sobre las características de la geometría generada, ofreciendo una gran variedad de parámetros que pueden ser modificados según las características específicas de los bordes que se desean generar.

De la misma manera, se implementó exitosamente un método que permite extraer bordes mediante el uso de mallas de triángulos. Aunque este método no haya mostrado mejoras sustanciales en términos de calidad y eficiencia en comparación con el primer método, su implementación abre la oportunidad para el desarrollo futuro de una herramienta que no solo extraiga los bordes de la imagen, sino que también genere triangulaciones de Delaunay al interior de las figuras, permitiendo su uso de forma directa con el mallador Polylla.

Finalmente, se desarrolló un programa capaz de recibir los bordes detectados por ambos métodos previamente descritos y transformarlos a un formato de archivo adecuado para la posterior generación de triangulaciones, en concreto el formato .poly. Al ejecutar este programa, se logra una reducción significativa en el tiempo necesario para obtener estos archivos, pasando de un proceso de horas a solo segundos.

Además, se realizó una comparación exhaustiva de los resultados obtenidos, evaluando la similitud de los bordes detectados con los bordes de referencia y analizando la distribución de las longitudes de las aristas. A partir de estos análisis, se logró identificar para cada método una combinación de parámetros ideales que permitiesen la obtención de bordes con una alta precisión. También se realizaron comparaciones de eficiencia, explorando las variaciones en los tiempos de ejecución según los parámetros ingresados y las características de las imágenes utilizadas.

En conclusión, los objetivos propuestos fueron cumplidos de forma satisfactoria, al proporcionarse una herramienta que simplifica y acelera el proceso de creación de geometrías mediante la detección de bordes en imágenes. Asimismo, se ha sentado una base para futuras investigaciones y mejoras, en particular la generación directa de triangulaciones de Delaunay.

5.1. Trabajo Futuro

Una de las principales modificaciones que se pueden realizar en la herramienta desarrollada es implementarla utilizando un lenguaje compilado. Esto se debe a que el uso de un lenguaje como Python resulta en tiempos de ejecución más prolongados. En el caso del método basado en triangulaciones, esto implica tiempos de espera de aproximadamente un minuto para obtener los resultados. Al utilizar un lenguaje compilado como C++, se podrían aprovechar de manera más directa las capacidades de la GPU y permitir la paralelización de tareas, lo que reduciría significativamente los tiempos de ejecución.

Otra sugerencia para el desarrollo futuro es la implementación de una interfaz gráfica, lo cual resulta especialmente beneficioso para un algoritmo basado en la triangulación de imágenes. Al permitirse la visualización del proceso de triangulación en todas sus iteraciones, se puede obtener una comprensión más clara y efectiva de las transformaciones que se llevan a cabo en cada paso del algoritmo. Además, facilitaría la identificación y corrección de posibles errores. Junto con esto, una interfaz gráfica también brindaría la oportunidad de realizar modificaciones manuales en la imagen de manera intuitiva, permitiendo al usuario modificar elementos de la malla generada según sus necesidades.

Una de las principales limitaciones de la herramienta desarrollada es su dependencia de que las imágenes tengan un fondo blanco para obtener resultados adecuados. Sin embargo, esta limitación se puede superar fácilmente mediante el uso de software de edición de imágenes, que permite modificar el color de áreas no deseadas. Este paso preliminar es más rápido y sencillo en comparación con el procesamiento realizado sin la herramienta. No obstante, sería recomendable considerar la incorporación de esta funcionalidad directamente en la herramienta, eliminando así la necesidad de depender de software externo para el procesamiento de las imágenes y evitando la adición de pasos adicionales al flujo de trabajo.

A pesar de que los resultados obtenidos con el método basado en triangulación no presentan ventajas evidentes en comparación con el algoritmo basado en Canny, existe la posibilidad de utilizar la implementación basada en triangulaciones para prescindir de otro paso en el proceso de creación de inputs para Polylla, en concreto la generación de triangulaciones de Delaunay a partir de los bordes detectados.

Actualmente, los bordes generados no son entregados de forma directa a Polylla; en su lugar, es necesario realizar una triangulación mediante el software Triangle. Sin embargo, existe la posibilidad de obviar este paso y utilizar las triangulaciones generadas por la herramienta directamente. Esto requeriría llevar a cabo modificaciones en el código para asegurar que las triangulaciones generadas dentro de la figura tengan las características de las triangulaciones de Delaunay, ya que la prioridad actual es la aproximación de los bordes de la imagen por sobre la generación de una triangulación que cumpla con estas condiciones.

Bibliografía

[1] Mohd Ansari, Diksha Kurchaniya, and Manish Dixit. A comprehensive analysis of image edge detection techniques. *International Journal of Multimedia and Ubiquitous Engineering*, 12:1–12, 11 2017.

[2] M. Botsch, L. Kobbelt, M. Pauly, P. Alliez, and B. Levy. *Polygon Mesh Processing*. CRC Press, 2010.

[3] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[4] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8:679 – 698, 12 1986.

[5] Alex Clark. Pillow (pil fork) documentation, 2015.

[6] J. De Loera, J. Rambau, and F. Santos. *Triangulations: Structures for Algorithms and Applications*. Algorithms and Computation in Mathematics. Springer Berlin Heidelberg, 2010.

[7] GIMP. Paths tool. `https://docs.gimp.org/en/gimp-tool-path.html`.

[8] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*. Elsevier, 1992.

[9] Panda PNG Image. `https://www.pngmart.com/image/343994`.

[10] Kai Lawonn and Tobias Günther. Stylized image triangulation. *Computer Graphics Forum*, 38(1):221–234, 2019.

[11] David E. Muller and Franco P. Preparata. Finding the Intersection of Two Convex Polyhedra. *Theoretical Computer Science*, 7(2):217–236, 1978.

[12] Sergio P. Salinas. Polylla-mesh. `https://github.com/ssalinasfe/Polylla-Mesh-DCEL`.

[13] Caroline A. Schneider, Wayne S. Rasband, and Kevin W. Eliceiri. NIH Image to ImageJ: 25 years of image analysis. *Nature Methods*, 9(7):671–675, 2012.

[14] Alejandro Ortiz y Hang Si Sergio Salinas, Nancy Hitschfeld. Polylla: polygonal meshing algorithm based on terminal-edge regions. *Engineering with Computers*, 2022.

[15] Jonathan Richard Shewchuk. Triangle: A two-dimensional quality mesh generator and delaunay triangulator. `https://www.cs.cmu.edu/~quake/triangle.html`.

[16] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[17] Irwin Sobel. An isotropic 3x3 image gradient operator. *Presentation at Stanford A.I. Project 1968*, 02 2014.

Anexo

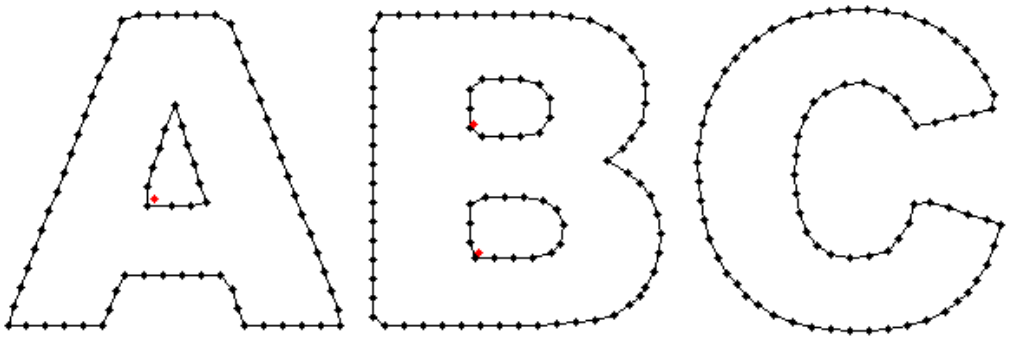

Figura 1: Resultado de aplicar el método basado en Canny con eliminación a intervalos fijos sobre la segunda imagen de referencia, con len = 10.

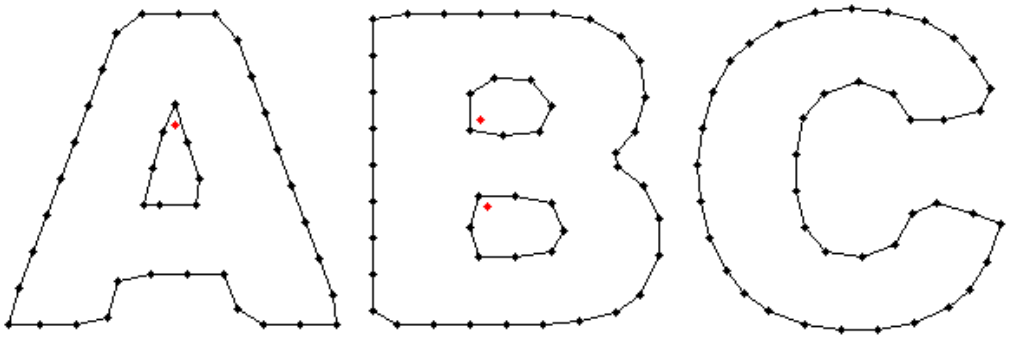

Figura 2: Resultado de aplicar el método basado en Canny con eliminación a intervalos fijos sobre la segunda imagen de referencia, con len = 20.

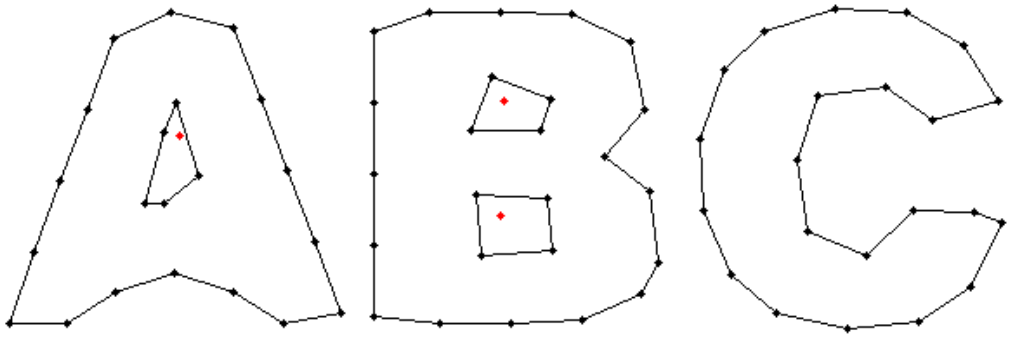

Figura 3: Resultado de aplicar el método basado en Canny con eliminación a intervalos fijos sobre la segunda imagen de referencia, con len = 40.

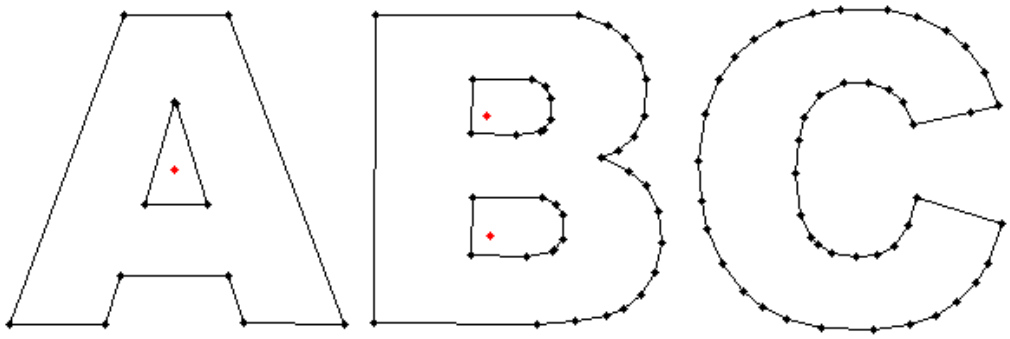

Figura 4: Resultado de aplicar el método basado en Canny con eliminación a intervalos variables sobre la segunda imagen de referencia, con maxdist = 1.

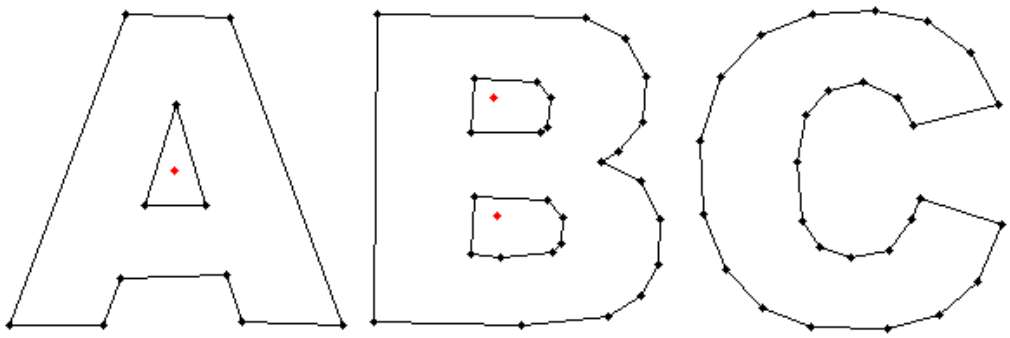

Figura 5: Resultado de aplicar el método basado en Canny con eliminación a intervalos variables sobre la segunda imagen de referencia, con maxdist = 2.

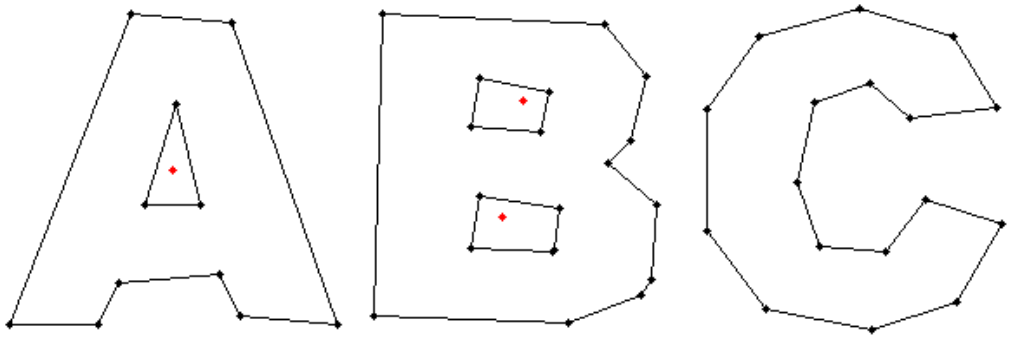

Figura 6: Resultado de aplicar el método basado en Canny con eliminación a intervalos variables sobre la segunda imagen de referencia, con maxdist = 5.

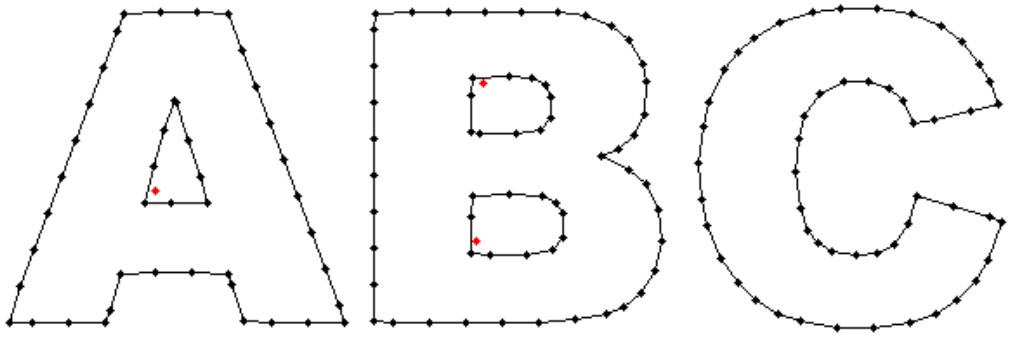

Figura 7: Resultado de aplicar el método basado en Canny con eliminación híbrida sobre la segunda imagen de referencia, con len = 20 y maxdist = 1.

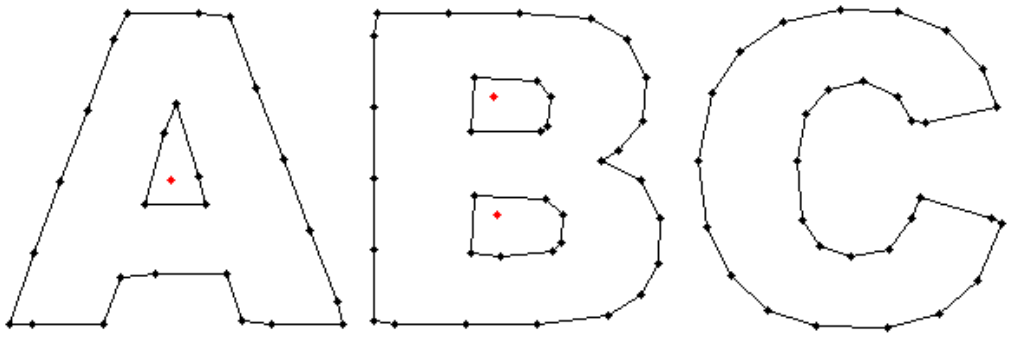

Figura 8: Resultado de aplicar el método basado en Canny con eliminación híbrida sobre la segunda imagen de referencia, con len = 40 y maxdist = 2.

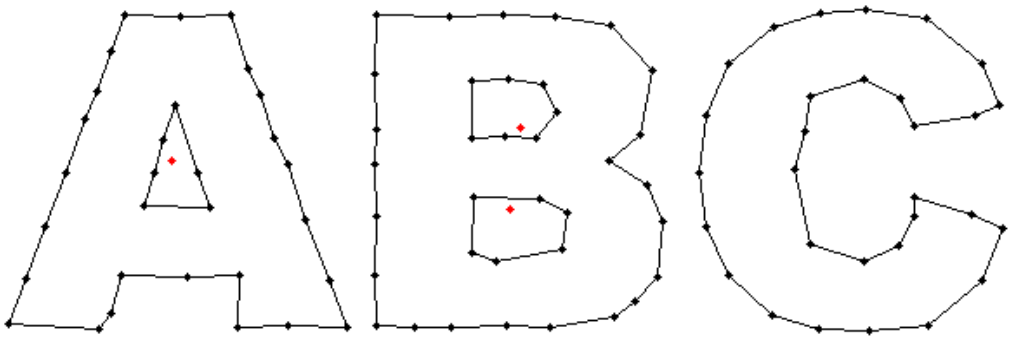

Figura 9: Resultado de aplicar el método basado en triangulaciones sobre la segunda imagen de referencia, con malla incial de 20×20.

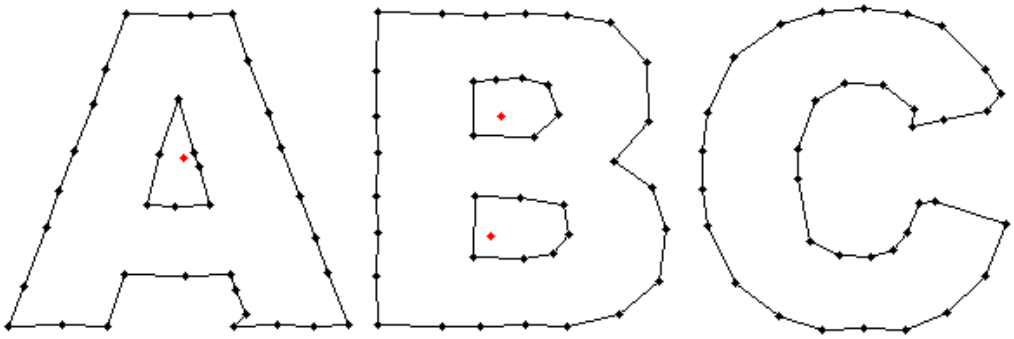

Figura 10: Resultado de aplicar el método basado en triangulaciones sobre la segunda imagen de referencia, con malla incial de 25×25.

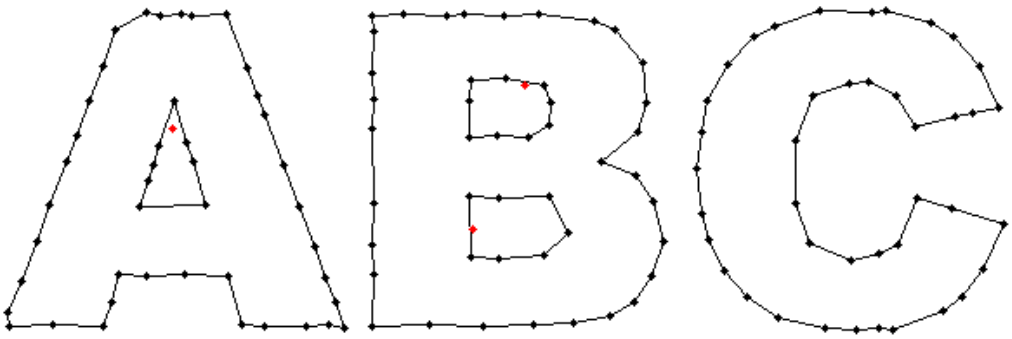

Figura 11: Resultado de aplicar el método basado en triangulaciones sobre la segunda imagen de referencia, con malla incial de 30×30.

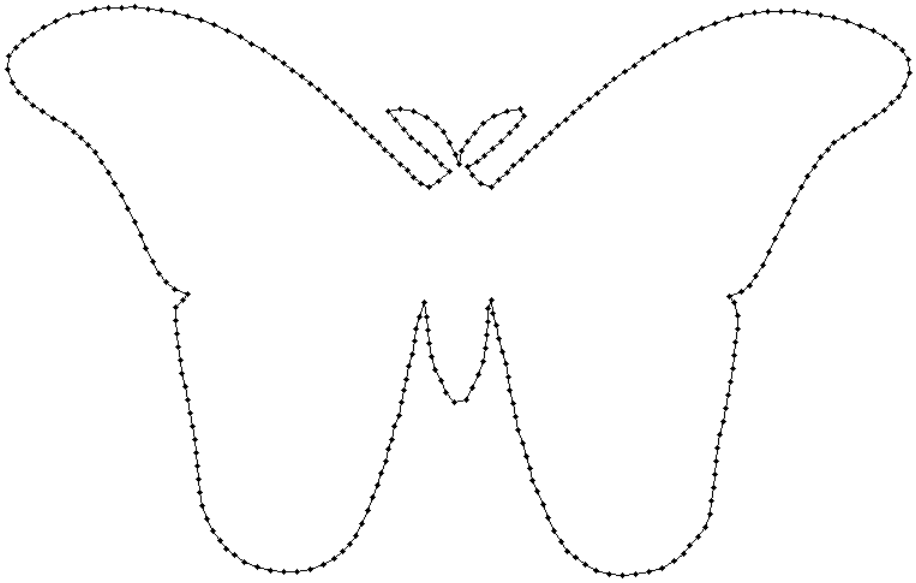

Figura 12: Resultado de aplicar el método basado en Canny con eliminación a intervalos fijos sobre la tercera imagen de referencia, con len = 10.

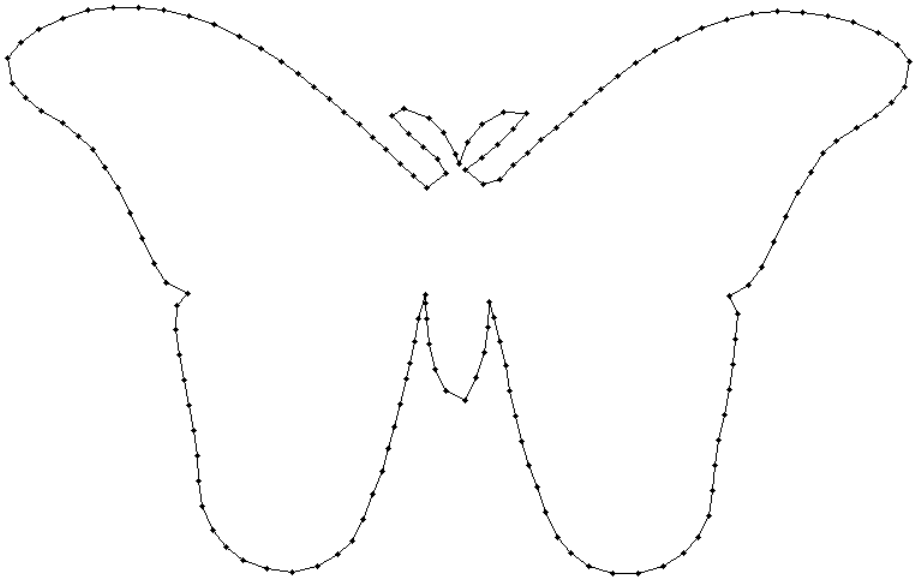

Figura 13: Resultado de aplicar el método basado en Canny con eliminación a intervalos fijos sobre la tercera imagen de referencia, con len = 20.

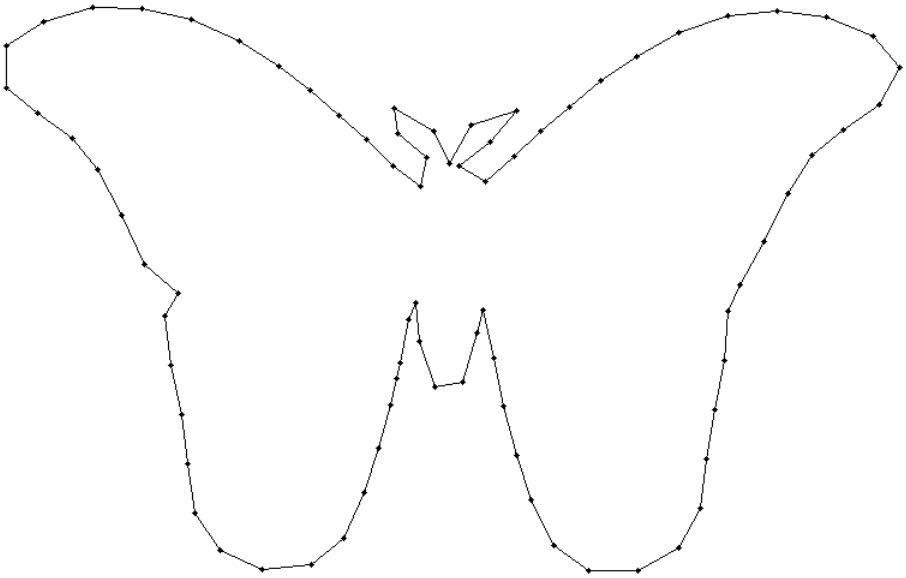

Figura 14: Resultado de aplicar el método basado en Canny con eliminación a intervalos fijos sobre la tercera imagen de referencia, con len = 40.

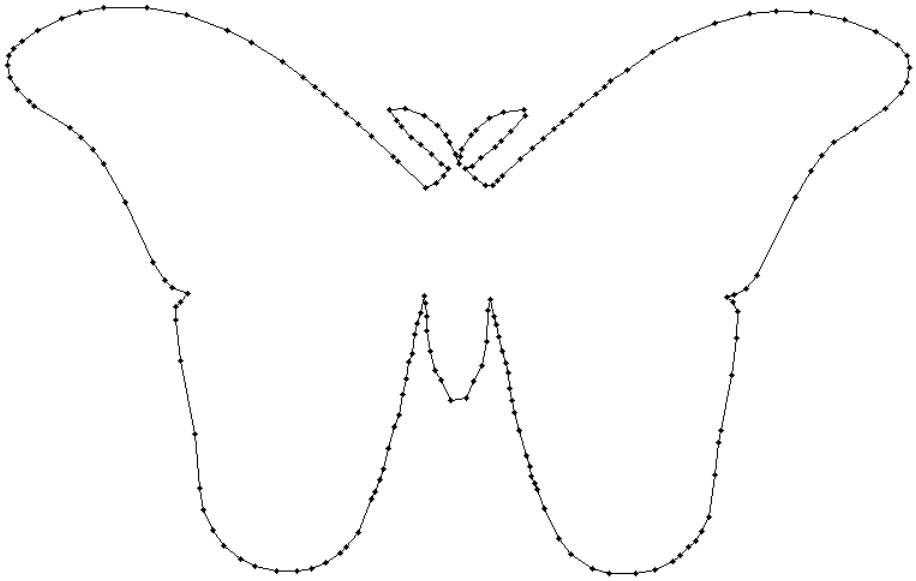

Figura 15: Resultado de aplicar el método basado en Canny con eliminación a intervalos variables sobre la tercera imagen de referencia, con maxdist = 1.

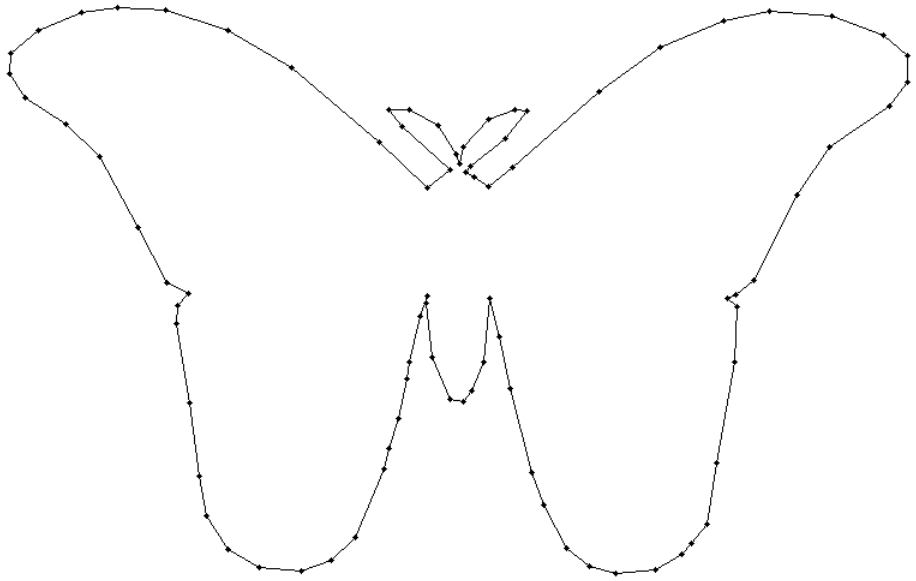

Figura 16: Resultado de aplicar el método basado en Canny con eliminación a intervalos variables sobre la tercera imagen de referencia, con maxdist = 2.

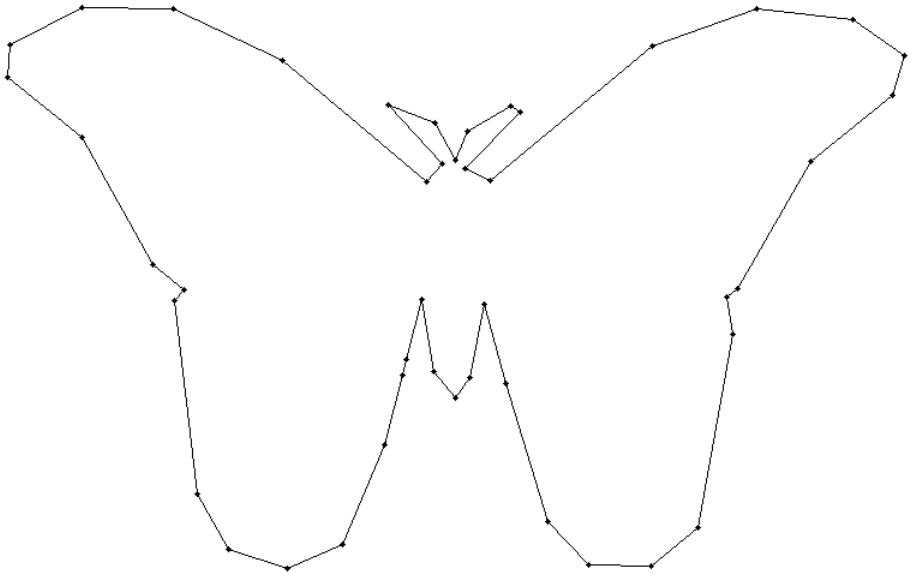

Figura 17: Resultado de aplicar el método basado en Canny con eliminación a intervalos variables sobre la tercera imagen de referencia, con maxdist = 5.

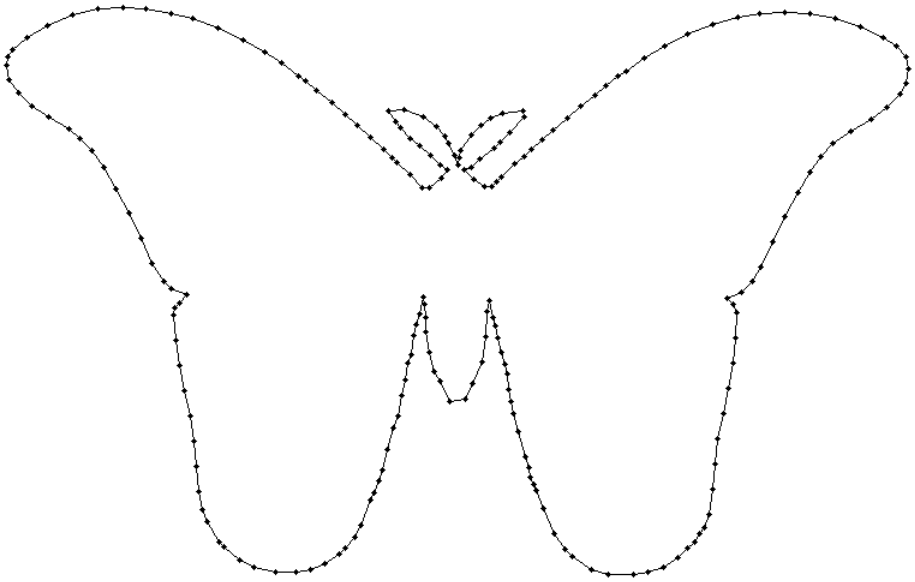

Figura 18: Resultado de aplicar el método basado en Canny con eliminación híbrida sobre la tercera imagen de referencia, con len = 20 y maxdist = 1.

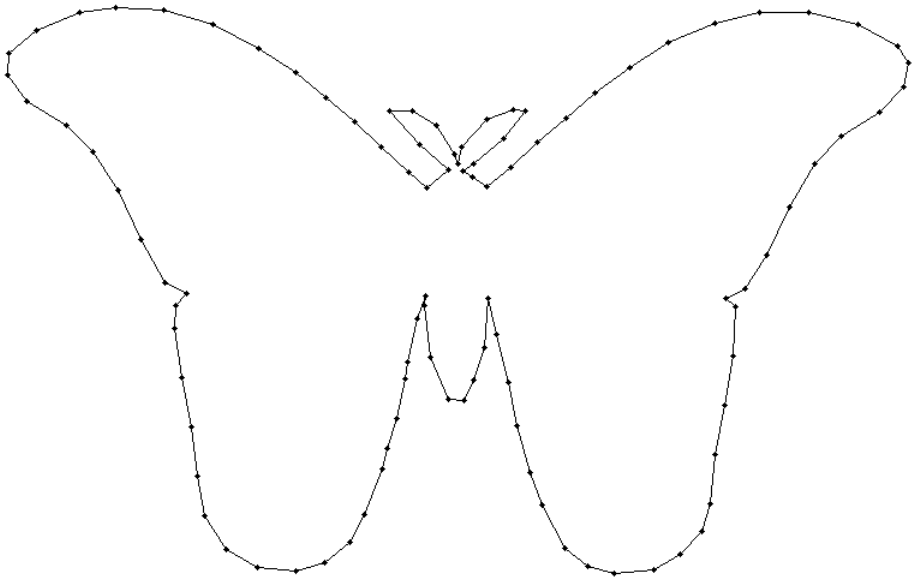

Figura 19: Resultado de aplicar el método basado en Canny con eliminación híbrida sobre la tercera imagen de referencia, con len = 40 y maxdist = 2.

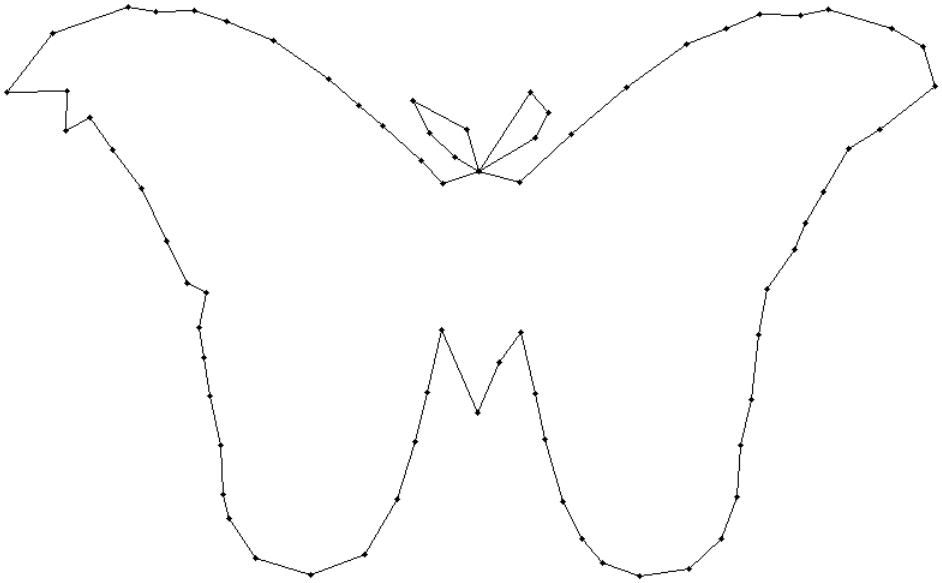

Figura 20: Resultado de aplicar el método basado en triangulaciones sobre la tercera imagen de referencia, con malla incial de 20×20.

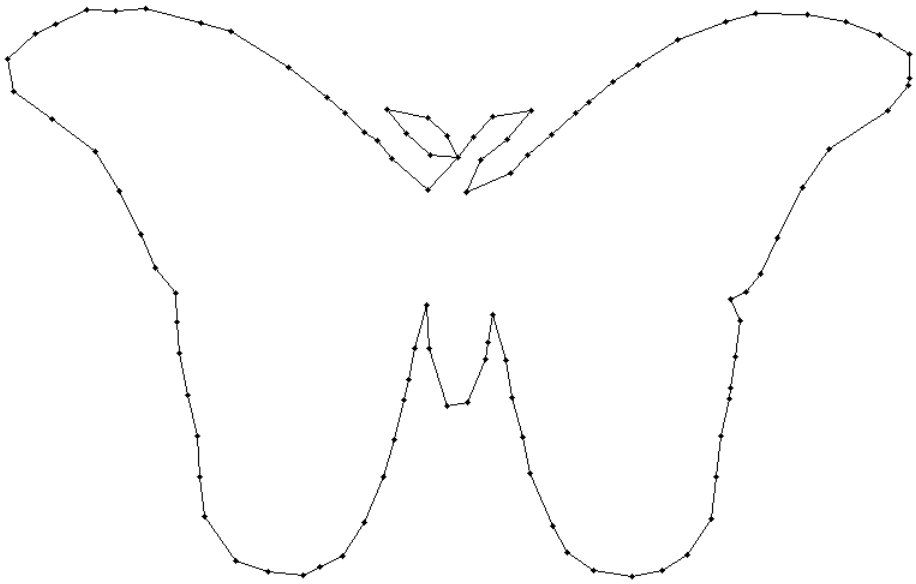

Figura 21: Resultado de aplicar el método basado en triangulaciones sobre la tercera imagen de referencia, con malla incial de 25×25.

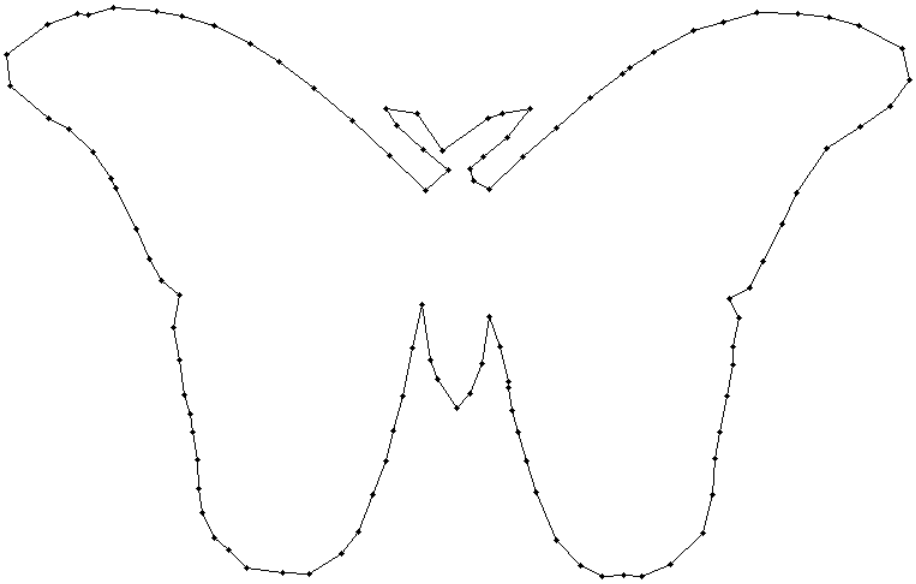

Figura 22: Resultado de aplicar el método basado en triangulaciones sobre la tercera imagen de referencia, con malla incial de 30×30.

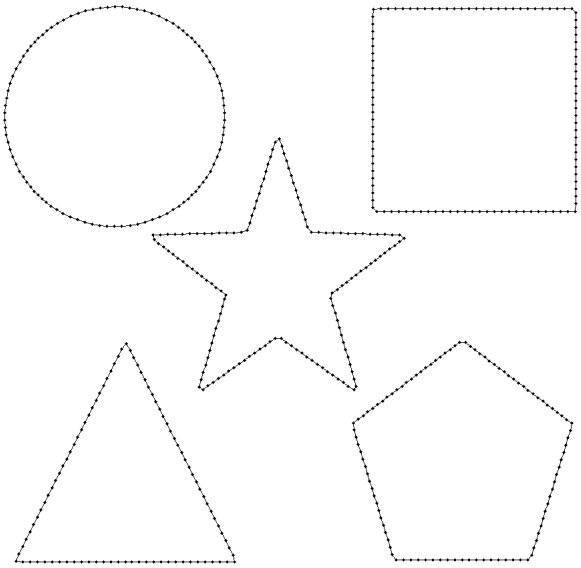

Figura 23: Resultado de aplicar el método basado en Canny con eliminación a intervalos fijos sobre la cuarta imagen de referencia, con len = 10.

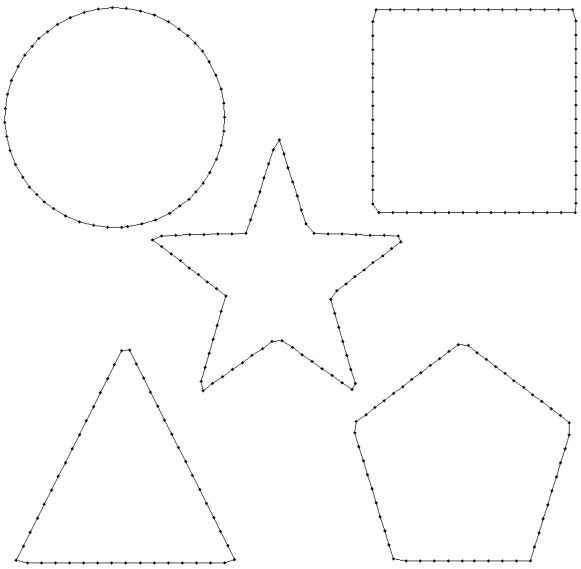

Figura 24: Resultado de aplicar el método basado en Canny con eliminación a intervalos fijos sobre la cuarta imagen de referencia, con len = 20.

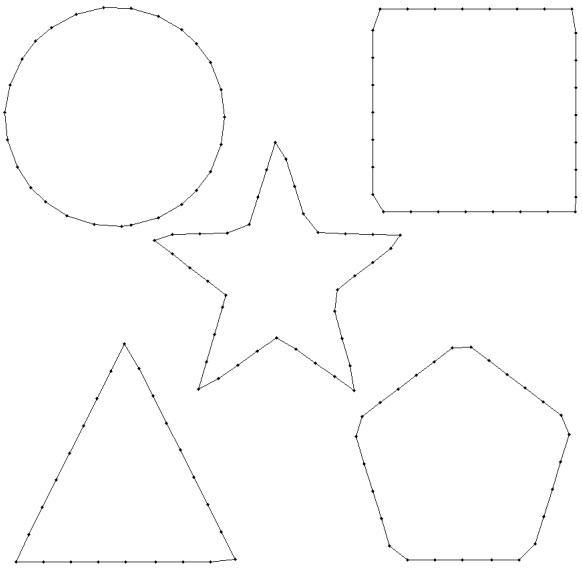

Figura 25: Resultado de aplicar el método basado en Canny con eliminación a intervalos fijos sobre la cuarta imagen de referencia, con len = 40.

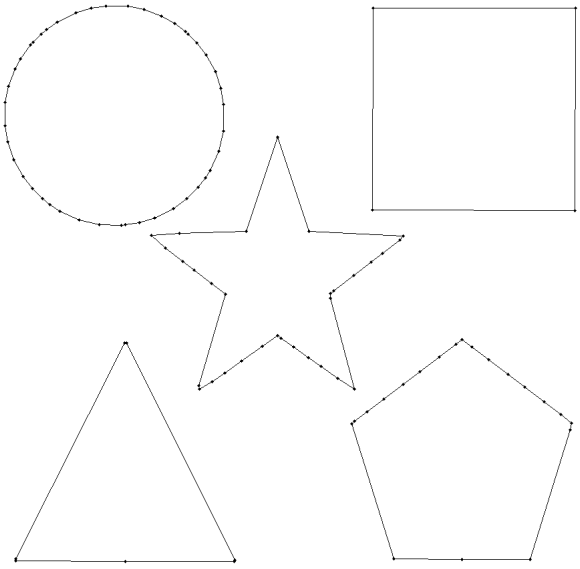

Figura 26: Resultado de aplicar el método basado en Canny con eliminación a intervalos variables sobre la cuarta imagen de referencia, con maxdist = 1.

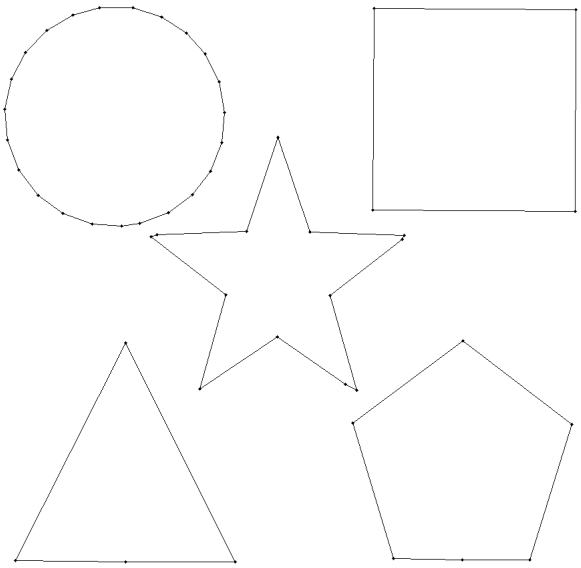

Figura 27: Resultado de aplicar el método basado en Canny con eliminación a intervalos variables sobre la cuarta imagen de referencia, con maxdist = 2.

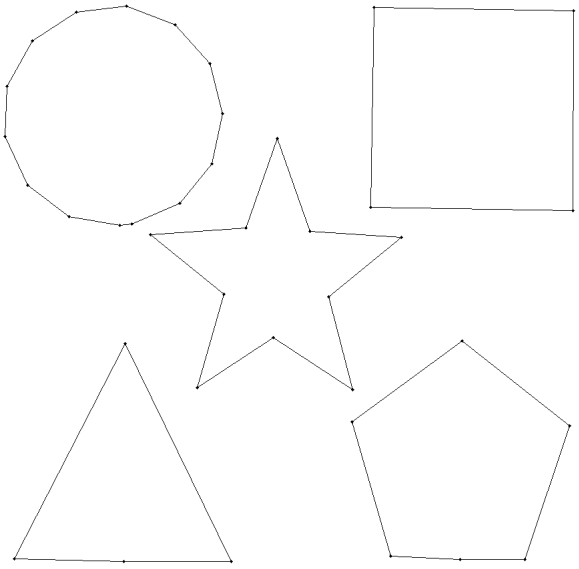

Figura 28: Resultado de aplicar el método basado en Canny con eliminación a intervalos variables sobre la cuarta imagen de referencia, con maxdist = 5.

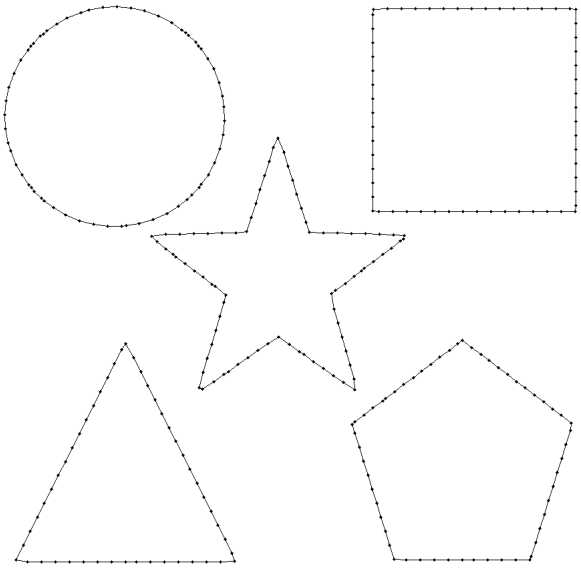

Figura 29: Resultado de aplicar el método basado en Canny con eliminación híbrida sobre la cuarta imagen de referencia, con len = 20 y maxdist = 1.

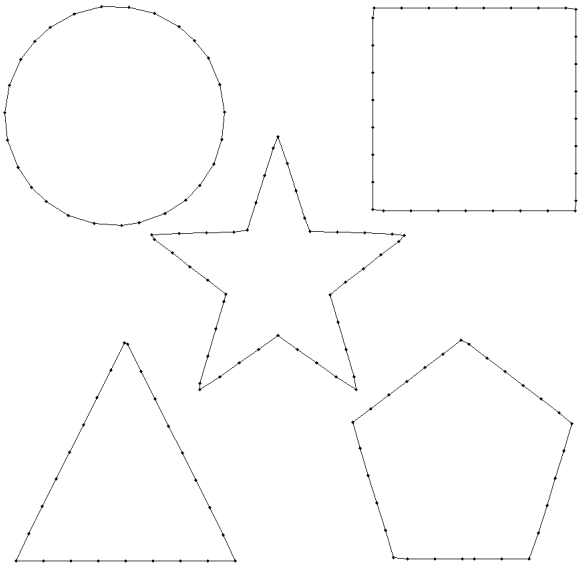

Figura 30: Resultado de aplicar el método basado en Canny con eliminación híbrida sobre la cuarta imagen de referencia, con len = 40 y maxdist = 2.

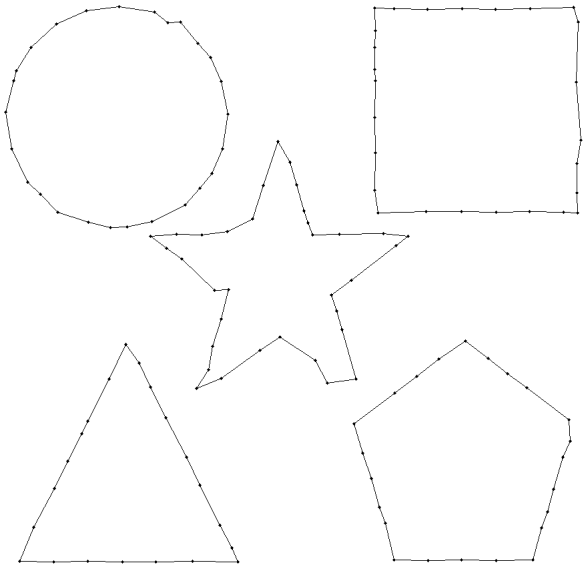

Figura 31: Resultado de aplicar el método basado en triangulaciones sobre la cuarta imagen de referencia, con malla incial de 20×20.

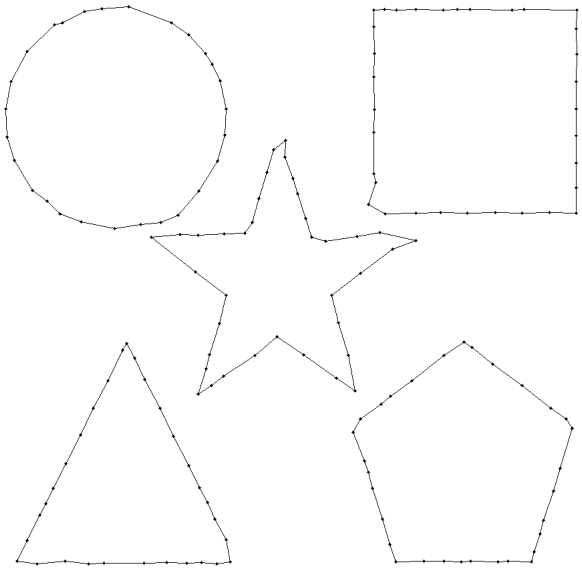

Figura 32: Resultado de aplicar el método basado en triangulaciones sobre la cuarta imagen de referencia, con malla incial de 25×25.

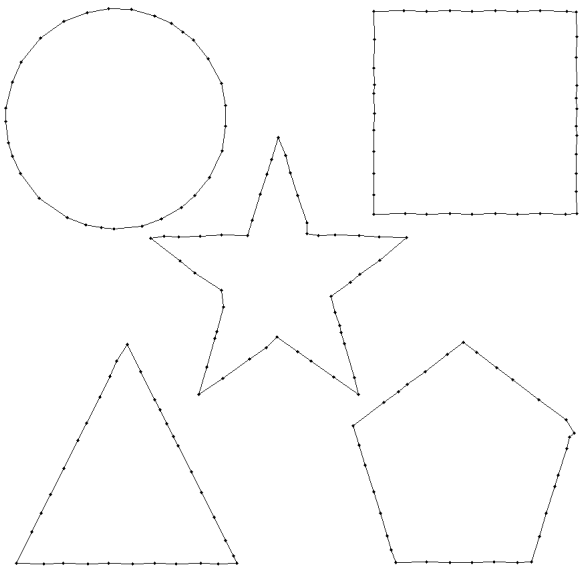

Figura 33: Resultado de aplicar el método basado en triangulaciones sobre la cuarta imagen de referencia, con malla incial de 30×30.