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

DETECCIÓN, CONSTRUCCIÓN Y EVALUACIÓN DE POLÍGONOS NO SIMPLES, A PARTIR DE IMÁGENES

PROPUESTA DE TEMA DE MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

AMÉRICO IGNACIO FERRADA DÍAZ

PROFESOR GUÍA:
NANCY HITSCHFELD

SANTIAGO DE CHILE
2019

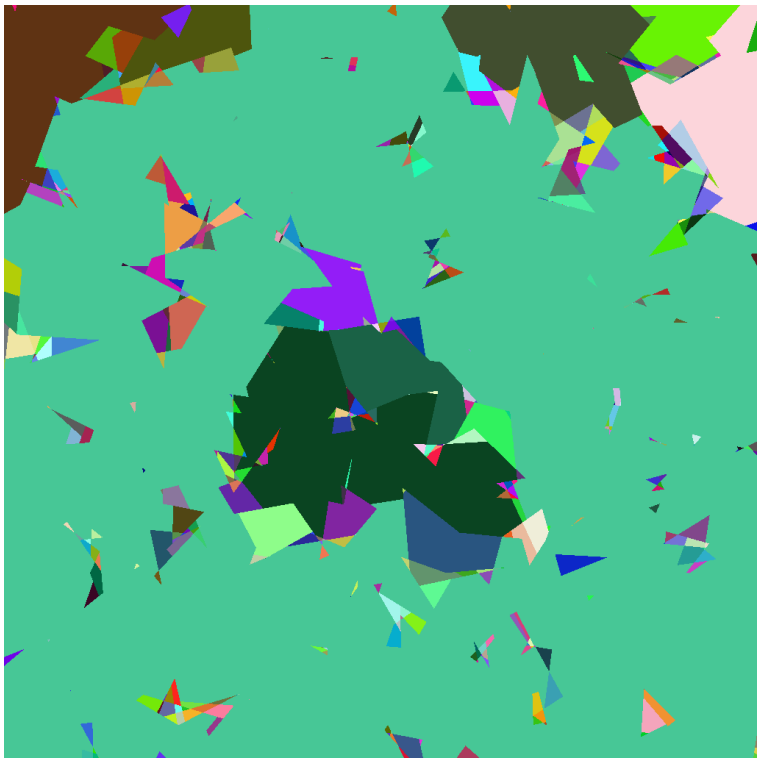
1. Introducción

El presente trabajo de memoria se sitúa en el contexto del diseño de algoritmos eficientes y robustos para simular fracturas producidas en rocas durante procesos de extracción minera y determinar su impacto en la estabilidad de las rocas restantes. Las fracturas pueden producir que ciertas rocas queden de manera muy inestable, apoyadas unas con otras. Esto es muy peligroso pues se pueden producir derrumbes, afectar a personas y a la infraestructura minera, en general.

Para estudiar el problema real (en tres dimensiones), se está desarrollando un sistema computacional para modelar las fallas/fracturas en una roca y qué tipo (forma) de rocas más pequeñas se puede producir. En el ámbito de la geometría computacional, este problema se reduce a calcular la forma y volumen de poliedros (sólido cerrado, definido por caras (polígonos), arcos (segmentos) y vértices(puntos)) formado por un número finito de planos. El poliedro resultante puede tener cualquier forma, arcos y caras colgantes, en el interior.

Antes de comenzar a abordar el problema en tres dimensiones, y como una forma entenderlo primero en dos dimensiones, se desarrolló el algoritmo *Painter's Algorithm* que recibe como entrada un conjunto de segmentos, cada uno define una fractura en 2D y genera como salida una imagen. En la imagen generada los colores distintos identifican un polígono diferente (Ver Figura 1). Para evitar el cálculo de intersecciones de segmentos, aprovecha el coloreo de la pantalla y generar las regiones que representan los polígonos. Este se implementó en paralelo, en GPU.

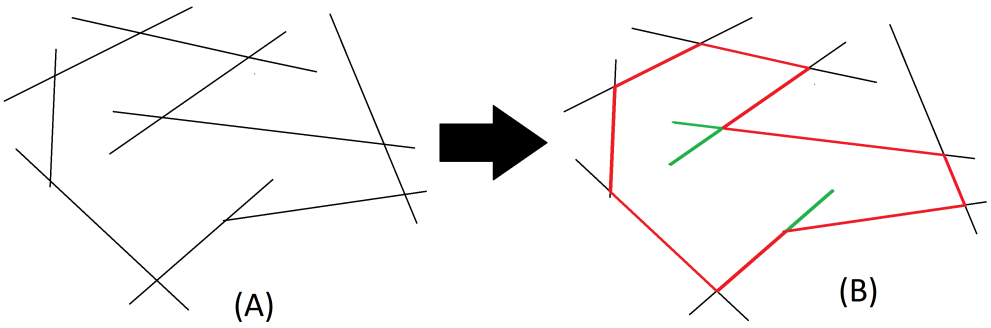
Figura 1: Ejemplo de imagen 2D resultante de la intersección de segmentos. Cada color es un polígono distinto.



Como se observa en la Figura 1, la forma de los polígonos resultantes puede ser muy diversa y compleja: pueden existir polígonos dentro de otro, segmentos colgantes y vértices aislados. Debido a esto el algoritmo no calcula explícitamente las intersecciones, sino que genera una imagen que representa parte de la solución al problema. Esta solución evita los errores de precisión y casos especiales, muy comunes en los algoritmos de geometría computacional.

Es así como surge el problema a resolver en esta memoria, que consiste en considerar como entrada una imagen, como la mostrada en Figura 1, en conjunto con una lista de segmentos iniciales (fracturas) y transformarlas en una estructura interpretable de polígonos(cortes de roca) y segmentos(fracturas), primero para evaluar su forma y luego para almacenarlos en un formato especifico de geometría vectorial. Este formato es requerido para su uso posterior en simulaciones numéricas fuera del alcance de esta memoria. Un ejemplo de salida lo podemos ver en Figura 2.

Figura 2: (A) Representación de líneas rasterizadas. (B) Resultado esperado: polígono en rojo, fracturas internas en verde.



Un polígono es una estructura geométrica en dos dimensiones que demarca un área finita y conexa en el espacio. Un polígono simple es una polilínea cerrada, cuyos segmentos no consecutivos no se intersectan. Por otra parte, las fracturas son una estructura que definimos en este proyecto. Esta se representa con un segmento que corta la superficie interna de un polígono pero que no logra dividir su superficie completamente. En el caso de esta memoria se trabajará solo con polígonos formados por segmentos rectos, sin segmentos curvos.

Este trabajo tiene además el objetivo de extraer propiedades geométricas de las estructuras calculadas, donde por ejemplo se requiere analizar la proporción entre el área de la cerradura convexa de los polígonos y el área del mismo. La cerradura convexa corresponde a una figura matemática que representa el polígono convexo más pequeño que envuelve otro. Eso es requerido, volviendo al problema original, para estudiar la estabilidad de un corte en la roca que se está modelando.

El resultado de este trabajo se planea utilizar como entrada de un generador de mallas de triángulos. Un ejemplo es Triangle [2] un software que lee como entrada un conjunto de polígonos, segmentos y puntos, tales como los calculados en esta memoria, y los transforma en un conjunto de triángulos, que se pueden utilizar en simulaciones numéricas.

Dentro de este problema es importante resaltar la necesidad de que la aplicación a desa-

rrollar funcione de forma eficiente y robusta, ya que este modelo se plantea extender a tres dimensiones. Para lograr esta eficiencia tenemos que tener en cuenta el costo computacional de calcular intersecciones. Para calcular la intersección entre dos segmentos se requiere solucionar una ecuación de primer grado y luego revisar si la solución de esta ecuación pertenece al segmento a comparar o no. Este procedimiento no es tan costoso en orden computacional. El problema depende del número de intersecciones que se requiere calcular que en el peor de los casos es $O(n^2)$ con n el numero de segmentos. Por otra parte se plantea evaluar la factibilidad de que la solución sea paralelizable en dispositivo de procesamiento gráfico.

2. Estado del Arte

A pesar de que existe trabajo relacionado similar, el problema abordado por esta memoria no ha sido desarrollado completamente. Las soluciones existentes para detectar polígonos, trabajan calculando intersecciones entre aristas, lo cual funciona bien para en aplicaciones donde se conoce la geometría esperada. Sin embargo, los algoritmos que utilizan cálculos de intersecciones sufren problemas de precisión y casos especiales, necesitando en general de soluciones complejas y propensas a errores.

Un ejemplo de solución, que solo considera los segmentos y no la imagen, puede ser la división del espacio en Quadtree en función de la densidad de segmentos que aparece en cada cuadrante. El Quadtree es una división del espacio en forma de árbol con el objetivo de generar una división homogénea del espacio, y luego para los segmentos cercanos, calcular sus intersecciones y almacenar los puntos de intersección. Finalmente recorrer los nuevos segmentos calculados, ignorando las fracturas(segmentos interiores), hasta contruir los polígonos.

El algoritmo anterior es una propuesta de solución que puede permitir la detección y construcción de los polígonos, siempre y cuando se pueda mostrar que el uso de un Quadtree nos permita detectar los segmentos requeridos para obtener la geometría deseada. Esta solución puede tener dos problemas: el primero, en el calculo de intersecciones existen problemas de precisión; por un error de punto flotante se puede llegar a cambiar la geometría esperada afectando la robustez de la solución. Esta es importante en el análisis de esta memoria. Otra consideración es que este algoritmo pueda escalar bien a tres dimensiones en 3D las fracturas se representan por planos y el quadtree es un octree.

Otro requerimiento es tener en cuenta que la geometría esperada no es del tipo estándar; pueden existir polígonos dentro de otro, segmentos y puntos aislados, lo cual algoritmos tradicionales no consideraran y generan errores. Un algoritmo interesante a tener en cuenta es el descrito en [1]; este resuelve un problema similar, pero requiere contar con los puntos de las intersección.

3. Objetivos

Objetivo General

Generar una aplicación que permita detectar, construir y evaluar cortes de roca, representados por polígonos no simples y fracturas. La aplicación debe recibir como input una imagen y un conjunto de segmentos que representas las fracturas iniciales, y entregar como resultado un conjunto de vértices(puntos), arcos(segmentos) y polígonos que representas la geometría contenida de la imagen. Además se debe permitir calcular propiedades sobre las estructuras generadas tales como la convexidad de los polígonos. La información se almacenará en un archivo de salida. Este debe estar en formato *planar straight line graph* (pslg) [1], para poder especificar polígonos y segmentos (fracturas) dentro de los polígonos.

Objetivos Específicos

Las metas esperadas dentro de el desarrollo de la memoria son:

1. Desarrollar un algoritmo que permita construir los polígonos(rocas) y detectar las segmentos (fracturas) desde una imagen de regiones y un conjunto de segmentos.
2. Implementar métricas que permitan evaluar qué tan convexo es un polígono.
3. Desarrollar una interfaz gráfica simple para visualizar los polígonos encontrados y sus propiedades geométricas (métricas).
4. Almacenar los polígonos y segmentos encontrados en formato pslg[1].

4. Solución Propuesta

Para realizar las pruebas de los algoritmos previas al desarrollo final se utilizará Processing, un lenguaje basado en el motor de Java, que esté enfocado a facilitar la interacción con el entorno gráfico para realizar las pruebas. Esto nos permitirá desarrollar un prototipo para evaluar posibles algoritmos. En primera instancia se comprobará experimentalmente el desempeño de los algoritmos. Esto se realizará generando pruebas de escala y experimentos que permitan confirmar lo esperado. Finalmente se traspasará la implementación a una ejecución en C++, para trabajar eficientemente con los bloques de memoria.

Los datos utilizados serán, como se ha especificado antes la imagen generada por *Painter's Algorithm*, en conjunto con una lista de segmentos en formato csv. Esos segmentos vienen como una lista de tuplas de puntos. Las listas son importantes para mantener los valores exactos de los segmentos, y para conservar la información de las fracturas en al aplicación,

[1] *Planar straight line graph*, es una forma de representar una estructura en dos dimensiones donde cada uno de los elementos del plano son segmentos de recta, puntos y polígonos. Permite puntos, segmento aislados y polígonos con agujeros.

estos últimos son usados por *Painter's Algorithm*, el cual es dependiente de la resolución de la imagen.

En un comienzo se discutió que la solución podría ser desarrollada en un formato web o como una aplicación de escritorio; por el momento nos centraremos en el desarrollo en formato escritorio. En el caso de que finalmente se genere el servicio web este se desarrollará en Python 3 con Flask, pero utilizando los llamados de c++ en el servicio Back.

El primer paso de solución es generar sistema base que permita realizar pruebas con distintos algoritmos, métricas y formatos de salida. Esto inicialmente se desarrollara de forma que pueda ser extensible y se ajuste a los cambios que posiblemente aparezcan durante el desarrollo de la memoria.

Una vez construido el esqueleto de la aplicación, se pasará a desarrollar las primeras soluciones propuestas, las cuales se basan en dos enfoques:

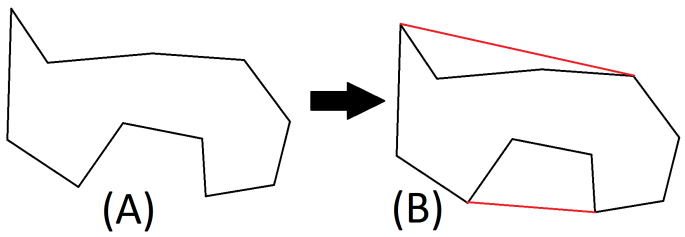
1. Generar una solución en base a la imagen y segmentos de entrada, tanto los colores de regiones y segmentos son usados para aproximar la ubicación y contracción de polígonos y fracturas creadas.
2. Re-implementar el algoritmo de *Painter's Algorithm*: Se planea extender este algoritmo en el cual usando los segmentos y la información de las regiones, se construyan los polinomios y fracturas. Esta alternativa puede romper con el paralelismo implementado en la solución original que utiliza cuda y c.

Las dos propuestas anteriores apuestan a encontrar las intersecciones de la imagen y solo calcular de manera exacta para los segmentos involucrados.

Sucesivamente se planea extraer los polígonos y los segmentos para cada una de estas regiones. Para esto no hay un algoritmo definido; una propuesta es un chequeo secuencial de los segmentos pertenecientes a la región y cálcular las intersecciones pertinentes, donde se espera no supere el orden $O(nm)$, donde n es el número de segmentos y m número de segmentos del polígono más grande en el peor de los casos.

Con los polígonos y fracturas ya calculados se requeriré guardar esta información en formato *planar straight line graph*, para lo cual se seguirá el formato descrito en [2], que nos exige generar tres archivos: .poly, .node y .edge.

Figura 3: Ejemplo cobertura convexa. (A) Polígono no convexo. (B) Cobertura convexa.



Para calcular las propiedades de los polígonos calculados basta implementar un algoritmo sobre los polígonos por cada métrica:

1. Razón entre el área del polígono y el área de su cerradura convexa.
2. Razón entre el perímetro del polígono y el perímetro de su cerradura convexa.

En la Figura 3. Podemos observar un ejemplo de cerradura convexa. La propiedad es importante dado que estructuralmente puede ser relevante si existe excesiva concavidad en polígono de un sector de un túnel o roca.

La aplicación esta abierta a futuro a incluir nuevas métricas propuestas en futuras instancias y otros formatos de salida.

Resumiendo los pasos a seguir en el desarrollo de la aplicación corresponde a:

1. Generar una solución simple pero fácilmente extensible que permita agregar: nuevas métricas, algoritmos y formatos de salida.
2. Generar una solución robusta y eficiente en C++ reusando y refinando el diseño anterior.
3. Explorar el algoritmo que pinta regiones, y establecer como aprovechar su eficacia en calcular las aristas pertenecientes a cada región.
4. Con las aristas, calcular intersecciones, polígonos y fracturas.
5. Guardar estructura previa en formato *planar straight line graph* (conjunto de .poly, .edge y .node).
6. Detectar y calcular propiedades geométricas de las estructuras generadas.

Esta es una memoria que se pretende utilizar en otros proyectos por lo cual requerirá del análisis de la funcionalidad de un software externo para atenerse a sus exigencias, aún así el foco de este proyecto esta bien delimitado.

Referencias

[1] Schneider, Sophie y Ivo F Sbalzarini: *Finding faces in a planar embedding of a graph.*

[2] Shewchuk, J. R.: *Triangle: A Two-Dimensional Quality Mesh Generator and Delaunay.* `https://www.cs.cmu.edu/%7Equake/triangle.html`.