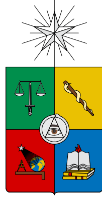

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE LAS CIENCIAS DE LA COMPUTACIÓN

FRACTURA DE POLÍGONOS COMPLEJOS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

GASTÓN IGNACIO JORQUERA AHUMADA

PROFESORA GUÍA:
MARÍA CECILIA RIVARA ZÚÑIGA

MIEMBROS DE LA COMISIÓN:
ERIC PIERRE TANTER
EDUARDO SALVADOR GODOY VEGA

SANTIAGO DE CHILE
ABRIL DE 2010

Resumen

Uno de los procesos del diseño de circuitos integrados digitales es la preparación de la información de las máscaras, o MDP por sus siglas en inglés (*Mask Data Preparation*). La preparación de máscaras recibe el diseño de un circuito y lo convierte en una secuencia de instrucciones que son leídas por una máquina generadora de máscaras. Este proceso es realizado por una serie de algoritmos, ordenados en forma de *pipeline*, donde la salida de uno es la entrada del siguiente.

Uno de los primeros algoritmos ejecutado es el llamado *Windfrac*, encargado de particionar, o fracturar, polígonos complejos en conjuntos de rectángulos y trapecios horizontales (cuyos lados paralelos son horizontales). Esta fractura inicial tiene gran importancia ya que, al reducir los distintos posibles polígonos de entrada a sólo rectángulos y trapecios horizontales, los algoritmos ejecutados después pueden ser simplificados e incluso tomar menos tiempo.

En esta memoria se estudia y documenta el funcionamiento del algoritmo *Windfrac*, y se reimplementa de una forma más legible y mantenible. El estudio del algoritmo, el cual fue el tema central y lo que más tiempo ocupó, contempla la revisión de ciertos conceptos geométricos y de geometría computacional necesarios para la total comprensión de éste.

Debido a que sólo existe un paper que explica un algoritmo parecido, toda la información acerca de cómo funciona *Windfrac* debió ser deducido a partir del código fuente mismo, cuya implementación era muy difícil de leer. El funcionamiento fue descifrado, principalmente, utilizando casos de prueba y depuradores para ir viendo, paso a paso, lo que el algoritmo hacía dado un polígono.

Una vez entendido el funcionamiento completo de *Windfrac* se reimplementó teniendo en cuenta legibilidad, mantenibilidad y ciertos detalles para mejorar la calidad de los resultados. La legibilidad y mantenibilidad se lograron con una implementación modular, es decir, utilizando estructuras de datos más especializadas y separando funcionalidades en archivos y funciones. La mejora de la calidad se logró escribiendo código para manejar esos casos particulares.

Finalmente, se realiza una discusión acerca del tema en estudio y sobre posibles mejoras que pueden ser llevadas a cabo en el futuro, las cuales podrían tener un gran impacto en el desempeño de la aplicación completa. Y, se concluye que el algoritmo fue satisfactoriamente comprendido y que su reimplementación soluciona los problemas de la implementación antigua.

Dedicado a . . .

Agradecimientos

"En primer lugar acabemos con Sócrates, porque ya estoy harto de este invento de que saber nada es un signo de sabiduría."

Isaac Asimov - La Relatividad del Error

"La ciencia se construye a partir de aproximaciones que gradualmente se acercan a la verdad."

Isaac Asimov - Anochecer

Asimov argumenta que nadie sabe nada, en sólo cuestión de días los bebés aprenden a reconocer a sus madres. Obviamente, Sócrates estaría de acuerdo con esto y explicaría que él se refería a que en las grandes abstracciones, sobre las cuales *discuten* los seres humanos, uno debe comenzar sin nociones preconcebidas y sin fundamentos. Básicamente, *partir* de nada, saber nada.

Asimov detestaba que esta famosa frase de Sócrates fuera utilizada para todo lo contrario, autocomplacerse en la ignorancia. Asimov era un hombre, lejos de ser ignorante, muy inteligente y sabio.

La mejor enseñanza que me deja Asimov es esta búsqueda de la verdad. Estas aproximaciones graduales que nos hacen estar cada vez más cerca. Estos errores que cada vez son menos *errados*. Asimov siempre estuvo dispuesto a aprender de cualquier persona y a transmitir su conocimiento. Y es así como quiero vivir mi vida.

Quiero expresar mis agradecimientos a la Universidad de Chile y a sus profesores, por siempre incentivar la búsqueda de la verdad. A la profesora María Cecilia Rivara, por la orientación entregada. A Synopsys, por permitirme realizar mi memoria en un problema real. A Domingo Morales, por siempre estar dispuesto a responder mis dudas, por muy simples que hayan sido. Y a todas las personas que me ayudaron con la redacción de este trabajo.

Índice General

Resumen I

Agradecimientos III

1. Introducción **1**

1.1. Fractura de Polígonos 4

1.1.1. Fractura Horizontal 4

1.1.2. Aproximación a Grilla 5

1.1.3. Resolución de la Grilla 5

1.2. Problema 6

1.2.1. Motivación 7

1.2.2. Objetivos Generales 7

1.2.3. Objetivos Específicos 7

1.3. Contenido de la Memoria 8

2. Conceptos Previos **9**

2.1. Conceptos Geométricos 9

2.1.1. Polígonos[1] 9

2.1.2. Trapecio y Trapecio Horizontal 11

2.2. Conceptos de Geometría Computacional 11

2.2.1. Interior y Exterior[8] 11

2.2.2. Algoritmos *Sweep-Line*[4] 17

2.2.3. Algoritmo de Detección de Intersección de Líneas[4][9][2] 18

2.2.4. Algoritmo *Healing*[7] 19

3. Procedimiento 22

3.1. Estudio y Análisis de *Windfrac* . . . 22

3.2. Mejora de *Windfrac* . . . 23

4. Algoritmo *Windfrac* 24

4.1. Definición[7] . . . 24

4.2. El Algoritmo[12][8] . . . 26

4.2.1. Algoritmo *Windfrac* a Grandes Rasgos . . . 27

4.2.2. Actualización de la Lista Activa . . . 30

4.2.3. Generación de las Primitivas . . . 31

4.2.4. Segunda actualización de la Lista Activa . . . 36

4.2.5. Intersecciones . . . 37

4.2.6. Aproximación a la Grilla . . . 39

4.3. Detalles de la Implementación Actual . . . 39

4.3.1. Definición de la Función *Windfrac* . . . 39

4.3.2. Almacenamiento de Aristas . . . 40

4.3.3. Implementación de la Lista Activa . . . 40

4.3.4. Implementación de los Eventos . . . 41

4.3.5. Aristas Superpuestas y Manejo de Intersecciones . . . 41

4.3.6. Precisión de las Coordenadas y Aproximación a Grilla . . . 41

4.3.7. Problemas de la Implementación Actual . . . 42

4.4. Detalles de la Reimplementación . . . 43

4.4.1. Módulo `algo` . . . 44

4.4.2. Módulo `algo-utils` . . . 44

4.4.3. Módulo `edge` . . . 45

4.4.4. Módulo `active-list` . . . 46

4.4.5. Módulo `winding` . . . 46

4.4.6. Módulo `event-queue` . . . 47

4.4.7. Módulo `event` . . . 47

4.4.8. Módulo `winding-list` . . . 48

4.4.9. Módulo `output` . . . 48

4.4.10. Módulo `grid` . . . 49

5. Resultados **50**

5.1. Legibilidad 50

5.2. Calidad de la Fractura 51

5.2.1. Coordenadas 51

5.2.2. Manejo de Intersecciones 51

5.2.3. Aproximación a Grilla 51

5.2.4. Aristas Superpuestas 52

5.3. Casos de Prueba 53

5.4. Desempeño 54

6. Discusión y Conclusiones **55**

6.1. Complimiento de Objetivos 55

6.2. Puntos a Mejorar 56

6.3. Trabajo Futuro 57

6.4. Conclusiones 57

Referencias **59**

Índice de Figuras

1.1. Pipeline de procesamiento en CATS. 2

1.2. Etapa de expansión del pipeline. 3

1.3. Fractura horizontal de un polígono complejo. 4

1.4. Aproximación a grilla de una intersección. 5

2.1. Distintos polígonos. 9

2.2. Clasificación de polígonos. 10

2.3. Distintos trapecios. 11

2.4. Una curva de Jordán que divide el espacio en dos regiones disjuntas. 12

2.5. Interior de polígonos con el método del área. 12

2.6. Interior de un polígono complejo utilizando la convención del área. 13

2.7. Interior de un polígono complejo utilizando la convención de la paridad. . . . 13

2.8. Cable que se auto-intersecta dibujado con un polígono. 13

2.9. Polígono con un agujero al medio. 14

2.10. Polígono con información en una de las regiones. 15

2.11. Número de vueltas del polígono en distintas partes del espacio. 15

2.12. Ejemplos de polígonos utilizando la convención del *winding-number*. 16

2.13. Definición alternativa del *winding-number*. 16

2.14. Cálculo del *winding-number* utilizando la nueva definición. 17

2.15. Líneas muy separadas trivialmente no se intersectan. 18

2.16. Primitivas no maximales y maximales en el eje x. 19

2.17. Ejemplo de resultados de *Healing*. 21

4.1. Una fractura de un polígono. 25

4.2. Ejemplos de primitivas superpuestas y primitivas no superpuestas. 25

4.3. Particiones “parecidas” al polígono original. 26
4.4. Particiones “no parecidas” al polígono original. 26
4.5. Particiones válidas de un polígono. 26
4.6. Dos aristas separadas en el eje y no pueden participar en la generación de la misma primitiva. 27
4.7. Vértices introducidos a la cola de eventos. 28
4.8. Las aristas azules pertenecen a la lista activa dada por la *sweep-line* roja. . . 29
4.9. La primera lista activa, definida por la primera *sweep-line*. 30
4.10. Actualización de la lista activa. 31
4.11. Intercambio de orden en la lista activa sólo cuando hay intersecciones. 31
4.12. Los distintos tipos de arista dependiendo de donde se encuentra el interior del polígono. 32
4.13. Generación de una primitiva. 32
4.14. Generación de primitivas por inserción. 34
4.15. Generación de primitivas por la izquierda. 35
4.16. Generación de primitivas por cruce. 35
4.17. Generación de primitivas por la derecha. 36
4.18. Inserción de nuevas aristas y eliminación de aristas procesadas. 37
4.19. Intercambio de orden en la lista activa en una intersección. 37
4.20. Orden de las aristas del polígono. 40
4.21. Caso en donde el resultado de *Windfrac* no es simétrico. 42
4.22. Caso en donde el resultado de *Windfrac* introduce cortes innecesarios. 43
4.23. Caso en donde el resultado de *Windfrac* genera un agujero. 43

5.1. Caso en donde el resultado de *Windfrac* es simétrico. 52
5.2. Caso en donde el resultado de *Windfrac* no introduce cortes innecesarios. . . 52
5.3. Caso en donde el resultado de *Windfrac* no genera un agujero. 52
5.4. Mediciones de tiempo de las distintas implementaciones de *Windfrac*. 54

Índice de Algoritmos

2.1. Algoritmo de detección de intersecciones en segmentos de línea rectas. 20
4.1. Algoritmo *Windfrac* a grandes rasgos. 29
4.2. Actualización de la lista activa del algoritmo *Windfrac*. 30
4.3. Generación de primitivas del algoritmo *Windfrac*. 33
4.4. Generación de primitivas por inserción. 34
4.5. Generación de primitivas por la izquierda. 35
4.6. Generación de primitivas por cruce. 36
4.7. Generación de primitivas por la derecha. 36
4.8. Segunda actualización de la lista activa del algoritmo *Windfrac*. 37
4.9. Actualización de la lista activa del algoritmo *Windfrac* tomando en cuenta posibles intersecciones entre aristas. 38
4.10. Segunda actualización de la lista activa del algoritmo *Windfrac* tomando en cuenta posibles intersecciones entre aristas. 38
4.11. Detección de intersecciones y actualización de la cola de eventos del algoritmo *Windfrac*. 39

Capítulo 1

Introducción

Synopsys es una empresa líder en el desarrollo de aplicaciones para el diseño automatizado de circuitos integrados digitales complejos (en Inglés, *EDA: Electronic Design Automation*). Provee al mercado global de electrónica con aplicaciones, propiedad intelectual y servicios usados en el diseño y manufactura de semiconductores[11].

Uno de los productos de Synopsys es CATS® (Sistema de Transcripción Asistido por Computador, en Inglés, *CATS: Computer Aided Transcription System*), una aplicación altamente escalable y flexible encargada de transcribir datos de diseño de circuitos complejos en instrucciones para máquinas generadoras de máscaras[10]. Esta transcripción es llamada preparación de información de máscaras (en Inglés, *MDP: Mask Data Preparation*).

En otras palabras, MDP es el proceso de traducir un conjunto de polígonos de un esquema de un circuito integrado en una serie de instrucciones para que pueda ser físicamente escrita por una máquina dibujadora de máscaras. Este proceso es realizado en la generación de patrones y manufacturación de circuitos integrados, sistemas microelectrónicos, entre otros[6].

La implementación de MDP en CATS es mediante un proceso que incluye varias etapas, cada una compuesta por diversos algoritmos, organizados en forma de pipeline (la salida de una etapa es la entrada de la siguiente)[7].

Estas etapas son las siguientes:

Lectura Corresponde a la lectura de un diseño de un circuito, soportando diversos formatos.

Expansión Aplana diseños definidos con jerarquías y particiona polígonos complejos en figuras más simples.

Funciones Unarias Funciones que se aplican a un diseño, por ejemplo, sizing para modificar el tamaño de las figuras.

Funciones Binarias Funciones que se aplican a dos diseños, por ejemplo, OR para indicar lugares donde hay figuras en alguno de los dos diseños.

Fractura Repoligonización y partición en un conjunto óptimo de polígonos simples.

Escritura Escritura del diseño procesado a un archivo en diversos formatos de máquina e internos.

La Figura 1.1 muestra la comunicación entre las etapas mencionadas anteriormente. Como se puede ver, si la operación escogida es binaria, entonces en ese caso es necesario cargar un segundo diseño.

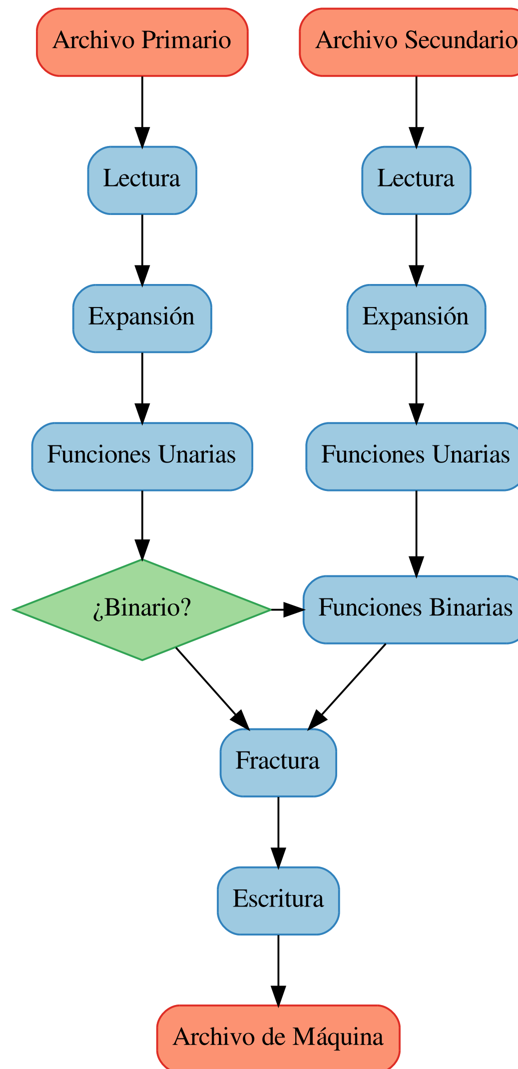

Figura 1.1: Pipeline de procesamiento en CATS.

Cabe destacar que la tendencia actual es ir mezclando estas etapas en pasos cada vez más cohesionados y más eficientes. La razón de este cambio de arquitectura de pipeline a una arquitectura más bien monolítica es debido a la gran cantidad de información[1] que se maneja

[1] Existen diseños de circuitos con más de 400 GB de información.

dentro de la aplicación, por lo que es necesario optimizar el traspaso de esta información entre las distintas etapas[7].

La etapa de *expansión*, además de aplanar un diseño jerárquico, como ya se dijo anteriormente, debe convertir polígonos complejos en un conjunto de figuras más simples que no se superpongan. Este proceso es realizado por el algoritmo denominado *Windfrac*, el cual es llamado polígono a polígono, junto con el algoritmo denominado *Healing*, el cual es llamado sobre todas las figuras generadas por *Windfrac*[7].

Por lo tanto, *Windfrac* es el encargado de, dado un polígono complejo (un polígono cóncavo o convexo, con o sin autointersecciones), generar un conjunto de figuras geométricas simples (triángulos, rectángulos o trapecios cuyos lados paralelos son horizontales) que no se superpongan entre ellas.

Por otro lado, *Healing* es el encargado de, dado un conjunto de figuras simples que representan todos los polígonos originales del diseño, generar otro conjunto de figuras simples que no se superpongan. Es decir, *Healing* tiene como función remover superposiciones entre figuras simples generadas por distintos polígonos a la salida de *Windfrac*.

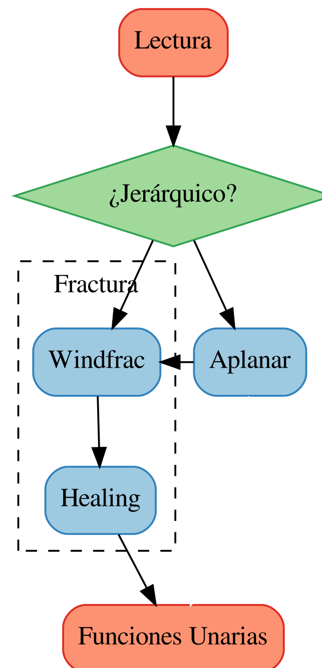

Figura 1.2: Etapa de expansión del pipeline.

La Figura 1.2 representa la relación entre las distintas fases que componen la etapa de expansión.

1.1. Fractura de Polígonos

El propósito de particionar los polígonos complejos es para que los algoritmos que son ejecutados después de *Windfrac* trabajen sobre figuras simples con propiedades bien definidas. Si se hubiera dejado la entrada con polígonos cualquiera, entonces los algoritmos ejecutados sobre esta entrada serían mucho más complejos y probablemente terminen tomando más tiempo que fracturar y ejecutar sobre estas figuras simples.

Cabe destacar que hay algoritmos que funcionan sobre polígonos (repoligonizan la entrada antes de ser ejecutados) para obtener resultados de mayor calidad, sin embargo estos polígonos igualmente deben tener ciertas propiedades, por ejemplo, no pueden tener autointersecciones ni intersecciones entre polígonos.

1.1.1. Fractura Horizontal

Fractura horizontal es el proceso de convertir el interior de un polígono en un conjunto de primitivas que tiene aproximadamente la misma forma que el polígono original, pero sin que las primitivas queden superpuestas.

Una primitiva es una figura simple, capaz de ser impresa por una máquina dibujadora de máscaras. Una primitiva puede ser un rectángulo, un trapecio horizontal (un trapecio con sus lados paralelos horizontales) o un triángulo (que es un trapecio horizontal degenerado, donde uno de sus lados tiene tamaño nulo).

Dos primitivas se consideran superpuestas si el área que encierran están superpuestas, permitiendo tener aristas y vértices en común. Es decir, se definen las áreas de las primitivas como abiertas, por lo que el contorno unidimensional del polígono no es considerado como interior del polígono.

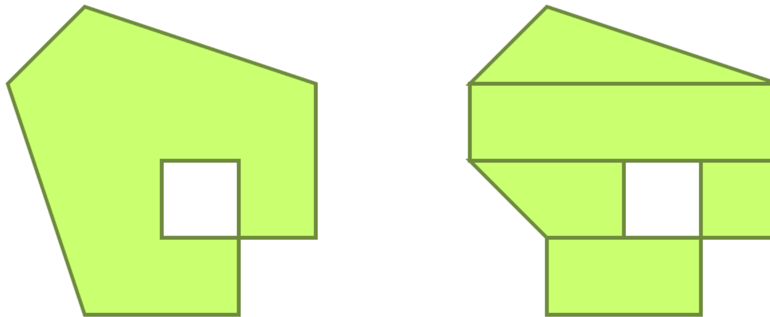

Figura 1.3: Fractura horizontal de un polígono complejo.

La Figura 1.3 muestra un ejemplo de una fractura horizontal. El área verde claro representa el interior del polígono mientras que las líneas verde oscuro son el contorno del polígono y de las primitivas. La figura de la izquierda es el polígono original y el conjunto de primitivas de la derecha (los tres rectángulos, el trapecio y el triángulo) son el resultado de la fractura. Como se puede apreciar, ninguna de las cinco primitivas queda sobre otra, sólo se tocan en aristas y vértices en común.

1.1.2. Aproximación a Grilla

Un punto importante que hay que tomar en cuenta, constituyendo una restricción del problema, es que todos los vértices de todos los polígonos y primitivas deben quedar posicionados sobre una grilla.

Es crucial tener esto en cuenta porque las intersecciones pueden no calzar en la grilla, por lo que habría que aproximarlas. Esta aproximación debe ser realizada con cuidado ya que una aproximación incorrecta se puede ir degradando a lo largo del proceso hasta formar un circuito defectuoso.

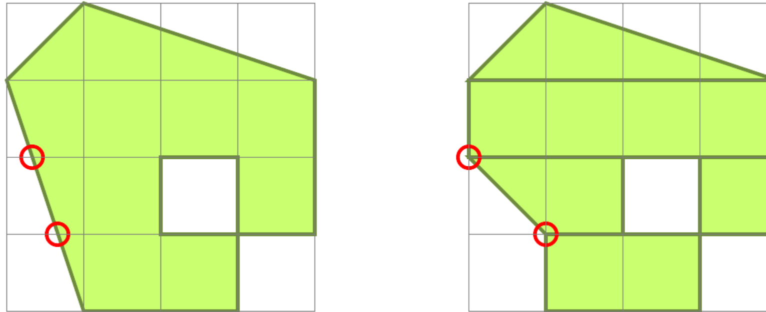

Figura 1.4: Aproximación a grilla de una intersección.

La Figura 1.4 muestra cómo las intersecciones marcada con rojo del polígono original debieron ser movidas para que las primitivas resultantes de la fractura tengan todos sus vértices en puntos de la grilla.

1.1.3. Resolución de la Grilla

Existen dos tipos de máquinas que imprimen máscaras: raster y VSB (en Inglés, *VSB: Variable Shape Beam*). Ambas máquinas funcionan utilizando una grilla como referencia. Las máquinas raster van bombardeando, unidad a unidad de la grilla, donde hay información (parecido a una impresora de matriz de puntos), mientras que las máquinas VSB tienen un conjunto de figuras base y bombardean estas figuras instantáneamente, alineadas a la grilla.

Debido a que ambos tipos de generación de máscaras necesitan de una grilla de referencia y que estas grillas pueden ser de distintos tamaños, se hace necesario poder manejar distintas resoluciones. Por lo tanto, dada una resolución, se supone que los vértices de los polígonos de un diseño (ya sean primitivas o polígonos complejos) siempre están ubicados en la grilla.

Si bien las máquinas pueden imprimir máscaras a distintas resoluciones, una vez escogida, el diseño completo debe ser impreso a esa misma resolución.

En CATS se manejan dos resoluciones distintas: la resolución del diseño, almacenada en el archivo de entrada, y la resolución de fractura, almacenada en el archivo de salida. Esto se debe a que un diseño puede estar especificado a cierta resolución, pero, querer realizar la fractura a una mayor (para generar mejores resultados) o menor resolución (debido a que la máquina no soporta la resolución del diseño).

Todo esto es transparente para el algoritmo *Windfrac*, puesto que la información sobre la resolución le llega de manera implícita. *Windfrac* trabaja sobre la grilla de los números enteros, esto quiere decir que “llevar a grilla” significa mover un punto a un entero. La interpretación de la relación entre los enteros y la resolución de entrada o de la fractura depende de variables fuera de *Windfrac*.

Por ejemplo, un punto en un diseño puede estar ubicado a 3 micrones del origen. Si la resolución de este archivo es 1 (hay un punto de grilla por cada micrón), entonces, a *Windfrac* le llegará el punto como el entero 3. Pero, si el archivo de entrada está a resolución 0.001 (hay 1000 puntos de grilla por cada micrón), entonces, ese mismo punto le llegará como el entero 3000.

Cabe destacar que es posible que a *Windfrac* le llegue un punto de forma decimal. Esto sucede cuando la resolución de fractura es menor que la resolución de entrada y se fuerza a que el algoritmo fracture con la resolución de fractura. En este caso, al llevar a la resolución de la fractura, ciertos puntos pueden quedar con decimales.

Por ejemplo, si la resolución de entrada es de 0.1 micrones (hay 10 puntos de grilla por micrón), entonces, un punto a 0.2 micrones estaría representado por el entero 2 en la resolución del archivo de entrada. Pero, si la resolución de fractura es de 1 micrón y se fuerza a que *Windfrac* reciba los datos con la resolución de fractura, entonces, ese punto no llegará aproximado a 0, sino que llegará como 0.2.

1.2. Problema

El algoritmo *Windfrac* es de suma importancia ya que es utilizado cada vez que se desea trabajar con el diseño de un circuito. Si bien funciona de forma estable, actualmente tiene unos problemas.

Windfrac fue implementado hace más de 20 años por un desarrollador que ya no trabaja para Synopsys, por lo que es muy difícil realizar cambios debido a que nadie sabe bien cómo funciona el algoritmo mismo ni cómo está implementado.

Se desea mejorar la calidad de los resultados de la fractura realizada por *Windfrac*. Existen ciertos casos en donde el resultado del algoritmo no es el deseable, generando potenciales problemas en el diseño.

Además, es deseable que el algoritmo sea uniforme, ésto quiere decir que, si recibe como entrada el mismo polígono que uno ya ha procesado, entonces, debe devolver el mismo resultado, ojalá, sin volver a calcularlo.

A esto se suma un problema transversal crítico: el algoritmo está implementado de forma deficiente, es decir, sin documentación, con variables que tienen nombres sin sentido, sin uso apropiado de funciones ni estructuras de control, entre otros. *Windfrac* está implementado en una sola función de aproximadamente tres mil líneas de código.

De ahí surge la necesidad de documentar el funcionamiento del algoritmo *Windfrac* y de mejorar su implementación.

1.2.1. Motivación

El tema es interesante por varias razones. La primera, y probablemente la más importante, es que está asociado a una área que tiene mucho futuro ya que siempre será necesaria la investigación para ir mejorando y optimizando estos algoritmos geométricos, de tal forma de poder procesar la creciente cantidad de información que se maneja.

También es interesante porque está inserto en un área relativamente joven. Se creó el área de la Computación Geométrica a finales de los años 70[4], y es un área muy poco desarrollada en Chile.

Además, el tema es suficientemente complejo ya que involucra el estudio y documentación de la implementación de un algoritmo que lleva en funcionamiento por más de dos décadas, junto con el estudio y análisis de distintas formas de poder mejorarlo.

Cabe destacar que es un problema y necesidad del mundo real.

1.2.2. Objetivos Generales

Estudiar, analizar, investigar y mejorar la actual implementación de *Windfrac*. Esta mejora puede ser mediante correcciones al algoritmo actual, la reimplementación del mismo algoritmo, pero, tomando en cuenta nuevos supuestos, o bien la implementación de un nuevo algoritmo para resolver el problema de la fractura horizontal.

Además, se pide que el nuevo algoritmo sea competitivo con el actual, uniforme y consistente.

1.2.3. Objetivos Específicos

Los objetivos específicos son los siguientes:

- Estudiar, analizar y documentar:
  - La entrada del algoritmo.
  - El algoritmo implementado actualmente.
  - Posibles soluciones alternativas al problema de fractura.
- Mejorar (corregir, reimplementar o diseñar e implementar) el algoritmo para solucionar el problema de fractura horizontal.
- Hay que tomar en cuenta las siguientes restricciones:
  - Eficiencia competitiva con respecto al algoritmo actual, es decir, debe ser, por lo menos, del mismo orden que el actual y, ojalá, demorarse menos al ser ejecutado.
  - Resolver los problemas de calidad de los resultados.
  - Uniformidad de los resultados del proceso, es decir, polígonos iguales deben producir resultados iguales, ojalá, sin la necesidad de volver a realizar el proceso.

  - Consistencia en la toma de decisiones, es decir, al momento de tomar decisiones sobre las aproximaciones del polígono, que sean realizadas tomando en cuenta el contexto, no sólo la información local.
  - Encapsular la toma de decisiones de aproximación a grilla de tal forma que sea fácilmente intercambiable.
  - Ser independiente de la precisión de la grilla en la que se trabaja.
  - Ser independiente de la representación interna de los datos, es decir, debe funcionar para coordenadas de 32bit y 64bit.

- Se debe especificar la entrada del algoritmo si es que es necesario modificarla con respecto a la actual. Esta modificación debe ser factible puesto que la entrada de este algoritmo es la salida de otro.

- El nuevo algoritmo debe pasar los casos de prueba existentes y solucionar los problemas actuales.

1.3. Contenido de la Memoria

Esta memoria está compuesta de los siguientes Capítulos:

Capítulo 2, Conceptos Previos Se definen los distintos conceptos geométricos y de computación geométrica utilizados en la memoria, en especial, los algoritmos de tipo *sweep-line*.

Capítulo 3, Procedimiento Explica los pasos seguidos en el desarrollo de esta memoria.

Capítulo 4, Algoritmo *Windfrac* Se define el problema que resuelve *Windfrac* junto con una explicación detallada de su funcionamiento, su implementación actual y cómo fue reimplementado.

Capítulo 5, Resultados Se muestran los resultados obtenidos de la nueva implementación de *Windfrac* en comparación con la implementación actual.

Capítulo 6, Discusión y Conclusiones Se discuten las decisiones tomadas al realizar el trabajo, además del futuro de *Windfrac*. También se enumeran las conclusiones de esta memoria.

Capítulo 2

Conceptos Previos

En este Capítulo se revisan diferentes conceptos que son necesarios para poder entender el funcionamiento del algoritmo *Windfrac*. Contiene las siguientes secciones:

Sección 2.1, Conceptos Geométricos Se definen ciertas figuras geométricas usadas por los algoritmos geométricos.

Sección 2.2, Conceptos de Geometría Computacional Se definen propiedades de los polígonos, y se explica uno de los tipos de algoritmos de computación geométrica: los algoritmos *sweep-line*.

2.1. Conceptos Geométricos

2.1.1. Polígonos[1]

Definición 1 *Un polígono (Figura 2.1) es una figura plana generada por un conjunto de segmentos de líneas rectas formando una curva cerrada.*

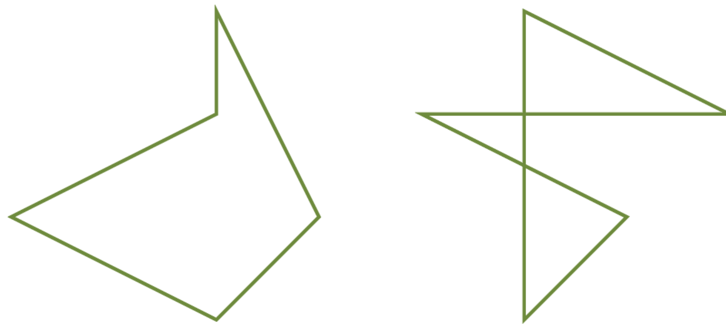

Figura 2.1: Distintos polígonos.

Estos segmentos de líneas forman una curva porque están conectados uno tras otro. Es una curva cerrada porque forma un ciclo al estar conectada la última línea con la primera.

Es posible clasificar los polígonos dependiendo de la forma de su contorno. Por lo tanto, un polígono se denomina:

Simple Si el límite del polígono no se auto-intersecta, es decir, si dos de sus aristas no consecutivas no se intersectan.

Complejo Si es no simple.

Convexo Si cualquier línea recta que atraviesa el polígono (que no sea tangente a un vértice o arista) corta al polígono en exactamente 2 puntos.

Cóncavo Si no es convexo.

Regular Si tiene todos sus ángulos y lados iguales.

Irregular Si no es regular.

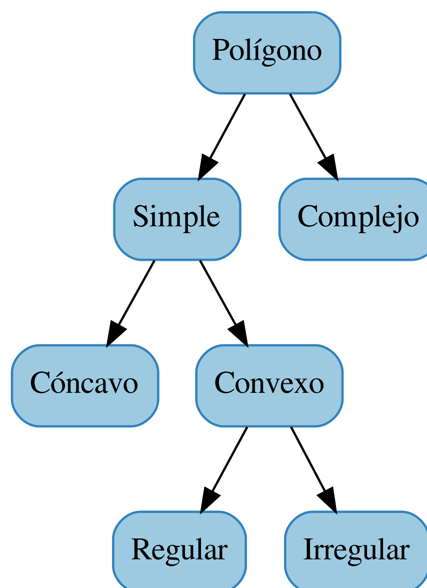

Figura 2.2: Clasificación de polígonos.

La Figura 2.2 muestra las distintas categorías y subcategorías de polígonos dependiendo de su forma.

Los segmentos de líneas se llaman aristas o lados, mientras que los puntos comunes entre cada arista se llaman vértices. Al conjunto de segmentos de líneas se le suele llamar contorno o límite del polígono.

2.1.2. Trapecio y Trapecio Horizontal

Definición 2 *Un trapecio es un cuadrilátero (un polígono de cuatro lados) que tiene dos lados paralelos y dos lados no paralelos[5].*

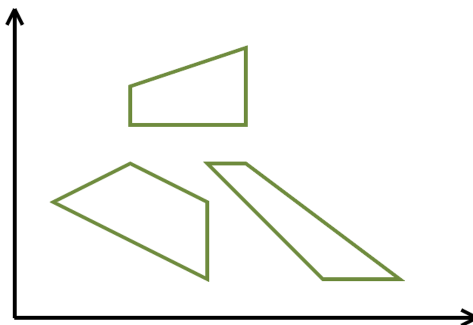

Figura 2.3: Distintos trapecios.

Definición 3 *Un trapecio horizontal es un trapecio en donde sus bases (los lados paralelos entre sí) son paralelos al eje x[7].*

Definición 4 *De igual forma, se dice que un trapecio vertical es un trapecio en donde sus lados paralelos son paralelos con el eje y[7].*

La Figura 2.3 muestra distintos trapecios, de los cuales el de abajo a la derecha es un trapecio horizontal y el de arriba es un trapecio vertical.

2.2. Conceptos de Geometría Computacional

2.2.1. Interior y Exterior[8]

Debido a que un polígono se construye a partir de un conjunto de segmentos de líneas rectas que definen una curva cerrada, tiene sentido poder determinar si un punto se encuentra *dentro* o *fuera* del polígono.

El interior de un polígono es de suma importancia para el diseño de circuitos digitales ya que así se define donde habrá material y donde no. En otras palabras, al generar las máscaras, el interior de los polígonos representan los lugares donde habrá material (también llamado *data* o *información*).

Para polígonos simples es trivial, debido al teorema de la curva de Jordán[3]:

Teorema 1 *Toda curva cerrada simple plana (también llamada curva de Jordán) divide el plano en dos componentes conexas disjuntas que tienen la curva como frontera común.*

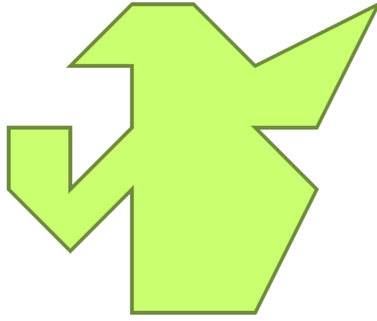

Figura 2.4: Una curva de Jordán que divide el espacio en dos regiones disjuntas.

Por lo tanto, si se le llama *interior* a la región del espacio limitada por la curva cerrada y *exterior* al resto del espacio, entonces un punto se considera *dentro* de un polígono si es que se encuentra en el *interior* de éste.

El problema ocurre cuando el polígono es complejo, ya que en estos casos el polígono separa el espacio en más de dos regiones, por lo tanto, no hay una forma tan intuitiva de definir su interior y exterior. Debido a esto, existen varias convenciones para determinar el interior y exterior de un polígono cualquiera dado.

A continuación se verán estas distintas convenciones.

Convención del Área Encerrada

Un punto se considera dentro del polígono si se encuentra encerrado por los límites de éste. Esta convención no involucra la interacción con otros polígonos, el interior y exterior se define polígono a polígono.

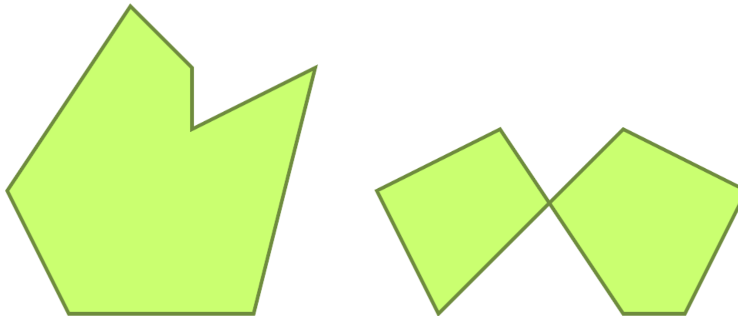

Figura 2.5: Interior de polígonos con el método del área.

Si bien esta convención se comporta bien con polígonos simples e incluso con algunos polígonos complejos, como en la Figura 2.5, no permite definir agujeros dentro de un polígono.

La Figura 2.6 muestra cómo el área interna siempre será considerada como interior al estar rodeada, de alguna forma, por el polígono mismo.

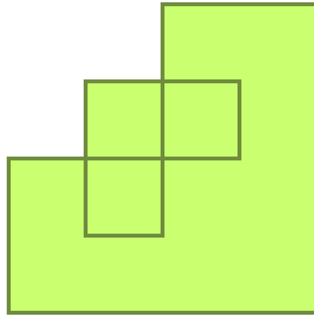

Figura 2.6: Interior de un polígono complejo utilizando la convención del área.

Convención de la Paridad

El método de la paridad es una convención comúnmente utilizada, la cual dice que un punto está dentro de un polígono dependiendo de la paridad del número de intersecciones entre las aristas del polígono y una línea recta dibujada desde el punto hacia el infinito en cualquier dirección. Si el número de intersecciones es impar, entonces se considera interior, si es par, entonces se considera exterior. Esta convención, al igual que la anterior, se define sin interacción entre distintos polígonos.

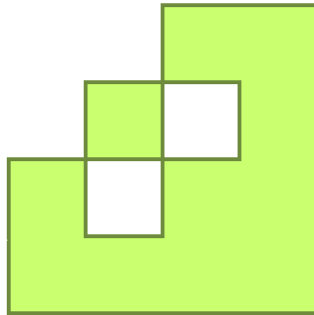

Figura 2.7: Interior de un polígono complejo utilizando la convención de la paridad.

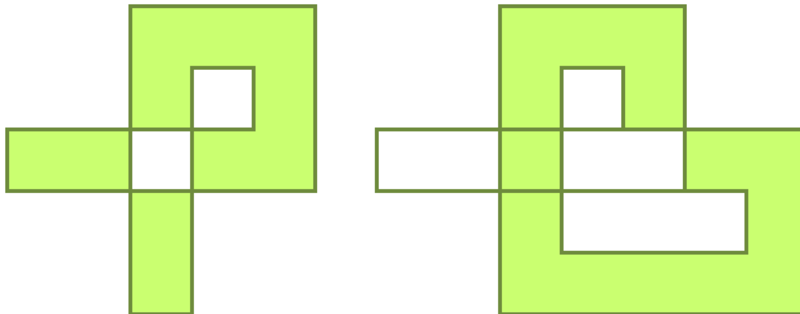

Figura 2.8: Cable que se auto-intersecta dibujado con un polígono.

Esta convención funciona para polígonos simples de la misma forma que la convención anterior, y, además permite definir agujeros en polígonos complejos. La Figura 2.7 mues-

tra polígonos complejos con agujeros, los cuales tienen un número par de intersecciones en cualquier dirección.

El principal problema de esta convención es que, si bien provee una definición consistente del interior y exterior de un polígono, en el mundo de circuitos integrados no es intuitiva. Por ejemplo, si se utiliza un polígono para representar un cable, como en la Figura 2.8, al autointersectarse producirá una región donde no habrá material. Es más, si se volviera a intersectar en la misma región, habrá material nuevamente.

Convención de la Orientación

Debido a que un polígono es una curva cerrada, es posible recorrer las aristas de éste una tras otra, siguiendo siempre el mismo sentido. Luego, dada una región limitada por el polígono, se puede decir que está definida a favor o en contra de las manecillas del reloj dependiendo de si al recorrer las aristas del polígono, esa región se recorre a favor o en contra de las manecillas del reloj.

La convención de la orientación toma en cuenta la orientación de las aristas que definen el polígono. Entonces, por ejemplo, límites definidos en contra de las manecillas del reloj implica que esa región es interior mientras que límites definidos a favor, son agujeros. Un polígono que define un agujero elimina exactamente un polígono que define un interior.

Esta convención, al definir explícitamente que cierto *tipo* de polígonos son agujeros, permite que exista interacción entre polígonos distintos. Por lo tanto, permite definir polígonos con agujeros al medio, como en la Figura 2.9

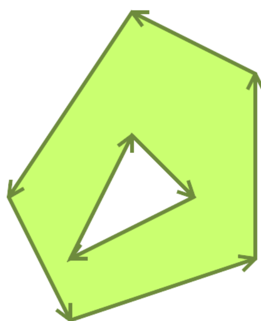

Figura 2.9: Polígono con un agujero al medio.

Un problema de esta convención es que siempre hay que tener en cuenta la orientación, en especial, al realizar operaciones como reflejar con respecto a algún eje, en donde se invertiría la orientación de todos los polígonos. Otro problema que introduce es su asimetría, por ejemplo, en la Figura 2.10 sólo una de las regiones del polígono es considerado como información puesto que una de las regiones está definida en contra de las manecillas del reloj mientras que la otra region está a favor (generando un agujero).

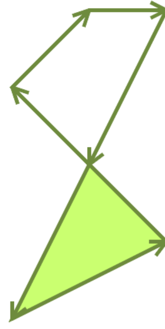

Figura 2.10: Polígono con información en una de las regiones.

Convención del ***Winding-Number***

Definición 5 *El winding-number o número de vueltas es un entero que representa el número total de veces que una curva cerrada recorre alrededor de un punto dado a favor de las manecillas del reloj[13].*

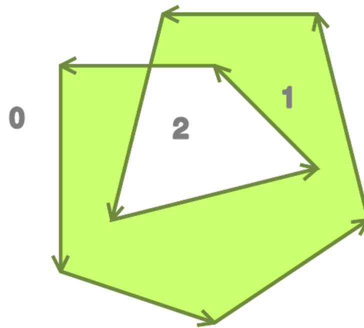

Figura 2.11: Número de vueltas del polígono en distintas partes del espacio.

La Figura 2.11 muestra que el número de vueltas alrededor de un punto cualquiera fuera del polígono es 0, y dentro del polígono hay una región en donde en cualquier punto el contorno del polígono da una vuelta completa y otra donde da dos vueltas. Si el polígono hubiera estado definido en orientación contraria, el número de vueltas sería negativo.

Cabe notar que utilizando esta definición, decir que un punto se encuentra dentro de un polígono si su *winding-number* es estrictamente positivo es equivalente a la convención de la orientación. La asimetría de esa convención aparece debido a la asimetría sobre la condición sobre el *winding-number*.

La convención del número de vueltas elimina esta asimetría estableciendo que un punto está dentro de un polígono si el *winding-number* es distinto de cero. De esta forma se obtiene que contornos definidos *en contra* de otros contornos eliminan capas de material, pero es independiente de la orientación.

La Figura 2.12 muestra varios ejemplos de polígonos complejos con lo que se considera interior utilizando la convención del *winding-number*.

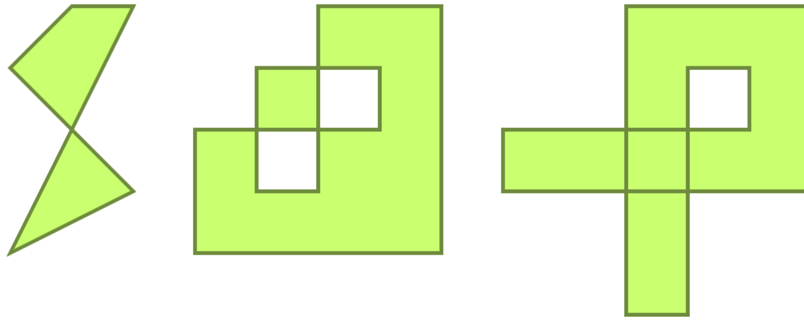

Figura 2.12: Ejemplos de polígonos utilizando la convención del *winding-number*.

La definición formal del *winding-number* no es práctica porque su implementación es muy engorrosa. Afortunadamente, hay una definición alternativa, que es la utilizada en la realidad.

Definición 6 *El winding-number es el número de aristas que, al intersectar una línea imaginaria desde el punto hacia el infinito a la izquierda, van bajando menos el número de aristas que, al intersectar esta misma línea imaginaria, van subiendo.*

Cabe destacar que, en realidad, la lína imaginaria se puede dibujar en cualquier dirección pero, nuevamente por simplicidad, se escoge hacia la izquierda.

Esta nueva definición es mucho más útil puesto que se puede ir desde la izquierda hacia la derecha sumando las aristas que bajan y restando aquellas que suben y, en cada momento, tendremos el *winding-number* para esa región. El *winding-number* cambiaría solo al encontrarse con una nueva arista.

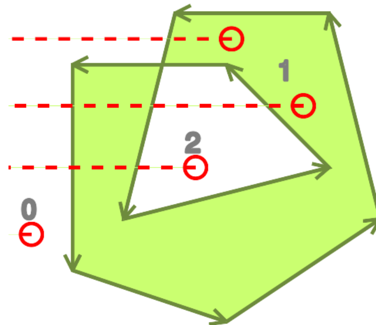

Figura 2.13: Definición alternativa del *winding-number*.

Como se puede ver en la Figura 2.13 cualquier punto a la izquierda del polígono tiene *winding-number* 0 porque no intersecta aristas. Luego, dentro del polígono, hay regiones cuyo *winding-number* es 1 porque intersecta sólo una arista que va hacia abajo, o bien intersecta 3 de las cuales dos van hacia abajo y una va hacia arriba. Además, hay regiones cuyo *winding-number* es 2 puesto que intersecta sólo dos aristas que van hacia abajo.

La Figura 2.14 muestra un ejemplo de cómo el *winding-number* se va calculando de izquierda a derecha a lo largo de la línea punteada. Como se puede apreciar, cualquier punto

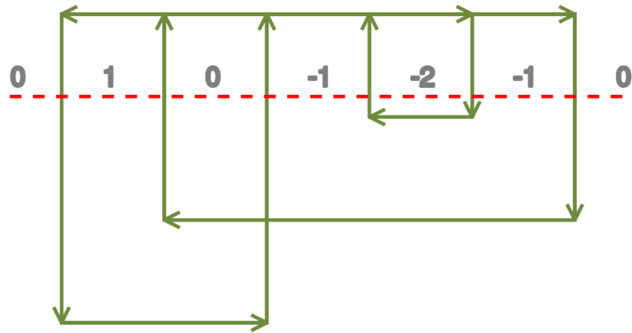

Figura 2.14: Cálculo del *winding-number* utilizando la nueva definición.

a la izquierda del polígono tendrá *winding-number* 0, debido a que ninguna arista intersecta la línea verde. Luego, al moverse hacia la derecha y cruzar la primera arista, el *winding-number* de cualquier punto de esa región será 1 puesto que intersecta sólo una arista que va hacia abajo. Y así se sigue sumando las que bajan y restando las que suben en cada región hasta que se llega al final en donde el *winding-number* debe valer 0 nuevamente, puesto que el polígono se define mediante una curva cerrada.

2.2.2. Algoritmos ***Sweep-Line***[4]

Definición 7 *Un algoritmo sweep-line es un tipo de algoritmo que utiliza una línea imaginaria para resolver problemas geométricos barriendo el espacio.*

Definición 8 *Un evento es un lugar en el espacio en donde la sweep-line se detiene para procesar los datos.*

Utilizando esta línea conceptual, se puede suponer que toda la información que queda detrás de la *sweep-line* ya está procesada, las decisiones se toman con la información que está sobre la *sweep-line* y lo que está adelante aún no se ha procesado.

Definición 9 *La lista activa es la lista de elementos que están sobre una sweep-line dada.*

Por lo tanto, la idea detrás de algoritmos de este tipo es imaginar una línea que se mueve o barre el plano, deteniéndose en algunos puntos (eventos). Las operaciones geométricas están restringidas a los objetos que están en la vecindad inmediata o intersectan la línea cuando ésta se detiene (lista activa), y la solución completa se obtiene una vez se haya pasado sobre todos los objetos.

Los eventos, así como la estructura de datos para almacenarlos y la estructura de datos para almacenar la lista activa dependen plenamente del problema que se desea resolver.

2.2.3. Algoritmo de Detección de Intersección de Líneas[4][9][2]

Un ejemplo de un algoritmo *sweep-line* es el algoritmo de detección de intersecciones entre segmentos de líneas rectas. Este algoritmo es útil por dos razones, sirve para ejemplificar cómo un algoritmo *sweep-line* mejora el tiempo de ejecución comparado con la versión de fuerza bruta y porque el algoritmo *Windfrac* es una modificación de este algoritmo.

El problema que resuelve este algoritmo es el siguiente:

Problema 1 *Dado un conjunto de segmentos de líneas rectas, se desea reportar todas las intersecciones entre elementos del conjunto.*

La aproximación ingenua a la solución sería realizar la detección de todas las líneas contra todas las otras líneas del conjunto, realizando $O(n^2)$ detecciones. En cierto sentido esto es óptimo puesto que si cada par de líneas se intersectan, entonces, cualquier algoritmo tomará tiempo $\Omega(n^2)$. Lo que sucede es que, en la práctica, sólo un subconjunto de todos los pares de línea se intersectan.

Debido a la geometría del problema, se pueden hacer ciertas "simplificaciones". Lo primero que uno puede notar es que no es necesario detectar intersecciones entre todas las líneas, basta con detectar intersecciones en aquellas que estén más o menos a la misma altura en el eje y. Si dos líneas están muy separadas, entonces, no es posible que se intersecten.

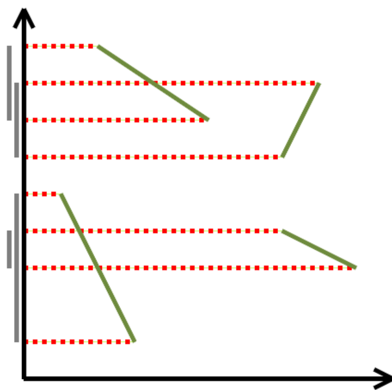

Figura 2.15: Líneas muy separadas trivialmente no se intersectan.

La Figura 2.15 muestra con claridad que sólo las líneas cuya proyección en el eje y se superponen tienen posibilidad de que se intersecten. También, se observa que, tanto las dos líneas de abajo como las dos líneas de arriba tienen posibilidad de intersectarse. Por lo tanto, uno podría utilizar una *sweep-line* horizontal para así detectar intersecciones sólo en aquellas líneas que toquen la *sweep-line*, ya que éstas tendrán su proyección en el eje y superpuestas.

El problema de este acercamiento es que no es suficiente. Si, por ejemplo, todas las líneas intersectan al eje x, entonces, nuevamente habría un momento en donde se tendría que detectar intersecciones entre todos los pares de líneas del conjunto.

Para resolver este problema hay que darse cuenta que no es necesario detectar intersecciones con todas aquellas líneas cuya proyección en el eje y se superponen. Basta con detectar

intersecciones entre las líneas que están a ambos lados. Si en algún momento del algoritmo aparece una nueva línea, entonces, se detecta intersecciones entre sus nuevos vecinos. Además, cuando dos líneas se intersectan, éstas intercambian su posición (la que estaba a la izquierda pasa a estar a la derecha y viceversa), por lo tanto, hay que detectar intersecciones con los nuevos vecinos. También, si una línea desaparece, entonces, los vecinos de esta línea ahora son vecinos entre ellos, así es que hay detectar nuevamente intersecciones.

Por lo tanto, el algoritmo de detección de intersecciones, a grandes rasgos, utiliza una *sweep-line* para barrer el espacio de abajo hacia arriba. En cada *sweep-line* el algoritmo mantiene una lista de líneas activas, que son las que están tocando la *sweep-line*. Los eventos del algoritmo son los vértices de las líneas y los puntos de intersección. En cada evento, el algoritmo actualiza la lista de líneas activas y detecta intersecciones con las líneas de ambos lados. Y, en el momento en que el algoritmo detecta que hay una intersección entre dos líneas, actualiza la cola de eventos y la reporta.

Una descripción más detallada sobre cómo se manejan los eventos (cuando aparece una nueva línea, cuando hay una intersección y cuando desaparece una línea), la cola de eventos y la lista activa se puede ver en el Algoritmo 2.1.

Las líneas 16, 20 y 23 del Algoritmo 2.1 se refieren a que los nuevos eventos sólo pueden ser las intersecciones entre los segmentos involucrados. Pero, sólo aquellas intersecciones arriba o a la derecha del evento actual son nuevos eventos válidos porque aquellos a la izquierda o abajo ya fueron procesados.

Se puede demostrar que este algoritmo, basado en una *sweep-line*, toma tiempo $O(n\log(n)+k\log(n))$, donde n es el número de segmentos de línea y k es el número de intersecciones. Este orden de magnitud nos dice que, si k es pequeño, como en la mayoría de los casos, el algoritmo se ahorrará mucho trabajo.

2.2.4. Algoritmo ***Healing***[7]

El algoritmo *Healing* es un algoritmo *sweep-line* encargado de realizar dos tareas: remover superposiciones entre primitivas y unir primitivas adyacentes, formando primitivas maximales en el eje x. Esto último es muy importante puesto que los algoritmos que son ejecutados despues de *Healing* suponen y utilizan el hecho de que las primitivas deben ser maximales en el eje x.

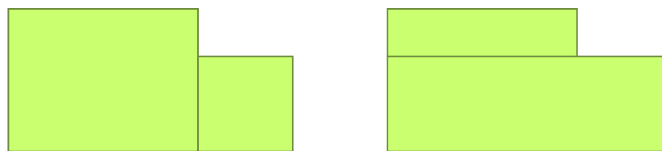

Figura 2.16: Primitivas no maximales y maximales en el eje x.

La Figura 2.16 muestra diversas primitivas. A la izquierda un conjunto de primitivas *no* maximales en el eje x mientras que a la derecha se ve otro conjunto de primitivas, que representan la misma figura, que sí son maximales en el eje x.

Input: Conjunto $\mathcal{L}$ de líneas.
Output: Conjunto de intersecciones entre líneas en $\mathcal{L}$.

1: Inicializar la cola de eventos $\mathcal{Q}$.
2: Inicializar la lista activa $\mathcal{T}$.
3: Insertar los vértices de los segmentos en $\mathcal{Q}$.
4: **while** $\mathcal{Q}$ no está vacío **do**
5: Obtener el siguiente evento (aquel con mayor coordenada y) p.
6: Sea $U(p)$ todos los segmentos cuyo punto superior es p.
7: Encontrar todos los segmentos en $\mathcal{T}$ que contengan p. Sea $L(p)$ el subconjunto de segmentos encontrados que tienen p como su punto inferior. Sea $C(p)$ el subconjunto de segmentos encontrados que contienen el punto p en su interior.
8: **if** $L(p) \cup U(p) \cup C(p)$ contiene más de un segmento **then**
9: Reportar p como punto de intersección.
10: **end if**
11: Eliminar de $\mathcal{T}$ los segmentos en $L(p) \cup C(p)$.
12: Insertar en $\mathcal{T}$ los segmentos en $U(p) \cup C(p)$. El orden de inserción debería ser el orden en que intersectan una *sweep-line* justo debajo de p, de izquierda a derecha. Si hay un segmento horizontal, quedaría al final de todos los segmentos que contienen a p.
13: /* Eliminar y reinsertar los elementos de $C(p)$ reinvierte su orden. */
14: **if** $U(p) \cup C(p) = \emptyset$ **then**
15: Sea s_l y s_r los vecinos a la izquierda y derecha de p en $\mathcal{T}$.
16: Encontrar nuevos eventos entre s_l, s_r y p.
17: **else**
18: Sea s' el segmento de $U(p) \cup C(p)$ de más a la izquierda en $\mathcal{T}$.
19: Sea s_l el vecino a la izquierda de s' en $\mathcal{T}$.
20: Encontrar nuevos eventos entre s_l, s' y p.
21: Sea s'' el segmento de $U(p) \cup C(p)$ de más a la izquierda en $\mathcal{T}$.
22: Sea s_r el vecino a la derecha de s'' en $\mathcal{T}$.
23: Encontrar nuevos eventos entre s'', s_r y p.
24: **end if**
25: **end while**

Algoritmo 2.1: Algoritmo de detección de intersecciones en segmentos de línea rectas.

Se puede especificar si se desean remover las superposiciones o no, sin embargo, en ambos casos, polígonos adyacentes son unidos.

El algoritmo *Healing* es llamado sobre toda la entrada, inmediatamente después de que *Windfrac* es ejecutado. Básicamente, una vez que se termine de aplanar y fracturar toda la entrada, la cual es realizada polígono a polígono, se debe mejorar la calidad de la información generada.

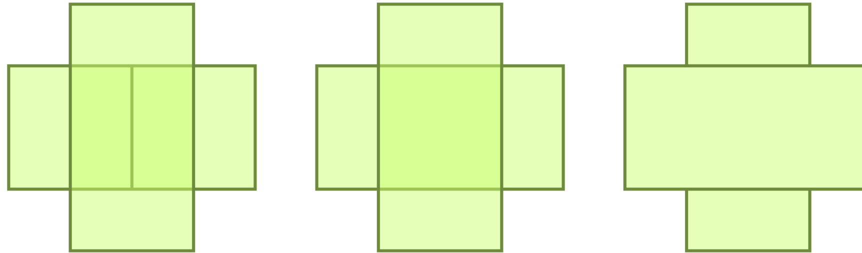

Figura 2.17: Ejemplo de resultados de *Healing*.

La Figura 2.17 muestra unos ejemplos del resultado de aplicar *Healing* a 3 primitivas. La figura de la izquierda son las primitivas originales, dos rectángulos adyacentes y un rectángulo superpuesto. La figura que se encuentra al medio es el resultado si se permiten superposiciones (sólo se unieron los rectángulos superpuestos). La figura de la derecha es el resultado si no se permiten superposiciones.

A grandes rasgos, el algoritmo utiliza una *sweep-line* para barrer todo el diseño de abajo hacia arriba. A medida que avanza, procesa de izquierda a derecha todas las figuras que intersectan esta *sweep-line*. Lo que realiza en este proceso es ver si los conjuntos de figuras deben ser unidos (si dos primitivas se tocan o superponen) o fracturados (luego de unir dos primitivas quizás hay que fracturar alguna para dar paso a la nueva). Si la *sweep-line* deja de intersectar a una primitiva (ya sea una nueva o una del diseño original), entonces, ésta es agregada al conjunto de primitivas, resultado del proceso de *Healing*.

Capítulo 3

Procedimiento

En este Capítulo se explican los pasos realizados para el desarrollo de la memoria. El procedimiento se separa en dos fases:

Sección 3.1, Estudio y Análisis de *Windfrac* Cómo se abordó el problema de documentar el funcionamiento del algoritmo *Windfrac*, basado principalmente en la implementación.

Sección 3.2, Mejora de *Windfrac* Qué fue lo que se decidió para mejorar los problemas de la implementación actual de *Windfrac*.

3.1. Estudio y Análisis de *Windfrac*

El primer paso fue estudiar detalladamente el paper "The Inside Story on Self Intersecting Polygons"[8], el cual es la base de la implementación de *Windfrac*.

Si bien el paper fue de gran ayuda para entender los conceptos utilizados por el algoritmo, como el *winding-number*, no fue muy útil para entender el algoritmo mismo. La razón de esto es simple, el paper explicaba un algoritmo ligeramente parecido al implementado. Cabe destacar que sí sirvió para tener una idea general del funcionamiento del algoritmo; para darse cuenta que es de tipo *sweep-line*, cuáles son los eventos y para tener una idea de cómo se procesa la lista activa..

Debido a esto, el funcionamiento detallado del algoritmo fue deducido, principalmente, a partir de la implementación misma[12]. Esto se realizó leyendo y analizando el código fuente, escrito en `C`, y estudiando casos de prueba, utilizando el depurador `gdb`, para ir paso a paso analizando el flujo de las instrucciones.

La implementación del algoritmo actual se divide en dos partes dependiendo del tipo de polígono que recibe, polígonos sólo con aristas horizontales o verticales (llamado caso *Manhattan*) y polígonos con aristas que poseen pendiente (llamado caso general). Debido a que la parte que maneja los casos *Manhattan* es mucho más simple que el caso general (ya que este último debe tomar en cuenta las posibles intersecciones entre aristas), se estudió primero el caso *Manhattan*.

Una vez entendido cómo funciona el caso *Manhattan*, se procedió a estudiar el caso general, el cual resultó ser mucho más complicado. Se procedió de la misma forma que con el caso *Manhattan*, estudiando el código fuente y utilizando el depurador para ir paso a paso ejecutando las instrucciones.

Adicionalmente, se buscó información asociada a la fractura de polígonos para buscar similitudes entre la literatura y la implementación, pero, sin éxito. La información que hay sobre fractura de polígonos es muy escasa e insuficiente, hacen simplificaciones sobre la entrada (algoritmos que funcionan sólo con polígonos simples) o proponen algoritmos de baja calidad (introducen muchos cortes innecesarios), entre otros.

Esta fase fue la que más tiempo tomó ya que la implementación de *Windfrac* es muy difícil de seguir y el algoritmo es bastante complejo.

3.2. Mejora de *Windfrac*

Una vez terminada la fase de estudio y análisis de la implementación actual del algoritmo, se procedió a ver cómo podría ser mejorado.

Los dos puntos más importantes que se debían tener en cuenta eran la mejora de la legibilidad (lo que mejora también la mantenibilidad) y la mejora de la calidad de los resultados, siempre manteniendo el desempeño y la correctitud del algoritmo actual.

Las opciones para mejorar el algoritmo eran realizar cambios menores a la implementación actual, reimplementar el mismo algoritmo o implementar un algoritmo completamente nuevo.

La primera opción no era válida puesto que la ilegibilidad de la implementación actual hacía imposible rescatar alguna línea de código del algoritmo mismo. Esto obligó a que fuera necesario, implementar todo desde cero, ya sea el mismo algoritmo o un nuevo algoritmo.

La tercera opción tampoco era válida debido a la poca información que existía al respecto. Todos los algoritmos existentes eran insuficientes para resolver el problema que soluciona el algoritmo *Windfrac*.

Por lo tanto, se decidió, por la segunda opción, reimplementar el mismo algoritmo utilizando funciones y estructuras de datos, de tal forma que sea más legible, y, tomando en cuenta los detalles acerca de la calidad. De esta forma se mantiene la correctitud y el desempeño, al menos teórico.

Cabe destacar que la reimplementación de *Windfrac* es de un algoritmo muy parecido al de la implementación actual, salvo por un par de diferencias. El nuevo algoritmo *Windfrac* está diseñado de una forma mucho más parecida al algoritmo de detección de intersecciones en segmentos de líneas rectas, explicado en el Capítulo 2. La principal razón de realizar estos cambios es para aumentar la legibilidad de la implementación.

Capítulo 4

Algoritmo *Windfrac*

En este Capítulo se explica el funcionamiento completo del algoritmo *Windfrac*, desde su definición hasta su implementación. Contiene las siguientes secciones:

Sección 4.1, Definición Se define formalmente el problema que resuelve *Windfrac*, estableciendo la entrada que recibe y la salida que debe generar.

Sección 4.2, El Algoritmo Se explica el funcionamiento completo del algoritmo de una forma abstracta, incluyendo las estructuras de datos utilizadas, el flujo del algoritmo y la generación de primitivas.

Sección 4.3, Detalles de la Implementación Actual Se explica cómo está implementado el algoritmo actual junto con los problemas que tiene.

Sección 4.4, Detalles de la Reimplementación Se explica cómo fue reimplementado el algoritmo para resolver los problemas de la implementación actual.

4.1. Definición[7]

Windfrac es un algoritmo para resolver el siguiente problema:

Problema 2 *Dado un conjunto de aristas $\mathcal{E} = e_1, e_2, \ldots, e_n$ con vértices alineados en una grilla que representan un polígono cualquiera, generar un conjunto de primitivas sin superposiciones y alineadas en la misma grilla tales que representen de la mejor forma posible el polígono original.*

En otras palabras, *Windfrac* recibe un polígono cualquiera (simple o complejo) definido por sus aristas cuyos vértices están alineados sobre una grilla dada, y debe generar un conjunto de primitivas (rectángulos, trapecios horizontales o triángulos, que no son más que trapecios horizontales degenerados en donde uno de sus dos lados paralelos tiene longitud nula) que no se superpongan entre ellas y que la unión de sus interiores sea lo más parecido al interior del polígono original.

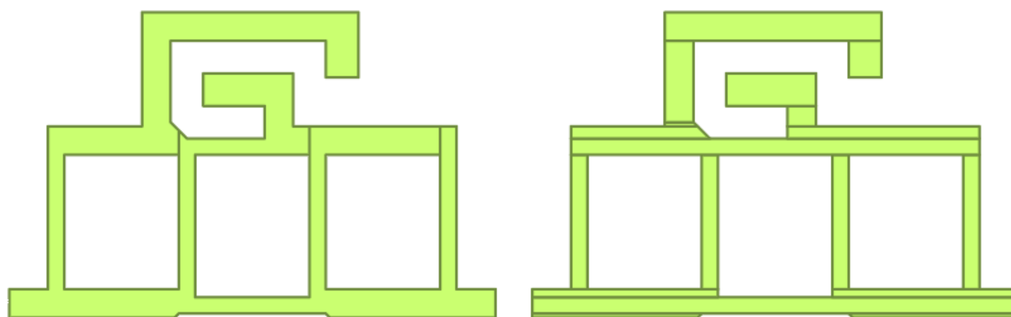

Figura 4.1: Una fractura de un polígono.

Se dice que dos polígonos están superpuestos si sus interiores se superponen. Debido a que el interior de los polígonos se considera abierto, entonces, los límites del polígono no cuentan, es decir, las aristas y vértices no son tomados en cuenta al momento de determinar si dos polígonos están superpuestos o no. Esto último implica que, dos polígonos que comparten sólo aristas o vértices, no están superpuestos.

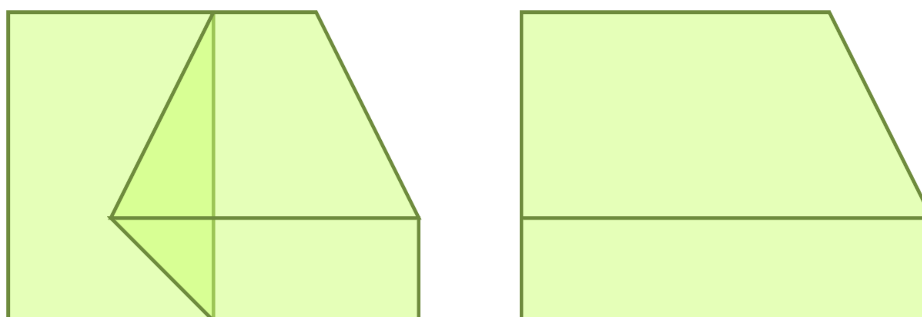

Figura 4.2: Ejemplos de primitivas superpuestas y primitivas no superpuestas.

La Figura 4.2 muestra a la izquierda un conjunto de primitivas que sí están superpuestas porque su interior está superpuesto. A la derecha, se ve un conjunto de primitivas que no están superpuestas debido a que sólo comparten aristas y vértices.

Debido a que el interior de los polígonos representan el material en el diseño de circuitos digitales, entonces, es deseable que el resultado de la partición sea lo más parecido al polígono original, alterando así lo menos posible la definición de información del diseño. En efecto, es un requisito que en ningún momento (ya sea al ejecutar *Windfrac*, o en los pasos que siguen hasta la generación del archivo que contiene la información de la máscara) el resultado de la partición pueda tener diferencias de una unidad de resolución o más.

Como se puede ver en la Figura 4.3, la partición introdujo dos nuevos vértices, en los puntos encerrados por los círculos rojos. Lo importante es que estos puntos fueron movidos *menos* de una unidad de resoluciíon, por lo tanto, cumple con la restricción de “parecerse” al polígono original. Pero, si el resultado de la partición hubiera sido el de la Figura 4.4, entonces, el conjunto de primitivas resultante “no se hubiera parecido” al polígono original porque el vértice nuevo rodeado por el círculo rojo de abajo fue movido más de una unidad de resolución.

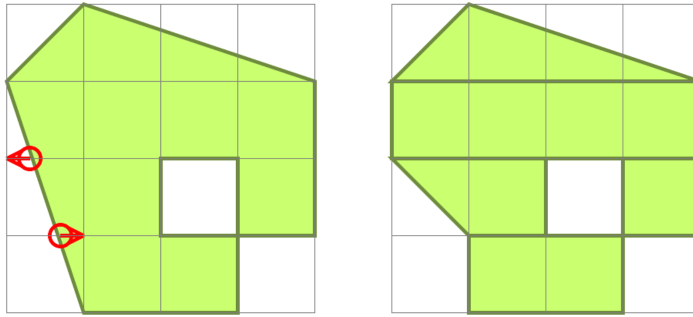

Figura 4.3: Particiones “parecidas” al polígono original.

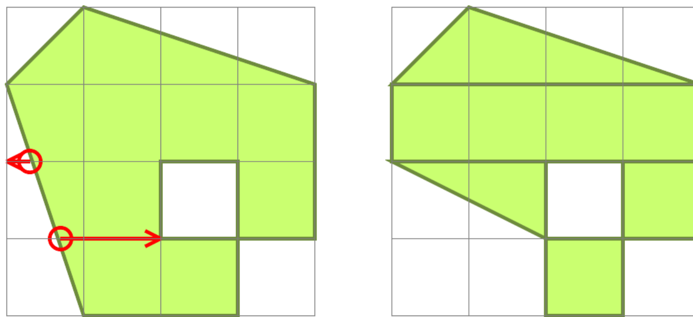

Figura 4.4: Particiones “no parecidas” al polígono original.

4.2. El Algoritmo[12][8]

Existen muchas formas distintas de particionar un polígono en un conjunto de primitivas tales que no se superpongan. La definición del problema no especifica que un conjunto de primitivas sea mejor que otro, sus únicas restricciones, como ya fue mencionado anteriormente, son que deben “parecerse” al polígono original y no deben superponerse.

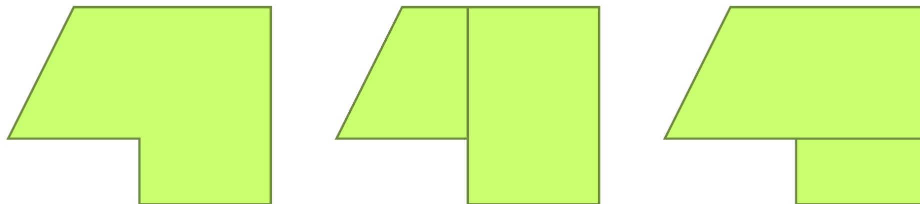

Figura 4.5: Particiones válidas de un polígono.

La Figura 4.5 muestra dos particiones válidas (al medio y a la derecha) del polígono de la izquierda. En ambos casos, se generan dos primitivas, un rectágulo y un trapecio horizontal. Por lo tanto, como se puede apreciar, no hay una forma única de hacerlo y tampoco hay una forma mejor que otra.

Antes de hablar sobre el algoritmo que genera estas particiones, es importante notar un par de detalles geométricos acerca del problema. Las primitivas generadas pueden ser sólo rectángulos o trapecios horizontales, esto implica dos cosas: muchos cortes introducidos son

horizontales y dos aristas muy lejos en el eje y no pueden participar en la generación de la misma primitiva.

Que los cortes introducidos sean horizontales es directo de la restricción de la generación de primitivas. Todas las primitivas generadas tienen lados horizontales, los rectángulos tienen dos lados verticales y dos lados horizontales, los trapecios horizontales tienen sus dos lados paralelos horizontales y los triángulos tienen un lado horizontal (puesto que es un trapecio horizontal degenerado). Por lo tanto, cada vez que se genere una primitiva en donde no hay aristas horizontales involucradas, se introducirá un corte horizontal. En la Figura 4.5 se puede ver como, en la partición de la derecha, un corte horizontal fue introducido para poder generar el rectángulo inferior.

Que dos aristas muy lejos en el eje y no puedan participar en la generación de la misma primitiva también es directo. Las primitivas están compuestas de aristas horizontales y aristas no horizontales. Las aristas no horizontales deben estar a la misma altura en el eje y pero a distintas alturas en el eje x, de lo contrario no podrían generar una curva cerrada, que es la definición de un polígono. La Figura 4.6 muestra cómo es imposible que las aristas azules participen en la creación de la primitiva inferior con área verde oscuro, mientras que las dos aristas rojas sí podrían participar puesto que están a la misma altura en el eje y.

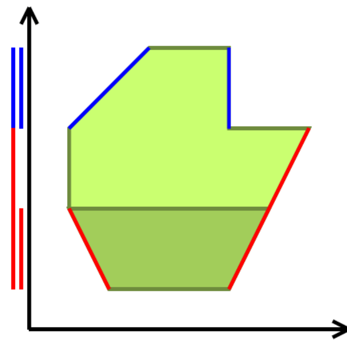

Figura 4.6: Dos aristas separadas en el eje y no pueden participar en la generación de la misma primitiva.

La importancia de estos detalles es que sugieren resolver el problema con un algoritmo *sweep-line*. Los cortes introducidos son a lo largo de la *sweep-line*, que debe ser horizontal, mientras que las únicas aristas capaces de generar primitivas son aquellas tocando la *sweep-line* puesto que están a la misma altura en el eje y.

4.2.1. Algoritmo *Windfrac* a Grandes Rasgos

Como fue dicho anteriormente, el funcionamiento del algoritmo fue deducido a partir de la implementación actual de *Windfrac*. Se tomó la decisión de explicar la versión mejorada en vez de la versión original, debido a que la original funcionaba de formas muy curiosas en ciertos momentos haciendo que explicarla sea más complicado. La versión mejorada no difiere mayormente de la original, realiza exactamente lo mismo, de la misma forma a grandes rasgos, sólo que ciertos aspectos específicos son realizados de forma distinta.

La idea del algoritmo *Windfrac* es utilizar una *sweep-line* horizontal para barrer el espacio desde abajo hacia arriba. Se trabaja sobre las aristas del polígono que intersectan la *sweep-line* y en cada *sweep-line* se procesarán las aristas de izquierda a derecha calculando el *winding-number*. Con este número se puede saber en qué momento una región es considerada interior para así decidir si se debe generar una primitiva con las aristas involucradas o si se debe continuar procesando aristas.

Para simplificar la explicación, no serán considerados los polígonos complejos (aquellos con autointersecciones). Este caso será explicado después.

Al ser *Windfrac* un algoritmo *sweep-line*, hay que definir cuáles serán los eventos y cuál será la lista activa.

Debido a que los cortes horizontales pueden ser introducidos sólo en los vértices del polígono, entonces, se pueden establecer los eventos como éstos. Luego, cada vez que la *sweep-line* pare en alguno de estos eventos , se generará una primitiva con la información justo debajo de la *sweep-line*. De esta forma, el corte introducido al generar la primitiva será a lo largo de la *sweep-line* y a la altura de alguno de los vértices del polígono. La Figura 4.7 muestra los vértices introducidos a la cola de eventos.

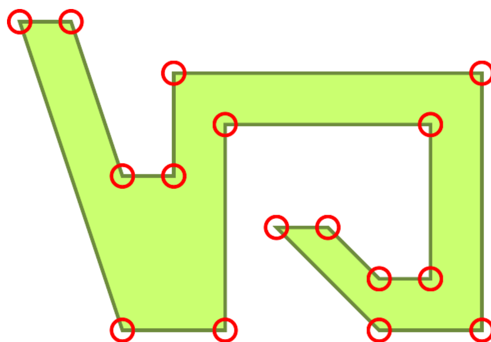

Figura 4.7: Vértices introducidos a la cola de eventos.

Por lo tanto, se necesita que la estructura de la cola de eventos permita insertar nuevos eventos, remover eventos, mirar el tope de la cola (que debe apuntar al siguiente evento) y obtener los eventos de forma ordenada (para poder procesar el polígono de abajo hacia arriba y, dada una *sweep-line*, de izquierda a derecha). A la cola de eventos se le llamará $\mathcal{Q}$.

La lista activa es más fácil, de hecho, ya se mencionó. La lista activa son todas las aristas que intersectan con la *sweep-line* incluyendo aquellas que intersectan justo en uno de sus puntos vértices (se podría decir que las aristas son segmentos cerrados). Como la lista activa se procesa de izquierda a derecha, entonces, conviene que esta lista contenga las aristas ordenadas por su coordenada x de izquierda a derecha. A la lista activa se le llamará $\mathcal{A}$. La Figura 4.8 muestra una *sweep-line* junto con la lista activa, que serían las aristas marcadas en azul.

Habiendo explicado las distintas componentes, el Algoritmo 4.1 muestra a grandes rasgos cómo se comporta *Windfrac*.

Los pasos 1 y 2 son bien sencillos, es simplemente crear estructuras de datos vacías para almacenar los eventos y la lista activa.

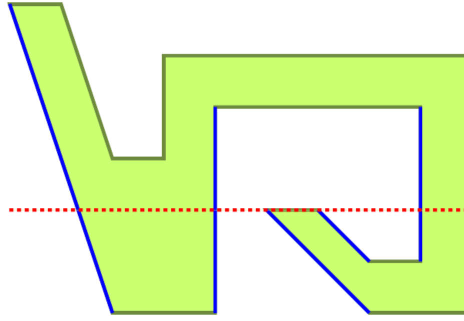

Figura 4.8: Las aristas azules pertenecen a la lista activa dada por la *sweep-line* roja.

Input: Conjunto $\mathcal{E}$ de aristas.
Output: Conjunto de primitivas que representan el polígono definido por $\mathcal{E}$.
1: Inicializar la lista de eventos $\mathcal{Q}$ vacía.
2: Inicializar la lista activa $\mathcal{A}$ vacía.
3: Insertar en $\mathcal{Q}$ la tupla (x, y, e) ((x, y) es la coordenada del vértice y e es la arista a la que pertenece) de los vértices de todas las aristas en $\mathcal{E}$. La inserción debe ser en *orden*.
4: **while** $\mathcal{Q}$ no está vacío **do**
5: Obtener la siguiente *sweep-line* p mirando el tope de $\mathcal{Q}$.
6: $ActualizarListaActiva(\mathcal{A}, p, \mathcal{Q})$
7: $GenerarPrimitivas(\mathcal{A}, p)$
8: $Actualizar2ListaActiva(\mathcal{A}, p, \mathcal{Q})$
9: **end while**

Algoritmo 4.1: Algoritmo *Windfrac* a grandes rasgos.

El paso 3 es iterar sobre las aristas del polígono insertando las coordenadas (x, y) de cada vértice, junto con la arista a la que pertenecen, a la cola de eventos $\mathcal{Q}$. Cabe destacar que la misma coordenada (x, y) puede estar insertada más de una vez en la cola de eventos si y sólo si pertenecen a distintas aristas. Como se dijo anteriormente, las intersecciones aún no serán consideradas.

El paso 4 es iterar mientras la cola de eventos no esté vacía. Una vez que todos los eventos hayan sido procesados (la *sweep-line* terminó de barrer el espacio), el polígono va a estar completamente particionado.

El paso 5 es simplemente obtener la siguiente *sweep-line* mirando la coordenada y del siguiente evento en $\mathcal{Q}$.

El paso 6 actualiza la lista activa utilizando la información de la nueva *sweep-line*. Esta actualización involucra agregar las nuevas aristas que intersectan la *sweep-line*. Debido a que la cola de eventos almacena las tuplas (x, y, e) es muy fácil obtener la siguiente arista que debe ser insertada (si (x, y) es el vértice de más abajo de la arista e) o removida (si (x, y) es el vértice de más arriba de la arista e).

El paso 7 genera todas las primitivas posibles con la información bajo esta nueva *sweep-line* y la nueva lista activa.

Finalmente, el paso 8 vuelve a actualizar la lista activa. Debido a que la generación de

primitivas se realiza con la información bajo la *sweep-line*, no es posible remover aristas en el paso 6, por lo que este paso elimina las aristas que dejarán de intersectar a la *sweep-line*.

Un punto importante que hay que tener en cuenta es que en la primera iteración del ciclo definido en los pasos 4-9 no se generan primitivas. Esto es porque no hay información debajo de la primera *sweep-line*. Por lo tanto, ese ciclo lo único que hace es inicializar la lista activa con las aritas que están tocando con la primera *sweep-line*, definida por el primer evento en $\mathcal{Q}$. La Figura 4.9 muestra la lista activa (aquellas aristas marcadas en azul) asociada a la primera *sweep-line*. También se puede notar que es imposible que hayan primitivas bajo la primera *sweep-line* puesto que no hay más vértices bajo ésta.

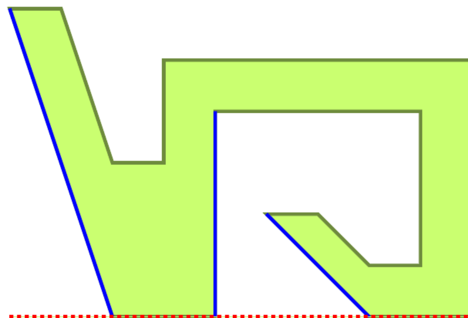

Figura 4.9: La primera lista activa, definida por la primera *sweep-line*.

4.2.2. Actualización de la Lista Activa

La actualización de la lista activa es un procedimiento relativamente simple. Como se dijo anteriormente, dada una nueva *sweep-line* hay que agregar aquellas aristas en $\mathcal{E}$ que ahora están intersectando la *sweep-line*.

Input: Lista activa $\mathcal{A}$, *sweep-line* p y el conjunto $\mathcal{E}$ de aristas.
1: **for** Cada arista s en $\mathcal{A}$, ordenadas de izquierda a derecha **do**
2:     **for** Cada arista e en $\mathcal{E}$ que esté a la izquierda de s al intersectar la *sweep-line* p **do**
3:         Insertar e a la izquierda de s en $\mathcal{A}$.
4:         Eliminar e de $\mathcal{E}$.
5:     **end for**
6: **end for**

Algoritmo 4.2: Actualización de la lista activa del algoritmo *Windfrac*.

En la Figura 4.10 se puede ver como cambia la lista activa al pasar de una *sweep-line* a la siguiente. Las dos aristas rojas de arriba son insertadas a la lista activa mientras que las dos aristas rojas de abajo serán removidas en la segunda actualización de ésta.

Un punto importante que vale la pena destacar es que, debido a que no se están tomando en cuenta las intersecciones, el orden de las aristas dentro de la lista activa nunca se ve alterada. Una vez insertada una arista en su posición, las únicas operaciones que son ejecutadas después, son insertar una arista antes o quitar una arista, pero nunca invertir aristas dentro de la lista activa. La razón de ésto es bien simple, puesto que no hay intersecciones, una arista

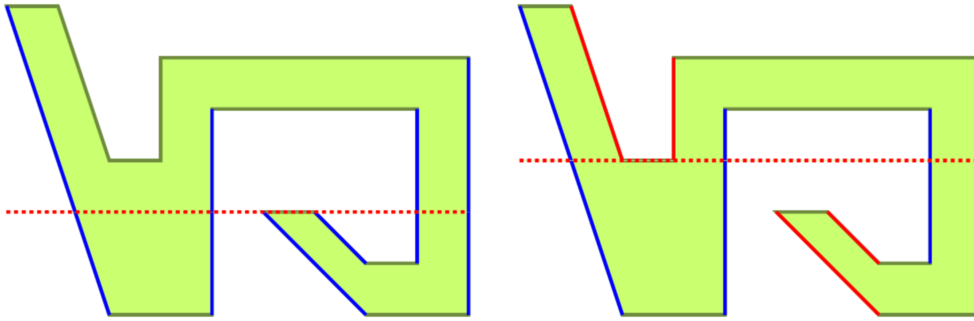

Figura 4.10: Actualización de la lista activa.

que empezó estando a un lado de otra nunca pasará a estar al otro lado sin una intersección de por medio.

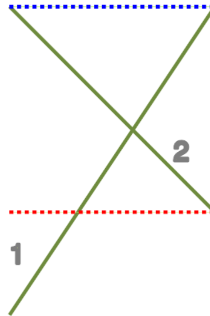

Figura 4.11: Intercambio de orden en la lista activa sólo cuando hay intersecciones.

La Figura 4.11 muestra que en la *sweep-line* roja la arista 1 está a la izquierda de la arista 2. Pero, en la *sweep-line* azul la arista 1 está a la *derecha* de la arista 2. Este tipo de intercambio en la posición con respecto a la coordenada x al momento de intersectar la *sweep-line* puede suceder sólo cuando hay intersecciones entre aristas.

4.2.3. Generación de las Primitivas

La generación de primitivas es el procedimiento más complejo del algoritmo. Como el algoritmo es de tipo *sweep-line*, todo el procesamiento se realiza sobre ésta. La pregunta es ¿cómo procesar la lista activa y cómo decidir cuando generar una primitiva?.

La respuesta a esta pregunta puede no ser tan directa. Primero, se verá cómo se procesa la lista activa para luego ver cómo se decide el momento exacto para generar una primitiva.

La lista activa se procesa de izquierda a derecha, y, a medida que se va procesando se va calculando el *winding-number*. Con el *winding-number* se puede saber si una arista dada es una arista "izquierda", una arista "derecha" o una arista "interna". Las aristas "izquierdas" son aquellas que no contienen información a su izquierda (al momento de generar una primitiva, este tipo de arista formaría parte de la izquierda de la primitiva), las aristas "derechas" son aquellas que no contienen información a su derecha (formarían parte de la derecha de una primitiva generada) y las aristas "internas" son aquellas que tienen información a ambos lados

(no forman parte de ninguna primitiva generada puesto que la información a su izquierda y derecha sería utilizada para generar la misma primitiva, haciendo uso de aristas “izquierdas” y “derechas”). Es importante notar que la cualidad de ser arista izquierda, derecha o interna depende de la *sweep-line*.

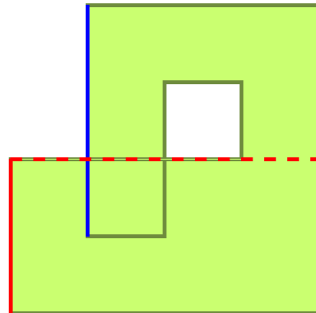

Figura 4.12: Los distintos tipos de arista dependiendo de donde se encuentra el interior del polígono.

En la Figura 4.12 se puede ver que, dada la *sweep-line* (línea roja punteada), las aristas rojas de la izquierda y de la derecha son aristas “izquierda” y “derecha” respectivamente, mientras que la arista azul es una arista “interna” por debajo de la *sweep-line*, pero, es una arista “izquierda” por arriba. Esto significa que la arista azul no definirá la primitiva con la información en esta *sweep-line*, sin embargo, puede que defina una primitiva en la siguiente *sweep-line*.

Al ir calculando el *winding-number* también es posible determinar si una región del polígono es considerada interior o exterior. Dependiendo de donde se encuentre el interior del polígono al momento de analizar una de las aristas de la lista activa se decide si se debe generar una primitiva o no. Cuando se decide generar una primitiva, ésta no es generada instantáneamente, sino que se espera hasta escanear una arista “derecha”. Cuando se escanea una arista “derecha” y se ha decidido generar una primitiva, entonces, la primitiva es generada desde la última arista “izquierda” escaneada, hasta esta arista “derecha” y desde la base de cualquiera de estas aristas hasta la *sweep-line*. La Figura 4.13 muestra de forma gráfica la generación de la primitiva dadas las aristas involucradas. La primitiva gris es aquella que será generada en esta iteración. Las aristas rojas son las que definen la primitiva, siendo aristas “izquierda” y “derecha”.

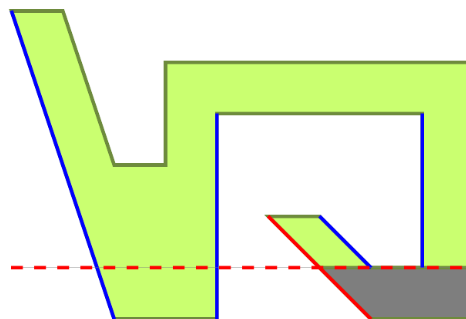

Figura 4.13: Generación de una primitiva.

Una vez se haya generado la primitiva, todas las aristas que estuvieron involucradas en esta generación (ya sea que definieron la primitiva o simplemente estuvieron en el interior de la primitiva generada) deben ser recortadas para que el algoritmo pueda continuar de forma normal. Si no se realiza este recorte, el algoritmo generaría primitivas superpuestas.

Los detalles del algoritmo para generar las primitivas se puede ver en el Algoritmo 4.3.

Input: Lista activa $\mathcal{A}$ y la *sweep-line* p.
1: Sea w_a el *winding-number* que va por arriba de p.
2: Sea w_b el *winding-number* que va por abajo de p.
3: $w_a \leftarrow 0$
4: $w_b \leftarrow 0$
5: **for** Cada arista s en $\mathcal{A}$, ordenadas de izquierda a derecha **do**
6:   **if** s está por encima de p **then**
7:     Actualizar w_a con s.
8:     $p_i \leftarrow GeneracionInsercion(w_a, w_b)$
9:     Continuar.
10:   **end if**
11:   **if** w_b es igual a 0 (s es una arista “izquierda”) **then**
12:     $l \leftarrow s$
13:     $p_l \leftarrow GeneracionIzquierda(w_a, w_b, s, p)$
14:   **end if**
15:   Actualizar w_b con s.
16:   **if** s cruza p **then**
17:     Actualizar w_a con s.
18:     $p_c \leftarrow GeneracionCruce(w_a, w_b)$
19:   **else**
20:     Eliminar s de $\mathcal{A}$.
21:   **end if**
22:   **if** w_b es igual a 0 (s es una arista “derecha”) **then**
23:     $p_r \leftarrow GeneracionDerecha(w_a, w_b, s, p)$
24:     **if** p_i o p_l o p_c o p_r (se forzó la generación de una primitiva) **then**
25:       Generar primitiva desde l hasta s y desde la base de l hasta p.
26:       **for** Cada arista a en $\mathcal{A}$ desde l hasta s **do**
27:         Recortar la parte inferior de a hasta p.
28:       **end for**
29:     **end if**
30:   **end if**
31: **end for**

Algoritmo 4.3: Generación de primitivas del algoritmo *Windfrac*.

Lo único que queda por explicar es cómo se toma la decisión de generar una primitiva basado en la arista que se está escaneando y en los *winding-number*s de arriba y de abajo de la *sweep-line* dada. Es decir, las funciones $GeneracionInsercion$, $GeneracionIzquierda$, $GeneracionCruce$ y $GeneracionDerecha$.

A continuación se explicarán estos cuatro casos en donde se decide si generar una primitiva o no.

Generación por Inserción

Este caso es chequeado sólo cuando se está escaneando una arista que está por encima de la *sweep-line* pero que la está tocando, es decir, una arista que tiene su vértice más abajo justo en la *sweep-line*.

Cuando se analiza este tipo de aristas, se debe forzar la generación de una primitiva sólo si es que hay información bajo la *sweep-line* y no hay información sobre ésta. Es decir, una primitiva debe ser generada si es que una arista por encima de la *sweep-line* es tal que introduce un "agujero".

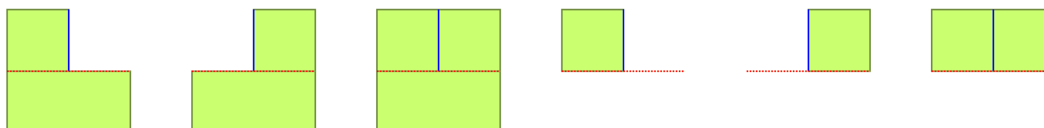

Figura 4.14: Generación de primitivas por inserción.

En la Figura 4.14 se puede ver claramente que el primer caso es el único que debería forzar la generación de una primitiva. El segundo caso no es necesario ya que, para que ese tipo de figura se forme, la arista izquierda debe terminar en la *sweep-line* u otra arista activa debe haber introducido el "agujero", forzando la generación en ambos casos. Se puede ver que el tercer caso no debe generar una primitiva puesto que hay información en todas partes, luego, cualquier corte introducido sería innecesario. Los últimos 3 tampoco deben generar primitivas porque no hay información bajo la *sweep-line*. El Algoritmo 4.4 tiene el detalle de cómo se decide en base al *winding-number*.

Input: El *winding-number* arriba y abajo, w_a y w_b respectivamente.
Output: Si esta arista fuerza que se genere una primitiva.
1: **if** $w_a = 0 \wedge w_b \neq 0$ **then**
2:    **return** Verdadero
3: **end if**
4: **return** Falso

Algoritmo 4.4: Generación de primitivas por inserción.

Generación por la Izquierda

Este caso es chequeado sólo cuando la arista que está siendo escaneada es una arista "izquierda" por debajo de la *sweep-line*, es decir, cuando no hay información a la izquierda de la arista y debajo de la *sweep-line*.

En este tipo de aristas, se debe forzar la generación de una primitiva cuando la arista siendo escaneada termina justo en la *sweep-line*, o cuando la arista cruza la *sweep-line*, pero, hay data a la izquierda de la arista y arriba de la *sweep-line*.

En la Figura 4.15 se puede ver que los primeros dos y los últimos dos casos son aquellos en los cuales se debe forzar la generación de una primitiva. El Algoritmo 4.5 muestra los

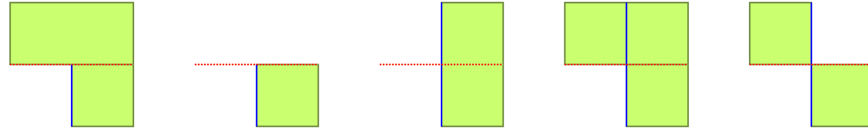

Figura 4.15: Generación de primitivas por la izquierda.

detalles de esta decisión. Cabe destacar que, a esta altura del Algoritmo 4.3, el *winding-number* arriba de la *sweep-line* aún no ha sido actualizado, por lo tanto, la condición $w_a \neq 0$ revisa, efectivamente, a la izquierda de la arista por sobre la *sweep-line*.

Input: El *winding-number* arriba y abajo, w_a y w_b respectivamente, la arista siendo escaneada s y la *sweep-line* p.
Output: Si esta arista fuerza que se genere una primitiva.
1: **if** s tiene su vértice superior en p **then**
2: **return** Verdadero
3: **end if**
4: **if** $w_a \neq 0$ **then**
5: **return** Verdadero
6: **end if**
7: **return** Falso

Algoritmo 4.5: Generación de primitivas por la izquierda.

Generación por Cruce

Este caso es revisado sólo cuando una arista cruza la *sweep-line*. Es de suma importancia para aristas "internas" puesto que, al no ser ni "izquierdas" ni "derechas", no caen en estos casos.

En este tipo de aristas, se debe forzar la generación de una primitiva si es que no hay información a la derecha de la arista por sobre la *sweep-line*. En el caso de que no haya información a la izquierda de la arista y por sobre la *sweep-line* cae en la situación de generación por inserción y por la izquierda.

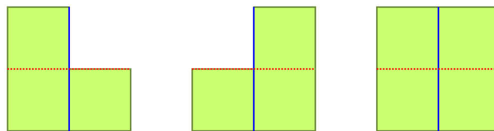

Figura 4.16: Generación de primitivas por cruce.

En la Figura 4.16 se puede ver que el único caso en el cual se forzaría la generación de la primitiva sería en el primero. En el segundo, la generación sería forzada en otro caso, mientras que en el tercero no es necesario generar una primitiva. El Algoritmo 4.6 muestra los detalles de cómo se fuerza la generación de la primitiva. Cabe notar que, a esta altura del Algoritmo 4.3, el *winding-number* arriba de la *sweep-line* ya fue actualizado, por lo tanto, el chequeo es a la derecha de la arista.

Input: El *winding-number* arriba y abajo, w_a y w_b respectivamente
Output: Si esta arista fuerza que se genere una primitiva.
1: **if** $w_a = 0 \land w_b \neq 0$ **then**
2: **return** Verdadero
3: **end if**
4: **return** Falso

Algoritmo 4.6: Generación de primitivas por cruce.

Generación por la Derecha

Este caso es simétrico a la generación por la izquierda. Se revisa cuando la arista siendo escaneada es una arista "derecha".

Al igual que en el caso por la izquierda, se debe forzar una primitiva cuando la arista termina en la *sweep-line* o, si es que la cruza, hay información arriba a la derecha.

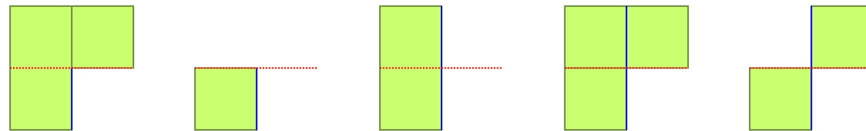

Figura 4.17: Generación de primitivas por la derecha.

La Figura 4.17 muestra que el único caso en el cual no es necesario forzar una primitiva es en el que se ubica al medio. En los otros 4, la primitiva debe ser generada. El Algoritmo 4.7 muestra los detalles.

Input: El *winding-number* arriba y abajo, w_a y w_b respectivamente, la arista siendo escaneada s y la *sweep-line* p.
Output: Si esta arista fuerza que se genere una primitiva.
1: **if** s tiene su vértice superior en p **then**
2: **return** Verdadero
3: **end if**
4: **if** $w_a \neq 0$ **then**
5: **return** Verdadero
6: **end if**
7: **return** Falso

Algoritmo 4.7: Generación de primitivas por la derecha.

4.2.4. Segunda actualización de la Lista Activa

La segunda actualización de la lista activa es igual de simple que la primera actualización. Lo que hay que hacer es, básicamente, eliminar aquellas aristas que dejarán de intersectarse con la *sweep-line*.

El Algoritmo 4.8 explica los detalles de cómo es realizado.

Input: Lista activa $\mathcal{A}$, *sweep-line* p y el conjunto $\mathcal{E}$ de aristas.
1: **for** Cada arista s en $\mathcal{A}$, ordenadas de izquierda a derecha **do**
2: **if** s está tocando por arriba a p **then**
3: Remover s de $\mathcal{A}$.
4: **end if**
5: **end for**

Algoritmo 4.8: Segunda actualización de la lista activa del algoritmo *Windfrac*.

En la Figura 4.18 se puede ver que las dos aristas rojas de arriba fueron insertadas en la actualización, mientras que las dos aristas rojas de abajo son removidas en la segunda actualización de esta lista activa.

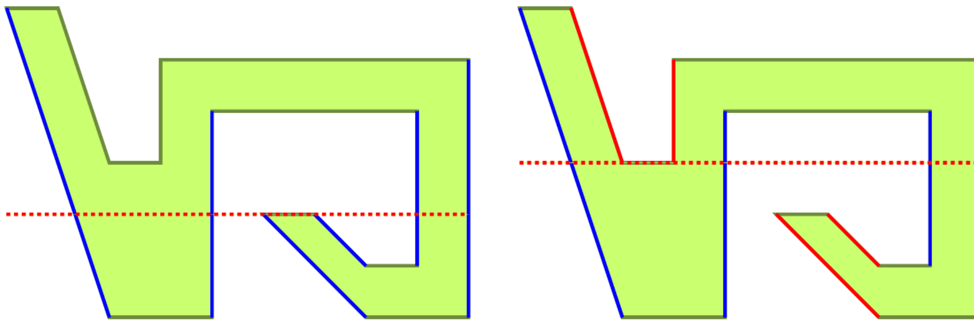

Figura 4.18: Inserción de nuevas aristas y eliminación de aristas procesadas.

4.2.5. Intersecciones

Luego de haber explicado cómo funciona todo el algoritmo sin intersecciones, introducir el tema de las intersecciones es relativamente simple.

Las intersecciones introducen varios detalles al algoritmo. Lo primero que hay que notar es que las intersecciones pueden introducir cortes en el polígono para generar primitivas. Lo segundo, es que una intersección invierte el orden de las aristas involucradas, como se puede ver en la Figura 4.19.

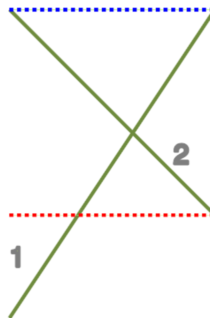

Figura 4.19: Intercambio de orden en la lista activa en una intersección.

Input: Lista activa $\mathcal{A}$, *sweep-line* p y el conjunto $\mathcal{E}$ de aristas.
1: **for** Cada arista s en $\mathcal{A}$, ordenadas de izquierda a derecha **do**
2: **for** Cada arista e en $\mathcal{E}$ que esté a la izquierda de s al intersectar la *sweep-line* p **do**
3: Insertar e a la izquierda de s en $\mathcal{A}$.
4: Sea l la arista a la izquierda de e en $\mathcal{A}$.
5: $DetectarInterseccion(l, e, \mathcal{Q})$
6: Eliminar e de $\mathcal{E}$.
7: **end for**
8: Sea l la arista a la izquierda de s en $\mathcal{A}$.
9: $DetectarInterseccion(l, s, \mathcal{Q})$
10: **end for**

Algoritmo 4.9: Actualización de la lista activa del algoritmo *Windfrac* tomando en cuenta posibles intersecciones entre aristas.

Se pueden resolver ambos problemas adaptando el algoritmo actual para que se parezca al algoritmo de detección de intersecciones. Para hacer esto, basta con modificar el algoritmo que actualiza y el que reactualiza la lista activa. En cada momento que se agrega una nueva arista, se detectan por intersecciones entre la arista insertada y sus vecinos, además, cada vez que se elimina una arista se detectan intersecciones entre las dos aristas vecinas que ahora son vecinas entre sí. Si se detecta una intersección, entonces, se calcula y se agrega la tupla (x, y, e_1, e_2) a la lista de eventos. Finalmente, si el evento que se está procesando pertenece a una intersección, entonces, se invierten las aristas involucradas en la intersección y se detectan intersecciones entre los nuevos vecinos de ambas aristas. Una vez realizada esta actualización de la lista activa, el algoritmo continúa como siempre.

Input: Lista activa $\mathcal{A}$, *sweep-line* p y el conjunto $\mathcal{E}$ de aristas.
1: **for** Cada arista s en $\mathcal{A}$, ordenadas de izquierda a derecha **do**
2: **if** s ya no intersecta la *sweep-line* p **then**
3: Sea l la arista a la izquierda de s en $\mathcal{A}$.
4: Sea r la arista a la derecha de s en $\mathcal{A}$.
5: Eliminar s de $\mathcal{A}$.
6: $DetectarInterseccion(l, r, \mathcal{Q})$
7: **end if**
8: **if** p es un evento perteneciente a intersección de aristas **then**
9: Sea a_1 y a_2 las aristas involucradas en la interseccion.
10: Sea l la arista a la izquierda de a_1 y r la arista a la derecha de a_2.
11: Invertir posiciones entre a_1 y a_2.
12: $DetectarInterseccion(l, a_2, \mathcal{Q})$
13: $DetectarInterseccion(a_1, r, \mathcal{Q})$
14: **end if**
15: **end for**

Algoritmo 4.10: Segunda actualización de la lista activa del algoritmo *Windfrac* tomando en cuenta posibles intersecciones entre aristas.

Los Algoritmos 4.9 y 4.10 muestran la adaptación de las funciones $ActualizarListaActiva$ y $Actualizar2ListaActiva$ respectivamente, tomando en cuenta las intersecciones. El detalle

de la función *DetectarInterseccion* se puede ver en el Algoritmo 4.11. Los detalles sobre cómo saber cuando dos segmentos de líneas rectas se intersectan se pueden ver en [9].

Input: Dos aristas vecinas a_1 y a_2, y la cola de eventos $\mathcal{Q}$
1: **if** a_1 y a_2 se intersectan **then**
2: Sea i_y la coordenada y del punto de intersección.
3: Insertar i_y en $\mathcal{Q}$ siempre y cuando no haya sido insertado anteriormente.
4: **end if**

Algoritmo 4.11: Detección de intersecciones y actualización de la cola de eventos del algoritmo *Windfrac*.

4.2.6. Aproximación a la Grilla

Debido a que ciertos vértices del polígono e intersecciones pueden no caer en la grilla, entonces, es necesario mover a grilla los vértices de las primitivas generadas.

No hay una única forma de hacer ésto. Los únicos puntos importantes a tomar en cuenta es que el resultado no se puede mover más de una unidad de resolución, no puede ser tal que genere primitivas superpuestas y debe ser consistente. Que sea consistente, significa que si en cierta situación se realiza cierta aproximación, entonces, en todas las situaciones iguales se debe hacer la misma aproximación.

4.3. Detalles de la Implementación Actual

Si bien la implementación del algoritmo sigue la misma idea que la explicación, hay muchas decisiones de implementación que difieren bastante de la explicación teórica. En particular, la entrada del algoritmo, la implementación de la lista activa y de los eventos, y la forma de manejar las intersecciones.

Las razones de por qué el algoritmo se implementó de ese modo son desconocidas debido a que no hay documentación y no es posible contactar al autor.

4.3.1. Definición de la Función *Windfrac*

La primera diferencia es la definición de la función que implementa el algoritmo. Esta función recibe como argumentos una lista de puntos (en precisión entera y en punto flotante con precisión doble) que representan los vértices del polígono, por lo tanto, es necesario recorrerlos creando aristas con cada par de puntos.

La estructura de datos que contiene la información de las aristas es bien compleja. Cada arista almacena la coordenada de más abajo de la arista (independiente de su dirección) junto con las diferencias hacia el siguiente punto. Estas coordenadas se guardan tanto en precisión entera como en punto flotante con precisión doble. Además, posee un valor que indica la dirección de la arista al intersectar una línea horizontal. Junto a ésto, se almacenan

punteros hacia la arista anterior y la siguiente de la lista activa, un puntero hacia el siguiente segmento que compone una arista y un puntero hacia la siguiente arista duplicada. Después se explicará cómo es usada esta información.

4.3.2. Almacenamiento de Aristas

Las aristas creadas son almacenadas en una cola de prioridad (implementada con un *heap*) en donde el primer elemento es aquella arista cuya coordenada principal (dada la estructura de datos que almacena las aristas, la principal sería la que está más abajo, puesto que la otra coordenada se implementa como diferencias desde ésta) se encuentra más abajo. Si dos aristas tienen su coordenada principal a la misma altura, entonces, estará primera aquella que está más a la izquierda. Si dos coordenadas tienen su coordenada principal en el mismo punto, estará primera aquella que se encuentra "a la izquierda" (calculado utilizando el producto cruz entre los vectores involucrados). Si dos aristas tienen ambas coordenadas en el mismo punto, estará primera aquella con dirección hacia abajo.

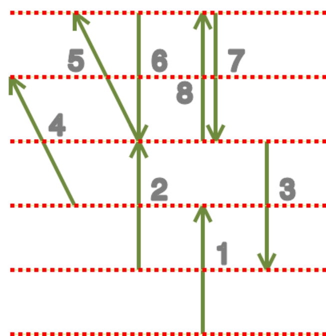

Figura 4.20: Orden de las aristas del polígono.

La Figura 4.20 muestra el orden en el que serían obtenidas las aristas almacenadas en el *heap* de aristas.

Debido a que el algoritmo recorre el polígono de abajo hacia arriba y, dada una *sweep-line*, de izquierda a derecha, entonces, este *heap* es muy útil porque siempre estará en el tope la siguiente arista que debe ser procesada.

Cabe destacar que la inicialización de la cola de eventos es realizada en tiempo $O(n \log n)$ debido a que es insertado una arista tras otra ordenadas dentro del *heap*, en vez de construir el *heap* a partir de todas las aristas ya creadas, que tomaría tiempo $O(n)$.

4.3.3. Implementación de la Lista Activa

La lista activa está implementada dentro de la estructura de datos de las aristas misma, como una lista doblemente enlazada. Una arista está en el *heap* de aristas, luego es eliminada de este *heap* y pasa a estar en la lista activa. El algoritmo tiene un puntero hacia la primera arista de la lista activa y esta arista tiene un puntero hacia la siguiente arista y así sucesivamente.

4.3.4. Implementación de los Eventos

La implementación de los eventos es uno de los puntos más confusos. Los eventos son obtenidos del *heap* de aristas. Debido a cómo son guardadas las aristas en este *heap*, entonces el siguiente evento se encuentra siempre en el tope del *heap*.

Por lo tanto, cada vez que se necesita el siguiente evento, es consultado el tope del *heap* y la nueva *sweep-line* se establece en la coordenada y del vértice principal de esa arista.

Luego, para actualizar la lista activa dada una *sweep-line*, las aristas son sacadas del *heap* de aristas y agregadas a la lista activa. Por lo tanto, en la siguiente iteración, nuevamente, en el tope del *heap* estará la siguiente *sweep-line*.

4.3.5. Aristas Superpuestas y Manejo de Intersecciones

La implementación del algoritmo maneja las aristas duplicadas (pero no aquellas con segmentos de arista superpuestas) y las intersecciones de una forma distinta a cómo se explicó.

Las aristas duplicadas son aristas que tienen ambos vértices en los mismos puntos, es decir, aquellas aristas que son idénticas. Éstas se almacenan en una lista enlazada, definida por punteros, desde la primera arista duplicada hacia la siguiente. Cuando se crean las aristas a partir de los vértices del polígono, la función revisa si es una arista duplicada comparándola contra el tope del *heap* de aristas. Si la arista está duplicada, entonces, se establece que el puntero de la primera arista apunte hacia su duplicado. Si, luego, se encuentra con otra arista duplicada, el puntero de la segunda arista apuntaría a la tercera.

El manejo de intersecciones es el otro punto complicado de la implementación. Cuando se detecta una intersección, las aristas involucradas son divididas en ese punto. La parte inferior es insertada en la lista activa mientras que la parte superior es reinsertada en el *heap* de aristas. Al dividir una arista, las partes se agregan a una lista enlazada definida por punteros desde la parte inferior de la división hacia la parte superior de la división. Si la parte superior es dividida nuevamente, entonces, ésta apuntará a su siguiente división.

De esta forma, se mantiene el invariante de que en el tope del *heap* se encuentra el siguiente evento. Además, realizando este tipo de operaciones, no es necesario modificar el funcionamiento del algoritmo para poder soportar las intersecciones.

Cada vez que se realiza una división de una arista duplicada, la operación es replicada hacia el resto de las aristas duplicadas. Esto es realizado utilizando la lista enlazada de las partes de una arista dividida, es decir, se crean aristas y se copia la información de las nuevas aristas, en vez de realizar la detección de la intersección y los cálculos de nuevo.

4.3.6. Precisión de las Coordenadas y Aproximación a Grilla

Como se dijo anteriormente, las aristas almacenan sus coordenadas en precisión entera y en punto flotante con precisión doble. Esto es, probablemente, por razones de optimización.

Cuando el polígono no contiene aristas con pendientes, el algoritmo utiliza la precisión entera. Pero, si el polígono contiene aristas con pendientes, entonces, utiliza la representación en punto flotante con doble precisión.

Otro punto importante es que todos los eventos (coordenadas y de ciertos vértices e intersecciones) son manejados en precisión entera. Esto quiere decir que cada vez que hay una intersección, la coordenada y es aproximada al entero más cercano mientras que la coordenada x es mantenida en punto flotante.

La coordenada x es aproximada sólo al momento de generar la primitiva, en donde es movida al entero más cercano.

4.3.7. Problemas de la Implementación Actual

La actual implementación del algoritmo *Windfrac* tiene varios problemas. Como ya fue mencionado anteriormente, lo principal, es que es muy difícil de modificar debido a su ilegibilidad. El algoritmo está implementado sobre 2 estructuras de datos (arista y *heap* de aristas) y en una sola función de aproximadamente tres mil líneas.

Además, la implementación misma genera resultados indeseables en ciertos casos, que son los que se verán a continuación.

Debido a que el algoritmo simplemente aproxima hacia el entero más cercano, se pueden construir casos en donde el resultado de la fractura no es simétrica, siendo que la figura original sí lo era. Esto sucede cuando se introduce un corte tal que los nuevos vértices quedan justo entre dos enteros.

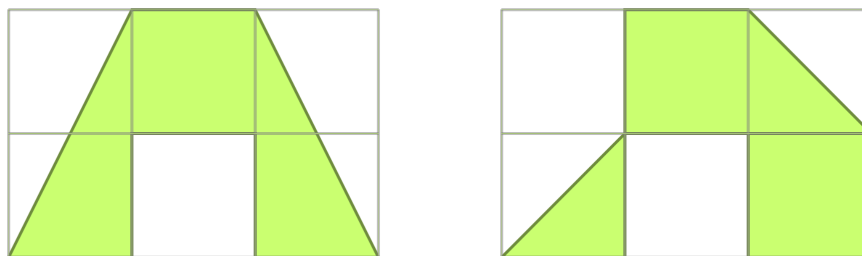

Figura 4.21: Caso en donde el resultado de *Windfrac* no es simétrico.

En la Figura 4.21 se puede ver cómo el polígono original simétrico (a la izquierda) es fracturado por *Windfrac* en un conjunto no simétrico de primitivas (a la derecha).

Otro detalle que tiene *Windfrac* es generar cortes innecesarios cuando el polígono tiene aristas superpuestas. Esto sucede porque, en realidad, el algoritmo no maneja ese caso particular, no toma en cuenta que una arista superpuesta puede no introducir un nuevo corte.

La Figura 4.22 muestra un polígono con aristas superpuestas (a la izquierda) que, en realidad, debería ser un solo trapecio horizontal, pero, la fractura realizada por *Windfrac* genera 3 trapecios horizontales (a la derecha).

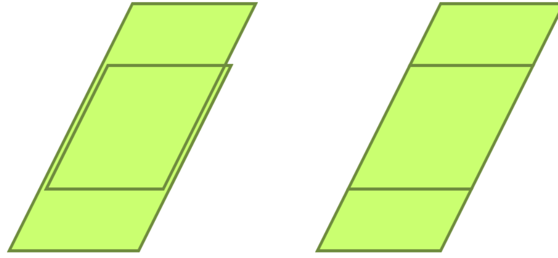

Figura 4.22: Caso en donde el resultado de *Windfrac* introduce cortes innecesarios.

Y, finalmente, debido a como el algoritmo maneja las aristas duplicadas, es posible construir un caso en donde una figura sólida sea fracturada en un conjunto de primitivas con un agujero al medio.

Ésto se puede ver en la Figura 4.23, en donde las dos aristas azules son aristas idénticas (a la izquierda) y la fractura de la derecha es el resultado de la implementación actual.

El problema surge por la forma en que se procesan las aristas, el algoritmo revisa arista por arista sin preocuparse de lo que viene. Por lo tanto, cuando al procesar la *sweep-line* roja, se encuentra con la primera arista duplicada, ésta al ir en dirección opuesta de la arista "izquierda", hace que el *winding-number* sea 0 convirtiéndola en una arista "derecha". Luego, la siguiente arista duplicada se convierte en una arista "izquierda" y la arista "derecha" roja introducirá un corte hasta la arista "izquierda" azul. Este corte introducido produce el triángulo en la fractura de la derecha, generando un agujero. El resultado esperado debería haber sido interpretar las dos aristas duplicadas como aristas "internas" (puesto que tienen información a ambos lados) y el corte debería haber sido hasta la arista de más a la izquierda, como se ve en la fractura de la derecha.

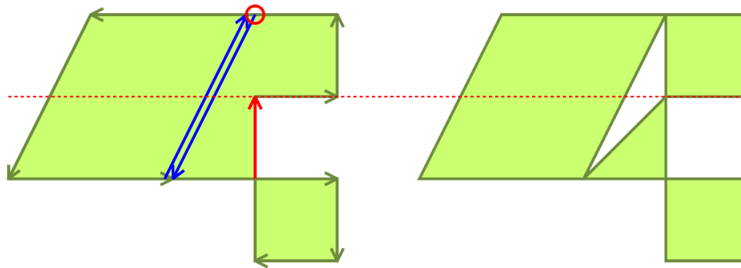

Figura 4.23: Caso en donde el resultado de *Windfrac* genera un agujero.

Todo esto hace que algunos resultados de *Windfrac* sean de baja calidad.

4.4. Detalles de la Reimplementación

La reimplementación del algoritmo *Windfrac* se hizo siguiendo rigurosamente la explicación de la Sección 3.2. De esta forma, se obtuvo una implementación bastante más fácil de leer y de seguir. Adicionalmente, se incluyeron partes para manejar específicamente los problemas que tenía la implementación antigua, como por ejemplo, el manejo de superposición de aristas, encapsulación de las aproximaciones a grilla, entre otros.

El algoritmo fue reimplementado de forma modular, es decir, las distintas partes de éste quedaron separados en módulos bastante independientes entre sí. Cada módulo tiene las estructuras de datos y funciones necesarias para cumplir su objetivo.

Es importante notar que en cada módulo se deben incluir funcionalidades que probablemente cambien, o decisiones difíciles, de tal forma que cambiarlas por código más adecuado sea simple.

Todos los módulos están implementados en archivos separados (encabezado e implementación) con el nombre del módulo como nombre de los archivos, por ejemplo, el módulo `algo-utils` está declarado y definido en los archivos `wf_algo_utils.h` y `wf_algo_utils.c` respectivamente. Todos los nombres de archivo tienen como prefijo `wf_` que es una contracción de *Windfrac*.

A continuación se detalla la información contenida en cada módulo.

4.4.1. Módulo `algo`

Este módulo declara y define la función principal de *Windfrac*, `wf_algo`, e implementa el ciclo principal. La declaración de la función es idéntica a la de la implementación antigua, de esta forma es posible ejecutar cualquiera de los dos algoritmos. El único objetivo de esta función es llamar a los otros módulos para que realicen el trabajo.

La implementación del ciclo principal es bien importante, ya que es el núcleo del algoritmo. Entendiendo este ciclo se entiende inmediatamente como funciona el algoritmo a grandes rasgos.

4.4.2. Módulo `algo-utils`

Este módulo declara e implementa las funciones, que forman parte del algoritmo, utilizadas por el módulo `algo`. Estas funciones son las que actualizan y procesan las estructuras de datos (la lista activa, la cola de eventos y los eventos) para realizar la fractura.

Estas funciones podrían haberse implementado dentro del módulo `algo` puesto que son parte del algoritmo mismo, pero se decidió implementarlas en un módulo por separado para que el usuario no tenga acceso a estas funciones. De esta forma sólo se puede llamar a la función `windfrac`.

Declara y define las siguientes funciones:

`wf_algoEdgesCreate` Genera un arreglo de estructuras `edge_t` a partir del arreglo de vértices del polígono.

`wf_algoEventQueueCreate` Crea la cola de eventos y la inicializa con los eventos asociados a los vértices del polígono.

`wf_algoActiveListCreate` Crea la lista activa y la inicializa con las aristas que tocan la primera *sweep-line*.

wf_algoActiveListUpdate Actualiza la lista activa dada la cola de eventos y la *sweep-line* actual.

wf_algoPrimitivesGenerate Genera las primitivas dada la lista activa y la *sweep-line* actual.

wf_algoActiveListClean Vuelve a actualizar la lista activa removiendo las aristas procesadas e intercambiando aristas que se intersectan.

4.4.3. Módulo edge

Este módulo contiene la estructura que almacena la información de una arista, `edge_t`, junto con los tipos de intersecciones que pueden tener las aristas y todas las funciones necesarias para modificar y actualizar esta información y operaciones entre aristas.

La importancia de este módulo es abstraer las funciones que manejan y detectan intersecciones entre las aristas. De forma que operar aristas, desde fuera de este módulo, sea muy simple y transparente. Como también, cambiar las funciones que intersectan aristas o agregar nuevas formas de intersectar es muy simple, basta con agregar o modificar las funciones asociadas.

La estructura `edge_t` contiene la siguiente información:

bot La coordenada de más abajo de la arista, independiente del sentido de ésta.

top La coordenada de más arriba de la arista.

clip El último valor en el eje y donde fue recortada la arista.

windingList Una lista con los `winding-numbers`. Esto será explicado en el módulo `winding-list`.

Almacenando el valor de `clip` se obtiene una mejor calidad en los resultados de la fractura porque así es más difícil que se propaguen errores de aproximación modificando la pendiente de la arista.

Las funciones implementadas en este módulo son las siguientes:

wf_edgeNew Pide memoria para una arista y la inicializa con valores por omisión.

wf_edgeDelete Libera la memoria utilizada por una arista.

wf_edgeInit Inicializa una arista con valores dados.

wf_edgeNewArray Pide memoria para un arreglo de aristas.

wf_edgeDeleteArray Libera la memoria utilizada por un arreglo de aristas.

wf_edgeWinding Retorna el *winding-number* de una arista dada la *sweep-line*.

`wf_edgeMerge` Mezcla dos aristas superpuestas, generando una con *winding-numbers* combinados.

`wf_edgeIntersectWithEdge` Intersecta dos aristas devolviendo el tipo de intersección y la coordenada donde ocurrió.

`wf_edgeCrossProduct` Devuelve el producto cruz entre dos aristas.

`wf_edgeIntersectWithSweepLine` Intersecta una arista con una *sweep-line* devolviendo la coordenada x donde ocurrió.

`wf_edgeCompare` Compara dos aristas. Esta comparación se realiza basado en una serie de reglas que toman en cuenta la información de ambas aristas.

4.4.4. Módulo `active-list`

Se decidió crear este módulo porque la lista activa puede ser implementada con distintas estructuras de datos. Por lo tanto, si en el futuro se desea utilizar otra estructura de dato para almacenar la lista de aristas activas, basta con reemplazar este módulo con el nuevo.

Este módulo define la estructura de datos para manejar la lista de aristas activa como una lista doblemente enlazada. La razón de esto es para simplificar la implementación de las funciones que operan sobre esta lista activa.

Las funciones asociadas a la lista activa son las siguientes:

`wf_activeListNew` Pide memoria para una nueva lista activa.

`wf_activeListDelete` Libera la memoria utilizada por una lista activa.

`wf_activeListFind` Busca una arista dentro de la lista activa.

`wf_activeListLast` Devuelve la última arista de la lista activa.

`wf_activeListInsertAfter` Inserta una arista antes de otra dada.

`wf_activeListInsertBefore` Inserta una arista después de otra dada.

`wf_activeListRemove` Remueve una arista de la lista activa.

`wf_activeListSwap` Invierte las posiciones de dos aristas vecinas dentro de la lista activa.

4.4.5. Módulo `winding`

Este módulo es muy simple, sólo tiene dos funciones que son utilizadas para saber si un *winding-number* representa data o no basado en el tipo de regla utilizado.

Estas funciones fueron encapsuladas por si en algún momento cambia la forma de determinar si un *winding-number* representa información o si representa vacío.

4.4.6. Módulo `event-queue`

Al igual que el módulo `active-list`, la cola de eventos puede ser implementada con distintas estructuras de datos. Por lo tanto, se decidió abstraer esta sección del código en un módulo para que cambiar la implementación de la cola de eventos por otra, si es que fuera necesario, sea simple.

Este módulo declara y define las funciones para utilizar la cola de eventos, junto con la estructura de datos donde se almacenan. Los eventos son almacenados en un árbol binario, para así poder encontrar el menor (aquel de más a la izquierda) y poder insertar nuevos eventos evitando los repetidos.

Las funciones implementadas son las siguientes:

`wf_eventQueueNew` Pide memoria para una nueva cola de eventos.

`wf_eventQueueDelete` Libera la memoria utilizada por la cola de eventos.

`wf_eventQueuePop` Remueve y retorna el siguiente evento.

`wf_eventQueuePopIf` Remueve y retorna el siguiente evento si y sólo si éste se encuentra en la *sweep-line* dada.

`wf_eventQueueTop` Retorna el siguiente evento sin removerlo de la cola de eventos.

`wf_eventQueueInsert` Inserta de forma ordenada el siguiente evento.

`wf_eventQueueUpdate` Actualiza la cola de eventos insertando posibles intersecciones entre dos aristas dadas.

4.4.7. Módulo `event`

Este módulo define la estructura `event_t` que almacena la información asociada a un evento. Esta información es la siguiente:

`point` La coordenada donde está ubicado el evento.

`type` El tipo de evento, puede ser un vértice de abajo de una arista, un vértice de arriba, vértices superpuestos o intersecciones.

`edge` La arista involucrada en el evento.

`other` Si el evento es de tipo superposición o intersección, entonces acá se almacena la otra arista involucrada.

Además, implementa las siguientes funciones:

`wf_eventNew` Pide memoria para un evento nuevo y lo inicializa con valores por omisión.

`wf_eventDelete` Libera la memoria utilizada por un evento.

`wf_eventInit` Inicializa un evento con valores dados.

`wf_eventCompare` Compara dos eventos, primero compara las aristas involucradas y luego el tipo de evento.

4.4.8. Módulo `winding-list`

El *winding-list* es una lista de *winding-numbers* asociados a rangos disjuntos dentro de una arista. De esta forma es posible que una arista tenga más de un *winding-number* y así se pueden mezclar aristas superpuestas generando una arista pero con los *winding-numbers* de ambas aristas. Es importante notar que a una altura dada, el *winding-list* puede tener sólo un *winding-number*.

Este módulo implementa el *winding-list* como una lista doblemente enlazada. Cada nodo de la lista tiene el *winding-number* y el rango del eje y en donde se aplica.

La forma de manejar la lista de *winding-number*s puede cambiar en cualquier momento, y, por esa razón, se decidió implementarla en un módulo por separado.

Las funciones implementadas para manejar el *winding-list* son las siguientes:

`wf_windingListNew` Pide memoria para una nueva lista de *winding-numbers*.

`wf_windingListDelete` Libera la memoria utilizada por el *winding-list*.

`wf_windingListMerge` Mezcla dos *winding-lists* en una, uniendo o separando rangos donde sea necesario.

`wf_windingListNumber` Retorna el *winding-number* dada una *sweep-line*.

4.4.9. Módulo `output`

Este módulo es el que conecta *Windfrac* con el resto de CATS. Es el encargado de pasarle las primitivas generadas al resto de la aplicación.

Estas funciones se encuentran en un módulo por separado porque no pertenecen al algoritmo mismo, por lo que podría ser cambiado en cualquier momento.

Implementa las siguientes funciones:

`wf_outputMakePrimitive` Intenta generar una primitiva dado un polígono cualquiera.

`wf_output` Genera una primitiva dados los cuatro puntos de ésta. Esta función revisa si la primitiva es un rectángulo, trapecio horizontal o trapecio vertical y le pasa la figura adecuada al resto de CATS.

4.4.10. Módulo `grid`

Este módulo es muy simple, sólo contiene las funcions para manejar comparaciones y aproximaciones en grilla. De esta forma, cualquier cambio en cómo manejar el tema de la grilla puede ser fácilmente modificable.

Debido a que la forma de aproximar los puntos a una grilla puede cambiar a lo largo del tiempo, o porque se puede decidir incluir formas mejores para tomar la decisión de cómo mover a grilla un punto, se decidió abstraer estas funcionalidades en un módulo.

Las funciones implementadas son las siguientes:

`wf_gridCompare` Compara dos números tomando en cuenta la grilla, es decir, dos números son iguales si son "parecidos".

`wf_gridApproximate` Aproxima un número a la grilla dado el centroide del polígono y el eje en el cual se desea aproximar.

Capítulo 5

Resultados

En este Capítulo se revisan los resultados obtenidos del trabajo realizado. Contiene las siguientes Secciones:

Sección 5.1, Legibilidad Se explica cómo fue mejorada la legibilidad de la implementación.

Sección 5.2, Calidad de la Fractura Muestra cómo la nueva implementación de *Windfrac* mejora la calidad de las fracturas.

Sección 5.3, Casos de Prueba Se muestra un resumen de los resultados obtenidos tras ejecutar las suites de los casos de prueba.

Sección 5.4, Desempeño Presenta las mediciones del tiempo que toma la nueva implementación del algoritmo en comparación con la antigua.

5.1. Legibilidad

La legibilidad de la implementación aumentó debido a las siguientes razones:

- Al ser implementado de forma modular, el problema fue dividido en partes más pequeñas, más fáciles de entender y más fáciles de abordar.
- Al ser separada la implementación en archivos hace que cada uno de éstos sea mucho más manejable. Y, al ser independientes entre sí, permite ver y estudiar uno a la vez.
- Las funciones tienen nombres más adecuados y se utilizan variables más descriptivas.
- Aproximadamente el 22 % de las líneas son comentarios que explican partes del código.
- La cantidad de líneas se redujo de $3,000$ a $1,800$ líneas aproximadamente.

Un incremento en la legibilidad del código tiene un gran impacto, ya que ésto permite que pueda ser modificado, ya sea para aumentar el desempeño, para modificar su comportamiento, o para actualizar ciertas partes, entre otros. Esto es un punto muy importante debido a que un código ilegible se traduce en código muerto que nadie puede modificar sin romperlo.

La implementación antigua de *Windfrac* se mantuvo más de 20 años sin cambios mayores.

5.2. Calidad de la Fractura

Se implementaron cuatro medidas para mejorar la calidad de la fractura. Estas son explicadas a continuación.

5.2.1. Coordenadas

Todas las coordenadas son manejadas en punto flotante con precisión doble. Esto quiere decir que el único momento en el cual son aproximadas las coordenadas es cuando se va a generar una primitiva para ser entregada al resto de CATS.

Además, al momento de recortar una arista, en vez de modificar el valor `bot` de ésta, el valor `clip` es actualizado. De esta forma, los valores `bot` y `top` se mantienen siempre iguales lo que implica que las pendientes de las aristas también se mantendrán siempre iguales.

Junto con esto, se implementó el algoritmo con una *sweep-line* en punto flotante con precisión doble, para evitar aproximaciones tempranas. La implementación antigua usaba una *sweep-line* en enteros.

Con estas medidas se evita la propagación de errores debido a consecutivas aproximaciones incorrectas.

5.2.2. Manejo de Intersecciones

La nueva implementación no separa las aristas cuando detecta una intersección, como lo hacía la implementación antigua. Al separar las aristas aumentaba la posibilidad de que existieran errores de aproximación, ya que las pendientes de éstas podrían verse afectadas al tener que aproximar el punto de intersección.

5.2.3. Aproximación a Grilla

Como se mencionó anteriormente, la aproximación a grilla, ahora, es realizada sólo cuando se va a generar la primitiva para ser entregada al resto de CATS. En todo el proceso de ejecución del algoritmo *Windfrac* ninguna coordenada es llevada a grilla, trabajando siempre sobre la figura a la máxima resolución posible.

Adicionalmente, la aproximación a grilla es realizada de tal forma que los vértices sean movidos de una forma más consistente. En la mayoría de los casos la coordenada es simplemente llevada al entero más cercano, pero si una coordenada se encuentra equidistante entre dos enteros, entonces, es llevado al entero tal que se aleje del centroide del polígono. De esta forma se pueden formar figuras más simétricas en la mayoría de los casos.

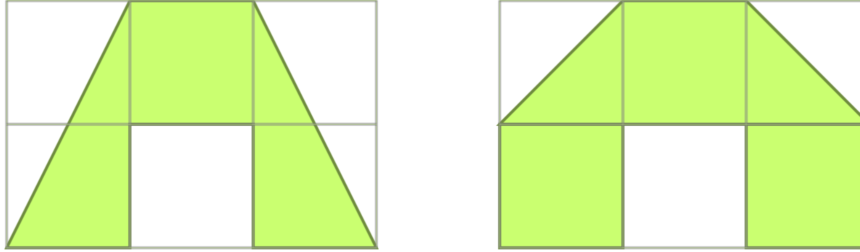

Figura 5.1: Caso en donde el resultado de *Windfrac* es simétrico.

La Figura 5.1 muestra cómo se genera un conjunto de primitivas que forman una figura simétrica al fracturar el polígono de la izquierda.

5.2.4. Aristas Superpuestas

Otra de las mejoras incorporadas fue el manejo explícito de aristas superpuestas mediante la lista de *winding-numbers*. Con esta medida fue posible evitar cortes innecesarios debido a la imposibilidad de notar cuando una arista es la continuación de otra. La Figura 5.2 muestra cómo el polígono de la izquierda es fracturado en un solo trapecio horizontal a la derecha.

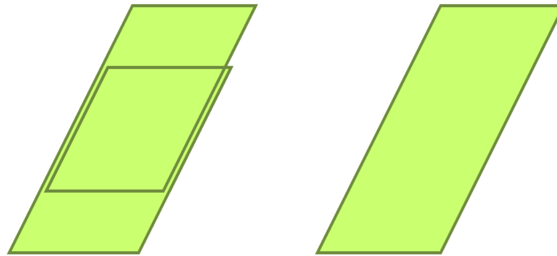

Figura 5.2: Caso en donde el resultado de *Windfrac* no introduce cortes innecesarios.

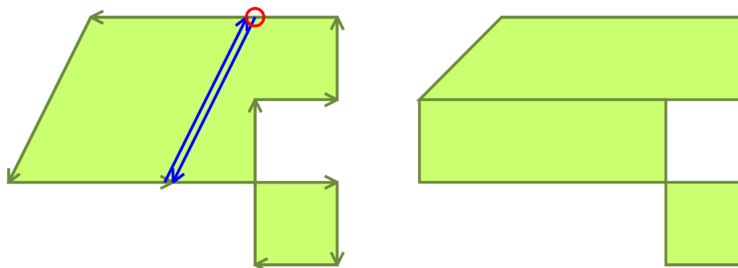

Figura 5.3: Caso en donde el resultado de *Windfrac* no genera un agujero.

Lo más importante de esta mejora, es que arregla el error del agujero inesperado en cierto tipo de polígonos. Al mezclar las aristas que están superpuestas, el algoritmo es capaz de procesarlas al mismo tiempo, en vez de una a la vez. En la Figura 5.3 se puede ver cómo el polígono de la izquierda genera un conjunto de polígonos que forman una figura sólida.

5.3. Casos de Prueba

Un caso de prueba consiste en un diseño de un circuito junto con un conjunto de operaciones a realizar, y un resultado esperado. El resultado esperado se genera a partir de los algoritmos ya funcionando, es decir, son pruebas de regresión. Estos casos de prueba son utilizados para asegurarse de que el código nuevo no rompa el código existente.

Al ejecutar un caso de prueba, el diseño junto con el conjunto de operaciones es entregada a la aplicación con código nuevo, y la salida es comparada con el resultado esperado. Esta comparación es realizada ejecutando un `Xor` entre los diseños, el cual será vacío si es que ambos diseños son idénticos. De esta forma, al ejecutar los casos de prueba, éstos pueden pasar, tener diferencias en el `Xor` o, simplemente, caerse.

Se corrieron varias suites de casos de prueba para verificar la nueva implementación del algoritmo *Windfrac*. En total sumaron aproximadamente mil casos de prueba distintos.

Pass	788 (80 %)
Xor	181 (18.4 %)
Assert	15 (1.6 %)
Total	984 (100 %)

Cuadro 5.1: Porcentajes de pruebas que pasaron y que fallaron.

El Cuadro 5.1 muestra el resumen del resultado luego de ejecutar los casos de prueba. De los 984 casos, 788 pasaron normalmente, 181 tuvieron diferencias entre lo esperado y lo obtenido y 15 fallaron debido a asserts en el código.

Es normal que algunas pruebas hayan fallado debido a diferencias entre lo esperado y lo obtenido, esto es debido a que la nueva implementación de *Windfrac* efectivamente devuelve resultados distintos en ciertos casos. Esto fue explicado anteriormente en la Sección 5.3. Lo que se debe hacer en esta situación es revisar caso a caso para ver si el resultado obtenido efectivamente es mejor que el esperado.

Se revisó aproximadamente el 25 % de los casos de prueba que fallaban por diferencias de *Xor*. En todos estos casos, las diferencias eran debido a la nueva aproximación a grilla, que toma en cuenta el centroide de la figura, por lo que genera primitivas más anchas en algunos casos.

No se encontraron casos interesantes de mejora de calidad, como casos con aristas superpuestas.

Con respecto a los 15 casos que se caen por asserts, deben ser corregidos. Estos son errores que quedaron en la reimplementación.

5.4. Desempeño

Si bien hubo bastantes mejoras en términos de calidad, también es importante analizar qué sucede en términos del desempeño, es decir, del tiempo que toma el algoritmo en ser ejecutado. Este punto es importante ya que un algoritmo lento puede hacer que todo el proceso de transcripción de datos tome aún más tiempo.

Para medir el desempeño, se ejecutaron ambos algoritmos sobre varios diseños, anotando el tiempo que cada algoritmo tomó al fracturar los distintos polígonos, junto con la cantidad de vértices de éstos. Luego, se promediaron los tiempos dada una cantidad de vértices y se confeccionó el gráfico que se ve en la Figura 5.4.

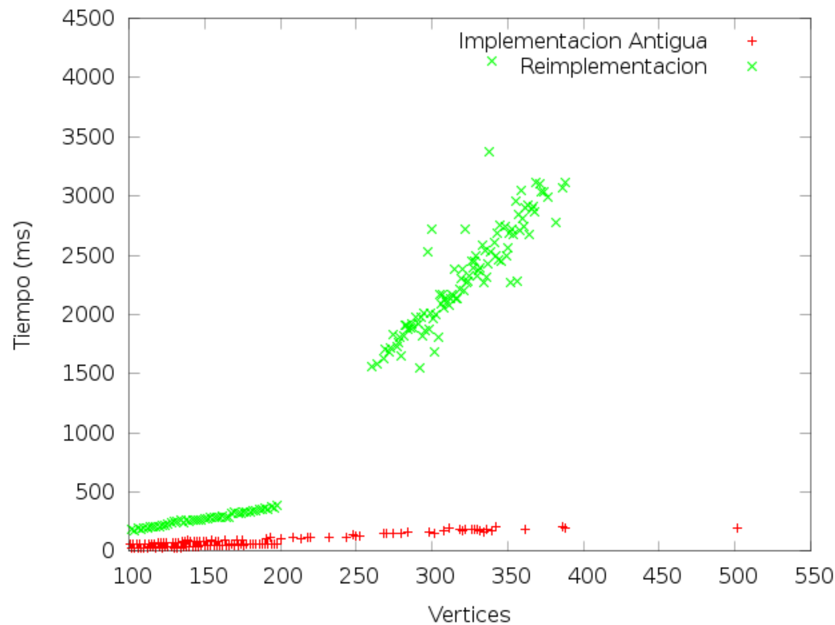

Figura 5.4: Mediciones de tiempo de las distintas implementaciones de *Windfrac*.

El gráfico de la Figura 5.4 muestra el desempeño de ambos algoritmos. El eje x marca la cantidad de vértices de los polígonos mientras que el eje y marca el tiempo en milisegundos que tomó en fracturar el polígono dado. Se puede ver claramente que la implementación nueva toma consistentemente más tiempo que la antigua.

No es necesario preocuparse mucho por estos resultados preliminares, puesto que la nueva implementación aún puede ser optimizada. Hay funciones que pueden ser declaradas *inline*, estructuras de datos que pueden ser optimizadas (como implementar árboles AVL para la cola de eventos), hay código que puede ser mejorado y hay técnicas que pueden ser agregadas (como la implementación de un *caché*).

Para realizar estas optimizaciones, es muy importante primero buscar los cuellos de botella y optimizar esos puntos. La gran ventaja de que la nueva implementación sea más legible es que estas optimizaciones pueden ser realizadas sin mucho problema.

Capítulo 6

Discusión y Conclusiones

En este Capítulo se discuten los resultados obtenidos del trabajo y se concluye esta memoria. Contiene las siguientes Secciones:

Sección 6.1, Cumplimiento de Objetivos Se analiza el nivel de cumplimiento de los objetivos planteados.

Sección 6.2, Puntos a Mejorar Se mencionan los puntos de la reimplementación que, a pesar de ser funcionales, aún pueden ser mejoradas.

Sección 6.3, Trabajo Futuro Se discuten ideas que complementarían el estudio del algoritmo *Windfrac*.

Sección 6.4, Conclusiones Se realizan las conclusiones finales de esta memoria.

6.1. Complimiento de Objetivos

A continuación se mencionan los objetivos del Capítulo 1, siendo complementados con los resultados obtenidos durante el desarrollo de esta memoria.

- *Estudiar, analizar y documentar la entrada del algoritmo, la implementación actual y posibles soluciones al problema de la fractura.* Cumplido. Este es el punto más importante de la memoria, entender el funcionamiento completo del algoritmo, y fue comprendido en su totalidad. Tanto la entrada del algoritmo, como el funcionamiento de éste y las posibles soluciones a los problemas actuales fueron debidamente estudiados y documentados.
- *Mejorar el algoritmo para solucionar el problema de la fractura horizontal.* Cumplido. Luego de haber estudiado las posibles soluciones, se decidió reimplementar el algoritmo. Éste fue ligeramente modificado para ser más entendible y luego fue reimplementado siguiendo al pie de la letra la descripción del nuevo algoritmo.
- *Restricciones*

  - *Eficiencia competitiva.* Cumplido. Debido a que la solución fue reimplementar el mismo algoritmo, el costo teórico se mantuvo. Cabe destacar que, en la práctica, el nuevo algoritmo toma más tiempo que el actual.
  - *Resolver los problemas de la calidad de los resultados.* Cumplido. Al haber implementado las mejoras explicadas anteriormente, fue posible mejorar la calidad de la fractura generada por *Windfrac*.
  - *Uniformidad de los resultados del proceso.* Cumplido. Debido a que todas las decisiones tomadas dentro del algoritmo son determinísticas, y que las aproximaciones dependen de la figura independiente de su posición, se obtuvo uniformidad en los resultados.
  - *Consistencia en la toma de decisiones.* Cumplido. Al realizar las aproximaciones se toma en cuenta el centroide del polígono. Incluso, es posible modificar (sin mayores dificultades) para que tome en cuenta el polígono entero, o partes de éste.
  - *Encapsulación de las aproximaciones.* Cumplido. Las aproximaciones quedaron encapsuladas en su propio módulo.
  - *Independiente de la precisión de la grilla.* Cumplido. Esto se realizó manteniendo el supuesto de que los enteros representan la grilla.
  - *Independiente de la representación de las coordenadas.* Cumplido. Debido a la modularidad de la aplicación, es posible modificar el tipo de variable para representar las coordenadas con cambios menores.

- *Especificación de la entrada del nuevo algoritmo.* Cumplido. La entrada se mantuvo igual, de esta forma es posible ejecutar cualquiera de los dos algoritmos.
- *Pasar los casos de prueba existentes.* No cumplido. Si bien la gran mayoría de los casos de prueba pasan (más del 80 %), hay un bajo porcentaje (aproximadamente el 18 %) que presenta diferencias y que deben ser revisadas (la mayoría deben representar diferencias debido a la mejora de calidad) y hay un bajísimo porcentaje (aproximadamente el 2 %) que no pasan las pruebas y que definitivamente deben ser revisados.

Entonces, se puede concluir que, a un nivel general, se cumplieron los objetivos planteados para esta memoria; el algoritmo *Windfrac* fue entendido y la implementación fue mejorada.

6.2. Puntos a Mejorar

Si bien los objetivos fueron cumplidos, esto no quiere decir que la reimplementación es óptima. A continuación se indican algunos puntos que pueden ser mejorados:

- La mejora más evidente es que se puede disminuir el porcentaje de pruebas que no son pasadas corrigiendo la reimplementación.
- Se propone implementar árboles AVL para la cola de eventos. Esto mejoraría el costo del peor caso al momento de insertar nuevos eventos en esta cola.

- Se propone implementar reglas más completas para aproximar puntos a grilla. Aún es posible construir casos en donde las aproximaciones no generarían figuras simétricas, por lo que se sugiere considerar regiones del contorno del polígono. Habría que tener cuidado de no aumentar el costo para evitar disminuir el desempeño del algoritmo.

- Se propone implementar un *caché* de polígonos. Se puede tener un *caché* de polígonos en donde se almacena un *hash* de éste junto con el resultado de la fractura, para que así en vez de recalcular la fractura de un polígono ya procesado, simplemente se devuelve el resultado. Esto debería aumentar enormemente el desempeño del algoritmo puesto que los diseños de circuitos normalmente tienen muchas figuras idénticas ubicadas en posiciones distintas.

6.3. Trabajo Futuro

El entendimiento del funcionamiento del algoritmo *Windfrac* abre las puertas a mejoras en el desempeño de CATS en general. A continuación se presentan posibles trabajos a realizar en el futuro:

- Como fue mencionado en el Capítulo 1, después de que se ejecuta *Windfrac* es ejecutado *Healing*. Es posible mezclar ambos algoritmos, basta con que *Windfrac* realice lo mismo solo que con todos los polígonos a la vez. El único punto importante que hay que tener en mente es que se necesita que cada polígono tenga su propio par de *winding-numbers* al momento de ser procesado, de esta forma una región tiene data si en alguno de los polígonos de esa región el *winding-number* dice que hay data. Puesto que tanto *Windfrac* como *Healing* realizan un proceso de tipo *sweep-line* sobre todos los polígonos (*Windfrac* lo realiza uno a uno mientras que *Healing* lo realiza todos a la vez), mezclando ambos algoritmos en uno se debería poder reducir este tiempo a, aproximadamente, la mitad.

- También como fue mencionado en el Capítulo 1, hay ciertos algoritmos que se ejecutan después de *Healing* que repoligonizan la entrada y son ejecutados sobre polígonos con ciertas propiedades definidas. Es posible implementar esta repoligonización al momento de ir generando las primitivas, de tal forma que se realice junto con *Windfrac*. De esta forma, si ninguna de los algoritmos escogidos para una ejecución de CATS en particular necesita de primitivas, *Windfrac* podría devolver polígonos y, así acelerar el tiempo de ejecución completo.

6.4. Conclusiones

Para lograr el objetivo principal de esta memoria fue necesario, primero que todo, entender a cabalidad los conceptos Geométricos y de Computación Geométrica utilizados en el algoritmo *Windfrac*. Así, de esta forma, se logró deducir el funcionamiento de éste, para luego mejorar su implementación.

En relación a lo anterior, se puede decir que esta memoria ha sido terminada de forma exitosa. Los objetivos planteados fueron cumplidos; el algoritmo fue descifrado y su implementación fue mejorada.

Pero, el trabajo no termina acá puesto que aún quedan optimizaciones que pueden ser realizadas, tanto a corto como a largo plazo. El entendimiento de *Windfrac*, que es uno de los pasos fundamentales dentro del *pipeline* de CATS, abrió las puertas a la investigación de nuevas formas de acelerar el proceso completo.

Finalmente, el verdadero éxito de esta memoria se verá en el tiempo, cuando logre reemplazar a la antigua implementación, y se dedique tiempo y esfuerzo a desarrollar las nuevas posibilidades descubiertas.

Referencias

[1] Behnke, H., Bachmann, F., Fladt, K., Suss, W., and Kunle, H. *Fundamentals of Mathematics, Volume II: Geometry*, third ed. The MIT Press, 1986.

[2] Bentley, J., and Ottmann, T. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers* (1979), 643–647.

[3] Berg, G., Julian, W., Mines, R., and Richman, F. The constructive jordan curve theorem. *Journal of Mathematics Volumen 5, N 2* (1975), 225–236.

[4] de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. *Computational Geometry: Algorithms and Applications*, third ed. Springer, 2008.

[5] Euclid. *Euclid's Elements.* Green Lion Press, 2002.

[6] Lavagno, L., Martin, G., and Scheffer, L. *Electronic Design Automation for Integrated Circuits Handbook.* CRC, 2006.

[7] Morales, D., and Leon, G. Reuniones, 2009.

[8] Newell, M., and Sequin, C. The inside story on self-intersecting polygons. *Lambda 2do Trimestre* (1980), 20–24.

[9] O'Rourke, J. *Computational Geometry in C*, second ed. Cambridge University Press, 1998.

[10] Synopsys. About CATS. `http://www.synopsys.com/Tools/Manufacturing/MaskSynthesis/CATS/Pages/default.aspx`.

[11] Synopsys. About Synopsys. `http://www.synopsys.com/Company/AboutSynopsys/Pages/default.aspx`.

[12] Synopsys. Implementación algoritmo *Windfrac*, 1985. Aproximadamente 3000 líneas de código.

[13] Weisstein, E. Contour winding number. `http://mathworld.wolfram.com/ContourWindingNumber.html`.