UNIVERSIDAD DE ALCALÁ

Escuela Politécnica

INGENIERÍA INFORMÁTICA

Trabajo Fin de Carrera

Estudio y visualización de envolventes convexas y triangulaciones con direcciones restringidas en Geometría Computacional

J. Daniel Expósito Polo

Junio 2009

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica

INGENIERÍA INFORMÁTICA

Trabajo Fin de Carrera

Estudio y visualización de envolventes convexas y triangulaciones con direcciones restringidas en Geometría Computacional

Autor: J. Daniel Expósito Polo

Director: David Orden Martín

TRIBUNAL:

Presidente:

Vocal 1º:

Vocal 2º:

CALIFICACIÓN:.....

FECHA:.....

DEDICATORIA

A mis padres por darme la gran oportunidad de estudiar una carrera.

A mi familia en general por apoyarme siempre en todo lo que hago.

A mi tutor de proyecto por otorgarme el honor de trabajar con él.

Y a todos mis amigos y compañeros por acompañarme en el trascurso de mi vida universitaria.

ÍNDICE

ÍNDICE GENERAL

ÍNDICE DE IMÁGENES

ÍNDICE DE DIAGRAMAS

ÍNDICE DE DEFINICIONES

ÍNDICE DE TABLAS

1. Resumen

La Geometría Computacional abarca desde problemas de visualización hasta problemas relativos al diseño de mundos virtuales. Dos de los aspectos más importantes que se estudian en el campo de la Geometría Computacional son: la envolvente convexa y las triangulaciones.

En el caso habitual se considera todo el conjunto de direcciones del plano, que se puede representar por [0,180) en función del ángulo que forman con la horizontal. Pero existen numerosas aplicaciones en las que el conjunto de direcciones se encuentra restringido. Esto ha fomentado el estudio para el caso de direcciones restringidas de problemas clásicos en Geometría Computacional.

Este proyecto trata de indagar en dichos estudios para obtener la siguiente definición de envolvente convexa con direcciones restringidas: *Dado un conjunto de puntos, su cierre O-convexo es la intersección de todos los semiplanos escalera cerrados que lo contienen.*

De esta definición se puede extraer la de triangulación con direcciones restringidas. Un triángulo se puede definir como la envolvente convexa de un conjunto de tres puntos, como consecuencia, se puede definir un O-triángulo (triángulo con direcciones restringidas) como la envolvente O-convexa de tres puntos. Una O-triangulación, por lo tanto, se puede definir como un conjunto de O-triángulos que cumple ciertas propiedades.

Asimismo en el presente proyecto se dan dos algoritmos: uno para el cálculo de envolventes convexas y triangulaciones con direcciones restringidas mediante la intersección de semiplanos escalera y otro para el cálculo de envolventes convexas (y triangulaciones) con direcciones restringidas mediante el inflado de las aristas.

Dichos algoritmos han sido implementados en un software que permite la visualización de triangulaciones y envolventes convexas con direcciones restringidas. De esta manera se hace más intuitiva la idea de direcciones restringidas y se dispone de una herramienta para su cálculo. Este software ofrece numerosas posibilidades como son: visualización paso a paso, una interfaz intuitiva mediante la cual introducir los datos de entrada, posibilidad de guardar y cargar distribuciones de puntos, posibilidad de configurar la interfaz y posibilidad de llevar a cabo una triangulación mediante el novedoso método del inflado, que supone la forma más visual para comprender el concepto de direcciones restringidas.

2. Introducción

Actualmente uno de los campos más importantes de la informática es la Geometría Computacional. La Geometría Computacional abarca desde problemas de visualización (utilizados en robótica) hasta problemas relativos al diseño de mundos virtuales (generación de mallas tridimensionales, asignar texturas a mallas tridimensionales…) Dos de los aspectos que se estudian en el campo de la Geometría Computacional son: la envolvente convexa y las triangulaciones.

La envolvente convexa y las triangulaciones se encuentran entre las estructuras más importantes en Geometría Computacional, tanto para conjuntos de puntos en dos dimensiones como en tres dimensiones (Seidel y Bern, 2004). Las aplicaciones de la envolvente convexa abarcan desde el reconocimiento de patrones hasta el procesamiento de imágenes (Berg et al., 2000); mientras que las triangulaciones se utilizan en aplicaciones tan vanguardistas como el modelado de personajes y escenarios tridimensionales de los videojuegos (Shreiner et al., 2007).

*La **envolvente convexa** de un conjunto de puntos P es la intersección de todos los semiplanos que contienen a P*. La envolvente convexa forma parte de todas las triangulaciones de un conjunto. A los vértices de la envolvente convexa se les llama puntos *extremos* o *frontera* del conjunto. A los puntos restantes se les llama *interiores* (Preparata y Hong, 1977). Por otra parte la triangulación es un método de obtener áreas de figuras poligonales, normalmente irregulares, (dos dimensiones) o de estimar el área de superficies (tres dimensiones) mediante su descomposición en formas triangulares.

En el caso habitual se considera todo el conjunto de direcciones del plano, que se puede representar por [0,180) en función del ángulo que forman con la horizontal. Pero existen numerosas aplicaciones, como diseño VLSI, CAD o informática gráfica, en las que el conjunto de direcciones se encuentra restringido (Rawlins y Wood, 1991). Como ejemplo, el diseño VLSI comenzó con direcciones horizontal y vertical (ángulos 0º y 90º) y se ha ido desarrollando hasta aceptar cualquier conjunto finito de direcciones.

Esto ha fomentado el estudio para el caso de direcciones restringidas de problemas clásicos en Geometría Computacional (Nilsson et al., 1992) como el cálculo de envolventes convexas (Martynchik et al., 1996), el estudio de la convexidad (Fink y Wood, 2004) o el de la visibilidad (Fink y Wood, 2003).

Sin embargo, y pese a ser una estructura fundamental como ya se ha mencionado, el estudio de triangulaciones con orientaciones restringidas parece no haber atraído el interés de la comunidad investigadora

Su relación con el cálculo de envolventes convexas (de tripletas de puntos) y su importancia justifican el interés del estudio de ambos problemas de una manera coordinada.

El presente trabajo realiza un estudio de envolventes convexas y triangulaciones tanto para el caso general como para el caso de direcciones restringidas (dicho estudio será llevado a cabo para dos dimensiones salvo que se diga lo contrario), estableciendo las diferencias entre ambos casos. En este proceso se ha hallado una definición de envolvente convexa con direcciones restringidas adecuada. Tras ello se ha procedido a la investigación y diseño de los algoritmos necesarios para el cálculo de la envolvente convexa con direcciones restringidas y de triangulaciones con direcciones restringidas. Como última consecuencia de este trabajo se ha creado una aplicación que demuestra todos los conocimientos adquiridos y que aplica los algoritmos investigados y/o desarrollados.

2.1. Objetivos

El propósito de este proyecto es realizar un estudio en el campo de la Geometría Computacional para comparar las características de la envolvente convexa y de las triangulaciones en los casos en que se tienen todas las direcciones y cuando las mismas están restringidas. En base a esta comparativa, se buscará desarrollar los mecanismos necesarios para visualizarlas.

Los objetivos específicos planteados son los siguientes:

1. Revisar las nociones de envolvente convexa y triangulación sin restringir dirección alguna.

2. Estudiar las diferentes definiciones de envolvente convexa con direcciones restringidas.

3. Obtener conclusiones del caso particular de las triangulaciones con orientaciones restringidas.

4. Estudiar los algoritmos para la representación de envolventes convexas y triangulaciones con direcciones restringidas.

    a. Buscar una solución para hallar el punto maximal en una dirección cualquiera.

    b. Buscar una solución para unir dos puntos mediante direcciones restringidas.

    c. Buscar una solución para calcular la envolvente convexa mediante un método similar a la intersección de semiplanos (mencionado anteriormente).

5. Desarrollar un programa de visualización de envolventes convexas y triangulaciones con direcciones restringidas.

    a. Implementar los algoritmos que solventan los problemas expuestos en los apartados que van desde el 4.a hasta el 4.c.

6. Redactar una memoria o informe final.

3. Introducción a las envolventes convexas y las triangulaciones

3.1. Introducción a las envolventes convexas

En esta sección se llevará a cabo una revisión del concepto de envolvente convexa. Asimismo se dará una breve perspectiva histórica de la envolvente convexa y finalmente se mostrarán algunos de los algoritmos más destacados para el cálculo de envolventes convexas y varios ejemplos de aplicación de las mismas.

3.1.1.Definición

Antes de pasar a definir la envolvente convexa, hay que definir el significado de convexo. *Un conjunto de puntos en el plano es **convexo** si su intersección con cualquier recta del plano es conexa*. Así, el propio plano es un conjunto convexo. También lo son los semiplanos determinados por una recta, tanto si incluyen el borde como si no (semiplanos cerrados o abiertos). Alternativamente, un conjunto de puntos del plano es convexo si el segmento que une dos cualesquiera de sus puntos está totalmente contenido en el conjunto. La intersección de conjuntos convexos es un conjunto convexo, ya que si dos puntos están en la intersección, están en cada uno de los conjuntos convexos que la forman, por lo que el segmento que los une está totalmente contenido en todos ellos y, por tanto, en la intersección (Abellanas, 2006).

Una vez definido el concepto de convexo se está en disposición de definir el concepto de envolvente (o cierre) convexa. *La **envolvente convexa** de un conjunto de puntos P es la intersección de todos los semiplanos que contienen a P* (O'Rourke, 1994).

Otras definiciones de envolvente convexa se enumeran a continuación:

- La envolvente convexa de un conjunto de puntos P es el conjunto de todas las combinaciones convexas de puntos de P. Una combinación convexa de puntos p_1, …, p_k es una suma de la forma $\alpha_1 p_1 + \cdots + \alpha_k p_k$ con $\alpha_i \geq 0$ para todo i y además $\alpha_1 + \cdots + \alpha_k = 1$ (O'Rourke, 1994).

- La envolvente convexa de un conjunto de puntos P en d dimensiones es el conjunto de todas las combinaciones convexas de d + 1 puntos (o menos) de P (O'Rourke, 1994).

- La envolvente convexa de un conjunto de puntos P es el menor conjunto convexo que contiene a dichos puntos, es decir, la intersección de todos los conjuntos convexos en los que P está contenido.

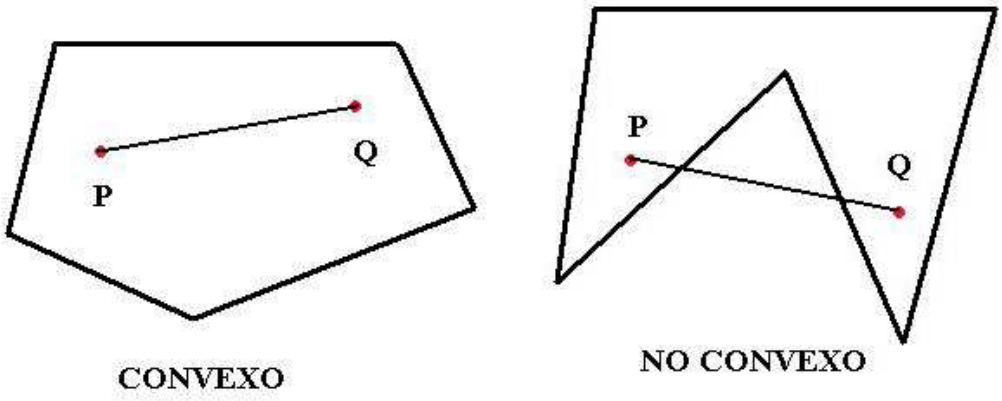


Imagen 3.1. Ejemplo de polígono convexo (izquierda) y no convexo (derecha).

3.1.2.Propiedades

La envolvente convexa presenta las siguientes propiedades:

- Siendo f una transformación afín y $C(\{p_1, p_2, \ldots, p_n\})$ la envolvente convexa de un conjunto P de puntos se cumple que:

$$f(C(\{p_1, p_2, \ldots, p_n\}) = C(\{f(p_1), f(p_2), \ldots, f(p_n)\})$$

Es decir las transformaciones afines (Por ejemplo: rotación, reflexión, traslación,…) respetan envolventes convexas.

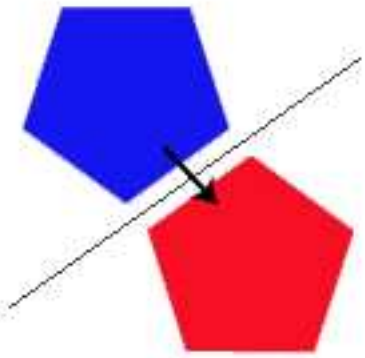


Imagen 3.2. Ejemplo representativo de transformación afín sobre conjuntos convexos.

- El cálculo de la envolvente convexa (en dos dimensiones) consiste en encontrar todos los puntos extremos de un conjunto de puntos P. Un punto p perteneciente al conjunto P es interior si existe un triángulo con vértices en P (distintos de p) de forma que p esté dentro del triángulo. (Ortega, 2006)

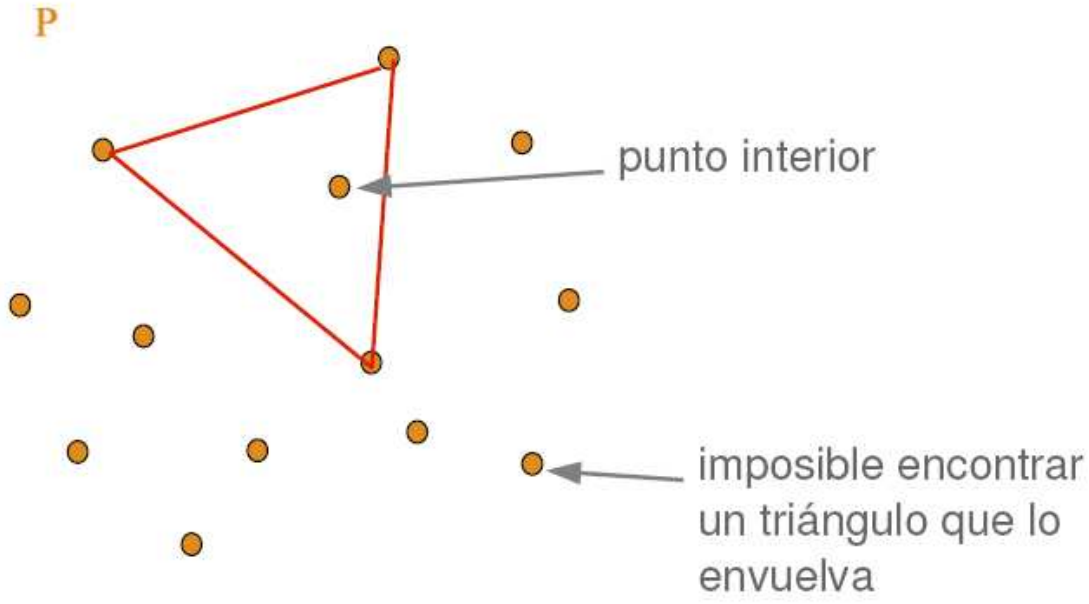


Imagen 3.3. Ejemplo ilustrativo de la segunda propiedad.

- Un punto p perteneciente al conjunto P es extremo si existe una recta pasando por p que deje al resto de puntos de P hacia un lado de dicha recta (recuérdese que se está trabajando en dos dimensiones). (Ortega, 2006)

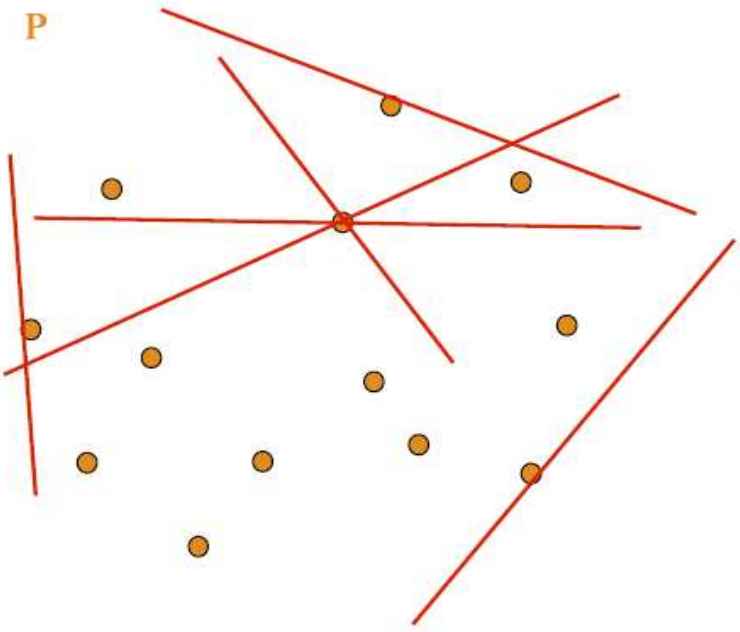


Imagen 3.4. Ejemplo ilustrativo de la tercera propiedad.

3.1.3. Historia de los algoritmos para el cálculo de la envolvente convexa

El primer método que utilizó el hombre para calcular la envolvente era por fuerza bruta, dicho proceso tenía un coste temporal de $O(n^4)$. Este coste es demasiado elevado y en 1972 Graham presentó el primer algoritmo de coste $O(n \log n)$: el *Scan de Graham*. En el empeño por mejorar este coste surgió el primer algoritmo de tiempo lineal, que fue propuesto por Sklansky en 1972. Se trataba de un algoritmo corto y elegante, a pesar de superar la cota temporal de $O(n \log n)$. Sin embargo y desafortunadamente se trataba de un algoritmo incorrecto. En 1978 (Shamos, 1978) se había demostrado que encontrar la envolvente convexa de un conjunto de puntos tenía un coste exacto de $n \log n$ (actualmente se mantiene este coste bajo cierto modelo de computación), y ya se habían presentado algoritmos sencillos para hacerlo con dicho coste. Evidentemente, dichos algoritmos podían utilizarse para encontrar la envolvente convexa de cualquier polígono, teniendo en cuenta únicamente las coordenadas de todos los vértices. Parece razonable pensar que la estructura de los polígonos simples podría proporcionar información suficiente para reducir la complejidad de encontrar envolventes convexas.

Para el caso de polígonos simples el primer algoritmo correcto de tiempo lineal fue propuesto por McCallum y Avis en 1979 El algoritmo de tiempo lineal generalmente aceptado como el "mejor" hasta el momento fue propuesto por Melkman en 1987. Parece poco probable que dicho algoritmo sea superado.

En el enlace que se presenta a continuación se puede ver una lista de algoritmos para el cálculo de la envolvente convexa en el caso de polígonos simples:

http://cgm.cs.mcgill.ca/~athens/cs601/

3.1.4.Algoritmos para el cálculo de la envolvente convexa

En este apartado lo que se pretende es dar una breve explicación de los algoritmos más importantes para el cálculo de la envolvente convexa en el caso de estar trabajando sobre cualquier conjunto de puntos.

3.1.4.1. Algoritmo de Graham

El algoritmo de Graham fue publicado en 1972 y consta de dos fases:

En la primera se ordenan los puntos angularmente alrededor de un punto interior al cierre convexo. Para hallar un punto interior al cierre convexo basta tomar el baricentro de tres de los puntos dados que no estén alineados.

En la segunda se suprimen de la lista ordenada los puntos que no son vértices del cierre convexo. Para ello se hace un recorrido de la lista (conocido como el scan de Graham) en el que se emplean áreas signadas para decidir si un punto debe ser eliminado o no. Si los puntos están ordenados en sentido antihorario, el área signada de los puntos P_{i-1}, P_i, P_{i+1} nos dice si el punto P_i debe ser eliminado. En efecto, si el área no es positiva, el punto P_i es interior al triángulo P_{i-1}, O, $P_{i+1,}$ donde O es el centro de la ordenación, y, por tanto, interior al cierre convexo. En este caso debe ser eliminado. Para que esta segunda fase proporcione la lista correcta de vértices del cierre convexo es preciso incluir un detalle más: cada vez que se elimine un punto de la lista, se debe dar un paso atrás en el recorrido, ya que al eliminar un punto puede suceder que los tres que pasan a ser consecutivos den lugar a un área signada no positiva, en cuyo caso habría que eliminar el punto intermedio de nuevo (Abellanas, 2006).

El número total de estas operaciones que se realizan es del orden de $O(n \log n)$.

El pseudocódigo correspondiente a una implementación de este algoritmo se presenta a continuación:

```
ALGORITMO GRAHAM (VAR S:NubePuntos, n:entero, VAR
Env:Poligono)
ENTRADA: La nube de puntos S de tamaño n
SALIDA: La envolvente convexa de S, CH(S)
INICIO
min = PuntoMenorOrdenada(S,n)
OrdenarAngularmente(S,n,min)
PilaCrea(Env)
PilaPush(Env,S[0])
PilaPush(Env,S[1])
PilaPush(Env,S[2])
```

```
i = 3
MIENTRAS i < n-1 REPETIR
      t = PilaPop(Env)
      SI Izquierda (PilaTope(Env), t, S[i])
      ENTONCES
      PilaPush (Env, t)
      PilaPush (Env, i)
      i = i+1
      FIN_SI
FIN_MIENTRAS
FIN
```

3.1.4.2. Algoritmo de Jarvis

El algoritmo de Jarvis, propuesto en 1973, empieza buscando un vértice del cierre convexo y, a partir de él, busca sucesivamente el resto de los vértices en el orden en que aparecen en la frontera. El primer vértice V_1 puede ser el punto de menor ordenada (si hubiera varios, el de menor abscisa de entre ellos). El siguiente, V_2, es aquél tal que la recta orientada que pasa por V_1 y V_2 deja el resto de los puntos a su izquierda. Esto mismo se repite sucesivamente, de forma que la recta orientada determinada por V_i y V_{i+1} deje todos los puntos a su izquierda hasta obtener de nuevo V_1.

En cada paso se encuentra un nuevo vértice V_{i+1} del cierre convexo. Para ello se emplea de nuevo la función área signada: el signo del área signada de V_i, P_i, P_j, indica cual de los dos puntos P_i o P_j precede al otro en el orden angular respecto de V_i, y, por tanto, cuál de los dos es candidato a ser el siguiente vértice del cierre convexo. Si se compara sucesivamente el candidato obtenido con los puntos restantes, al final se obtiene el vértice buscado.

Para hallar un nuevo vértice el algoritmo recorre la lista de puntos dados, de manera que en total el algoritmo realiza una cantidad de operaciones del orden $O(nh)$ siendo h el número de vértices del cierre convexo. En el peor caso, cuando el número de vértices sea proporcional al de puntos dados, la complejidad del algoritmo es $O(n^2)$ y, por tanto, peor que el de Graham. Sin embargo, este algoritmo aventaja al de Graham en aquellos casos en los que el cierre convexo tiene pocos vértices (Abellanas, 2006).

El pseudocódigo correspondiente a una implementación de este algoritmo se presenta a continuación:

```
ALGORITMO JARVIS (VAR P:NubePuntos, VAR p: Poligono)
i = 0
p[0] = obtener_punto_mas_abajo_izquierda(P)
do
     p[i+1] =
obtener_punto_tal_que_todos_esten_a_la_derecha(p[i],
p[i+1])
     i = i + 1
while p[i] != p[0]
```

3.1.4.3. Algoritmo Quickhull

Este algoritmo, presentado por Preparata y Shamos en 1985 (Preparata y Shamos, 1985), recibe su nombre por su similitud con el algoritmo quick-short de Hoare. Este algoritmo del tipo divide y vencerás se basa en la observación de que se pueden descartar la mayoría de los puntos interiores de un conjunto determinado pasando a centrarse en los puntos restantes.

El primer paso del Quickhull es encontrar los cuatro puntos extremos en las direcciones norte, sur, este y oeste y formar un cuadrilátero con estos puntos como vértices (este cuadrilátero puede ser degenerado), ahora todos los puntos en el interior de dicho cuadrilátero no son extremos, con lo cual pueden ser descartados.

Así nos quedan puntos repartidos en cuatro regiones (o menos) no conectadas entre sí, trataremos cada una de estas regiones independientemente. En cada una de ellas encontramos el punto más alejado a la recta que define dicha región obteniendo así cuatro nuevos puntos y un polígono de a lo más ocho lados que dividirá a los puntos que no hemos eliminado en ocho regiones que tratamos individualmente siguiendo la misma regla anterior. Así se seguirá actuando para cada nueva región según las reglas anteriores hasta que no queden puntos exteriores a la figura. La figura resultante es el cierre convexo.

El algoritmo Quickhull es de complejidad cuadrática.

El pseudocódigo correspondiente a una implementación de este algoritmo se presenta a continuación:

```
ALGORITMO QuickHull(a: punto, b:punto,S:NubePuntos)
Si S={a,b} ENTONCES devolver (a,b)
SINO
     c = índice del punto con máxima distancia a (a,b)
     A = puntos a la derecha de (a,c)
```

```
        B = puntos a la derecha de (c,b)
        Devolver concatenar(QuickHull(a,c,A),QuickHull(c,b,B)
    FIN_SI
    FIN_ALGORITMO


    FUNCIÓN ConvexHull(S:NubePuntos)
    a = punto más a la derecha de S
    b = punto más a la izquierda de S
    devolver concatenar(QuickHull(a,b,S),QuickHull(b,a,S))

    FIN_FUNCIÓN
```

3.1.4.4. Algoritmo Incremental

El algoritmo incremental se basa en la siguiente idea: ir añadiendo puntos mientras se va modificando el cierre convexo.

Este algoritmo consta de tres pasos:

- Paso 1: se elige un punto. Si el nuevo punto está dentro del cierre, no hay nada que hacer. En otro caso, se borran todos los bordes del polígono que se ven desde el punto, es decir, que trazando una línea desde el punto no se choque con ninguna otra.
- Paso 2: Se añaden dos líneas para conectar el nuevo punto al resto del antiguo cierre.
- Paso 3: se repite de nuevo desde el paso 1 para los puntos que estén fuera del cierre convexo actual, hasta que todos los vértices estén dentro.

El tiempo total que toma este algoritmo es del orden de $O(n^2)$.

El pseudocódigo correspondiente a una implementación de este algoritmo se presenta a continuación:

```
ALGORITMO_INCREMENTAL_CH(S)
     Ordenar los puntos de S por su coordenada X
     CH = triangulo(p1, p2, p3)
     PARA (4 ≤ i ≤ n ) HACER
          j ← ObtenerIndiceDelPuntoMasALaDerecha()
          u = j
          MIENTRAS (pihi no_es_tangente_a CH) HACER
               SI (u ≠ j) ENTONCES
```

```
                    borrar h4 de CH
               FIN_SI
               u = u -1
          FIN_MIENTRAS
     I = j
          MIENTRAS (pihl no es tangent a CH) HACER
               SI (I ≠ u) ENTONCES
                    borrar hi de CH
               FIN_SI
               I = I + 1
          FIN_MIENTRAS
          INSERTAR pi en CH entre hu y hi
     FIN_PARA
FIN_ALGORITMO
```

3.1.5.Aplicaciones de la envolvente convexa

Las aplicaciones de la envolvente convexa son muchas y muy diversas, y entre ellas se pueden destacar:

- Evitar colisiones. Si la envolvente convexa calculada para un *robot poligonal* evita colisiones con obstáculos, entonces el robot se comportará de la misma manera. Debido a que el cálculo de caminos que evitan colisiones es más sencillo con *robots poligonales convexos* que con *robots poligonales no convexos*, la envolvente convexa se usa con frecuencia en la planificación de caminos (Meeran y Shafie, 1991).

- La caja más pequeña. El área rectangular más pequeña que encierra a un polígono tiene al menos un lado alineado con la envolvente convexa del polígono y, por lo tanto, la envolvente es calculada en el primer paso de los algoritmos de obtención del rectángulo mínimo (Toussaint, 1983).

- Análisis de formas. Las formas pueden ser clasificadas con el objetivo de compararlas por su “convex deficiency tree”, estructuras que dependen para su cálculo del cierre convexo. El análisis de formas tiene uso en campos como el reconocimiento de matrículas de automóviles (Martinsky, 2007).

- Cálculo del diámetro y anchura de un conjunto. El diámetro de una nube de puntos P es la mayor distancia posible de entre todos los puntos de P. La anchura es la menor longitud, de entre todas las posibles, entre dos rectas paralelas que contengan en su interior todos los puntos

pertenecientes a un conjunto P. El uso de estas magnitudes va desde el cálculo de medidas de dispersión hasta el cálculo de trayectoria de robots.

Otras aplicaciones son: “Ray Tracing” (Foscari, 2007), videojuegos, cálculo de accesibilidad en sistemas GIS, etc.

3.2. Introducción a las triangulaciones

En esta sección se llevará a cabo una revisión del concepto de triangulación y se explicará un tipo particular de triangulación: la triangulación de Delaunay. Asimismo se dará una breve perspectiva histórica de las triangulaciones y finalmente se mostrarán algunos de los algoritmos más destacados para el cálculo de triangulaciones y varios ejemplos de aplicación de las mismas.

3.2.1.Definición

Dados n puntos del plano P = $\{p_1, \ldots, p_n\}$ no estando tres puntos alineados (*posición general*), una **triangulación** T es una subdivisión plana maximal asociada a P, es decir, es un grafo plano (V, A) con:

- V = P
- A = segmentos entre vértices de modo que no se puedan añadir más aristas sin que se crucen (en puntos no pertenecientes al conjunto).

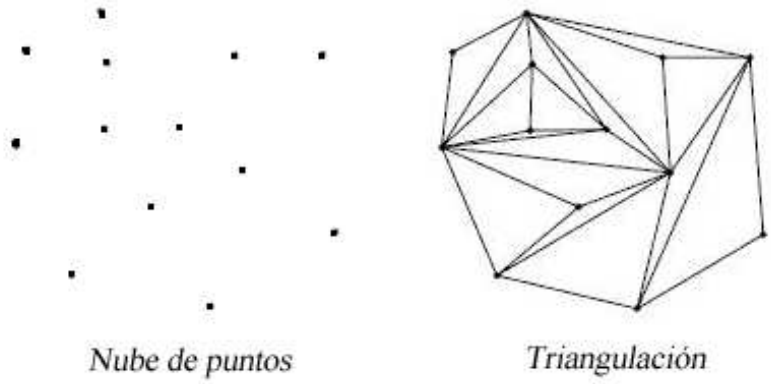


Imagen 3.5. Nube de puntos (izquierda) y una triangulación de la misma (derecha).

En cualquier triangulación de una nube de puntos, el número de triángulos y el número de aristas no varían. Por lo tanto, dado un conjunto P = $\{p_1, \ldots, p_n\}$ no todos alineados y suponiendo que la envolvente convexa de P tiene k vértices, cada triangulación T de P tiene siempre:

- $2n - 2 - k$ *triángulos*
- $3n - 3 - k$ *aristas*

3.2.2.Triangulación de Delaunay

El número de triangulaciones diferentes que se pueden obtener partiendo del mismo conjunto de puntos es elevado. El mínimo número de triangulaciones conocidas es $\sqrt{12}^{n}$ (*la configuración que da lugar a este número de triángulos se llama doble círculo*) y el máximo conocido es $\sqrt{72}^{n}$ (*la configuración que da lugar a este número de triangulaciones se llama doble cadena en zig-zag*). Una condición deseable para poder calificar una triangulación como buena es que sus triángulos sean, si no regulares, si lo más regulares posible (Abellanas, 2006). Una de estas triangulaciones calificada como buena es la de Delaunay. Se considera buena o regular en el sentido de que maximiza el menor ángulo.

Además la triangulación de Delaunay cumple la condición de Delaunay. Dicha condición afirma que la circunferencia circunscrita de un triángulo es *vacía*, es decir, no contiene otros vértices aparte de los tres que la definen.

Esa es la definición original para dos dimensiones y es posible ampliarla para tres dimensiones usando la esfera circunscrita en vez de la circunferencia circunscrita. También es posible ampliarla para espacios con más dimensiones, aunque no se usa en la práctica.

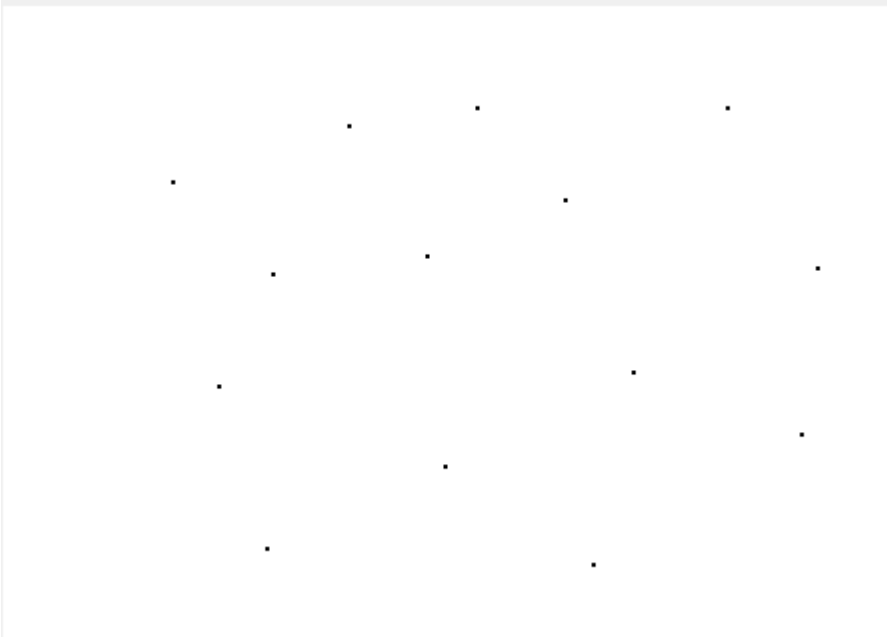


Imagen 3.6. Nube de puntos en el plano

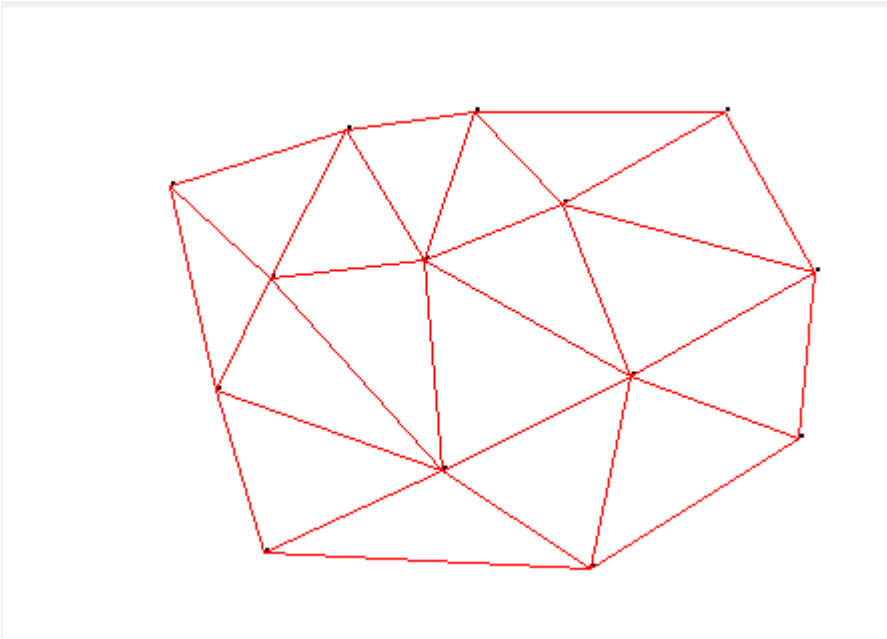


Imagen 3.7. Triangulación de Delaunay de la nube de puntos de la figura 3.6.

La triangulación de Delaunay tiene las siguientes propiedades:

- La envolvente convexa forma parte de la triangulación del conjunto de puntos.
- El ángulo mínimo sobre el conjunto de todos los triángulos está maximizado, es decir la triangulación de Delaunay es aquella en la cual el mínimo valor de ángulo de sus triángulos es máximo sobre el conjunto de todas las triangulaciones.
- La triangulación es única si no hay más de tres vértices sobre una misma circunferencia (*posición general*).

3.2.3.Historia de las mallas de triángulos

El uso sistemático de las triangulaciones modernas se deriva de la labor del matemático holandés Willebrord Snell, quien en 1615 estudió la distancia de Alkmaar a Bergen-op-Zoom, que era de aproximadamente 70 millas (110 kilómetros), utilizando una cadena de 33 triángulos cuadriláteros. Estas dos ciudades estaban separadas por un grado sobre el meridiano y con el fin de medir esta distancia fue capaz de calcular un valor para la circunferencia de la Tierra, una hazaña que se celebra en el libro titulado Eratóstenes Batavus (El holandés Eratóstenes), publicado en 1617. Snell calculó la

forma en que la fórmula para el plano podría corregirse para permitir la inclusión de la curvatura de la Tierra. También mostró cómo calcular la posición de un punto dentro de un triángulo mediante el reparto entre los ángulos de los vértices en el punto desconocido.

Los métodos de Snell fueron tomados por Jean Picard que en 1669-70 estudió un grado de latitud a lo largo del meridiano de París utilizando una cadena de trece triángulos que se extiende desde el norte de París hasta la torre del reloj de Sourdon, cerca de Amiens. Gracias a la mejora en la exactitud de los instrumentos, se califica a la medición de Picard como la primera medición razonablemente precisa del radio de la Tierra. La familia Cassini continuó con este trabajo a lo largo del siglo siguiente: entre 1683 y 1718 Jean-Dominique Cassini y su hijo Jacques Cassini estudiaron todo el meridiano de París desde Dunquerque a Perpignan, y entre 1733 y 1740, Jacques y su hijo César Cassini llevaron a cabo la primera triangulación de todo el país, incluido un nuevo estudio de los meridianos, lo que condujo a la publicación en 1745 del primer mapa de Francia construido sobre principios rigurosos.

Hasta este momento los métodos de triangulación estaban establecidos en la cartografía local, pero no fue hasta finales del siglo XVIII cuando otros países comenzaron a establecer los estudios detallados de redes de triangulación para realizar mapas de países enteros. La principal triangulación de Gran Bretaña fue iniciada por el *Ordnance Survey* en 1783, aunque no se terminó hasta 1853, y el Gran Estudio Trigonométrico de la India, que finalmente puso nombre y cartografió el Monte Everest y los otros picos del Himalaya, se inició en 1801. Mientras tanto, al famoso matemático Carl Friedrich Gauss le fue confiada la triangulación del reino de Hannover desde 1821 hasta 1825. Para esta triangulación desarrolló el método de los mínimos cuadrados para encontrar la mejor solución para los problemas de ajuste de los grandes sistemas de ecuaciones simultáneas.

Estas redes de triangulación de posicionamiento a gran escala se han seguido utilizando de manera similar hasta que recientemente han sido prácticamente sustituidas por el Sistema mundial de navegación por satélite, establecido desde la década de 1980. Sin embargo, muchos de los puntos de control (en España llamados vértices geodésicos) para los estudios anteriores aún sobreviven como un valor histórico en el paisaje, como ocurre con los pilares de hormigón usados para retriangular Gran Bretaña (1936-1962), o la creación de puntos de triangulación, para el Arco geodésico de Struve (1816-1855), nombrado por la UNESCO como Patrimonio de la Humanidad.

3.2.4. Algoritmos para el cálculo de la triangulación de Delaunay

En este apartado lo que se pretende es dar una breve explicación del algoritmo más sencillo de entender para el cálculo de la triangulación de Delaunay: el algoritmo incremental.

El algoritmo incremental tiene una complejidad en el peor de los casos de $O(n^2)$, en la práctica esta complejidad es de $O(n^{\frac{3}{2}})$. Además existe una versión incremental aleatorizada con una complejidad $O(n \log n)$.

La idea es sencilla, supongamos que partimos de una triangulación de Delaunay y en cada paso se añade un punto a la misma. La adición de un vértice a una triangulación de Delaunay puede provocar una serie de giros en las aristas que forman los triángulos dando lugar a una nueva triangulación de Delaunay que incluye el nuevo vértice.

A continuación se muestra un pseudocódigo de una implementación del algoritmo incremental:

```
Función Delaunay(Conjunto de puntos: P, Salida: T)
        ST = triángulo de área infinita
        Añadir(T, triángulo ST)
        Para cada punto p en P
           Añadir(p, T)
        Fin para cada
        Para cada triángulo t en T
           EliminarTriangulosConVerticesEnST(t, ST)
        Fin para cada
Fin función

Función Añadir (Punto : p, Triangulación T)
        Para cada triangulo t en T
           Si p está en la circunferencia circunscrita
de t
                Almacenar las aristas de t en
buffer_aristas
                Eliminar triángulo
           Fin si
        Fin para cada
        Eliminar aristas dobles de buffer_aristas
        Para cada arista a en buffer_aristas
```

```
            NuevoT = triangulo entre a y p.
            Añadir NuevoT a T
        Fin para cada
    Fin función
```

Para comprender mejor el proceso de adición de vértices se va a ejemplificar con una serie de ilustraciones.

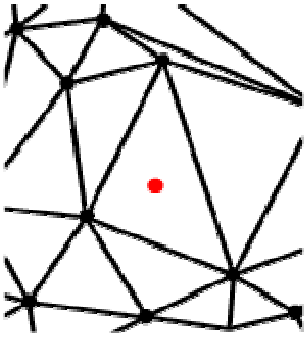


Imagen 3.8. Se quiere insertar un punto en una triangulación de Delaunay

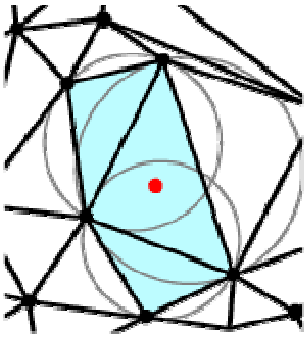


Imagen 3.9. Se buscan los triángulos que contienen al punto en su circunferencia circunscrita.

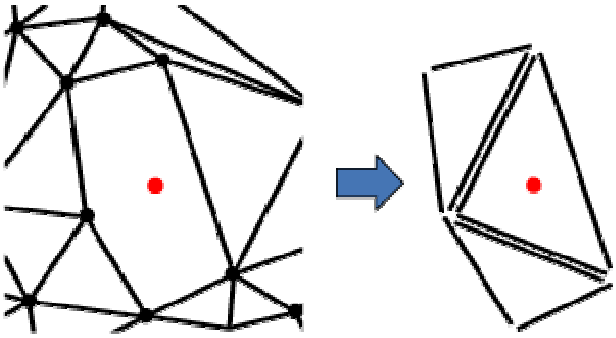


Imagen 3.10. Se eliminan los triángulos encontrados en el paso anterior. Las aristas de dichos triángulos pasan a ser *sospechosas* y se añaden a un *buffer* (derecha).

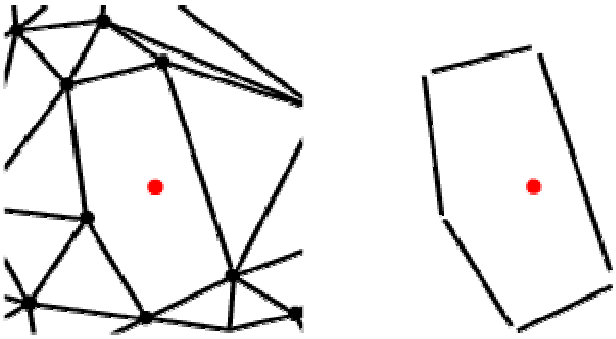


Imagen 3.11. Se eliminan las aristas dobles del *buffer*.

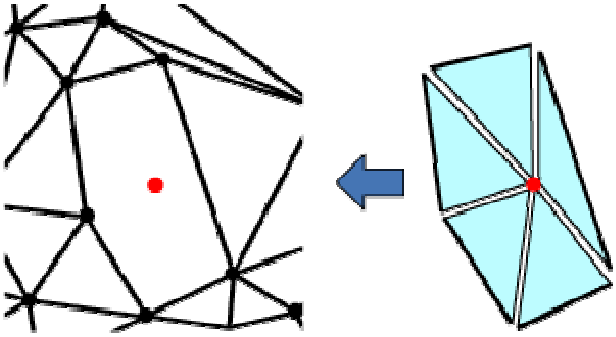


Imagen 3.12. Se forman nuevos triángulos usando las aristas restantes en el *buffer* y el punto a añadir.

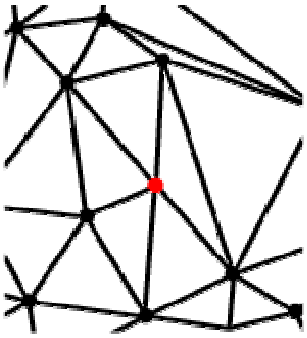


Imagen 3.13. Finalmente se añaden dichos triángulos a la triangulación.

3.2.5.Aplicaciones de las triangulaciones

Las aplicaciones de las mallas de triángulos son diversas, algunas de las más importantes son:

- En gráficos 3D de ordenador se usan redes de polígonos para modelar objetos tridimensionales, juntando los polígonos para imitar la superficie del objeto. En general se usan triángulos porque son los polígonos más simples y tienen muchas propiedades favorables, como que representan una superficie plana (Shreiner et al., 2007).

  Hay dos formas de modelar un objeto de superficies: modelarlo a mano o escanearlo con un *range scanner*. Al escanearlo se produce un relieve de la superficie formado por puntos discretos. Para usar ese relieve hay que transformarlo en una red de triángulos; esa transformación se llama "triangulación".

  La triangulación de Delaunay maximiza los ángulos interiores de los triángulos de la triangulación. Eso es muy práctico porque al usar la triangulación como modelo tridimensional los errores de redondeo son mínimos. Por eso, en general se usan triangulaciones de Delaunay en aplicaciones gráficas.

- Las mallas de triángulos también se usaron para realizar cálculos cartográficos como se ha comentado en la sección 3.2.3 (Vogel, 1997).

- En los sistemas de reconocimiento de objetos. Esta aplicación de las mallas triangulares está íntimamente ligada a la aplicación relativa a gráficos 3D (Felzenszwalb y Huttenlocher, 2005).

3.3. Conclusión de la sección

En esta sección de la memoria se ha llevado a cabo una revisión de los conceptos de envolvente convexa y triangulación. Además se han aportado algunos datos históricos, algoritmos y aplicaciones de estas estructuras.

Con ello se puede afirmar que se ha superado el primer objetivo propuesto en la presente memoria: *Revisar las nociones de envolvente convexa y triangulación sin restringir dirección alguna*.

Tras esta introducción se está en posesión de las nociones básicas necesarias para abordar el tema motivo de este proyecto: envolventes convexas y triangulaciones con direcciones restringidas.

4. Envolventes convexas y triangulaciones con direcciones restringidas

4.1. Envolvente convexa con direcciones restringidas

En esta sección se llevará a cabo un estudio de las envolventes convexas con direcciones restringidas. La base de este estudio es un artículo sobre envolventes {0, 90}-convexas (Ottman, 1984) que permite alcanzar conclusiones para el caso general.

4.1.1.Envolventes O-convexas

De un tiempo a esta parte la computación de la envolvente {0, 90}-convexa de una colección de polígonos rectilíneos (donde rectilíneo se refiere a que las direcciones del polígono son únicamente 0º y 90º) ha sido estudiada por varios autores. De estos estudios han surgido tres definiciones de envolvente convexa rectilínea. En esta sección se llevará a cabo un examen de dichas tres definiciones.

*Un conjunto de puntos en el plano se dice que es **convexo** si su intersección con cualquier recta del plano es conexa*. Análogamente, *un conjunto de puntos en el plano es **O-convexo** si su intersección con cualquier recta cuya dirección está en O es conexa*.

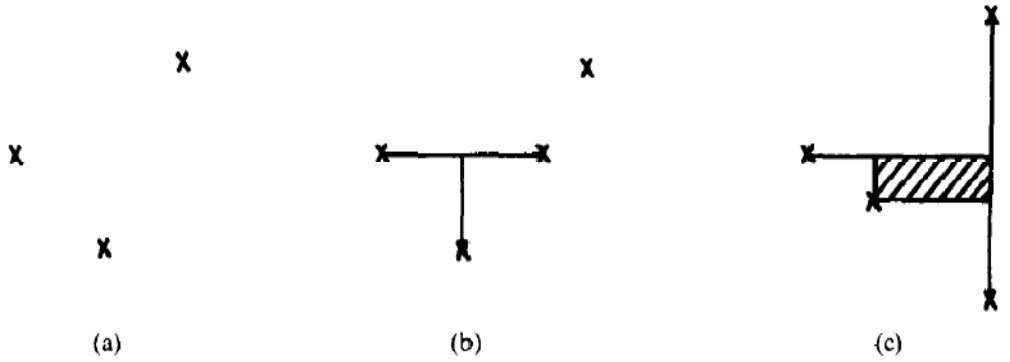


Imagen 4.1. Tres conjuntos {0, 90}-convexos

Obsérvese que un conjunto O-convexo puede ser disconexo, como en la Imagen 4.1. (a) y (b). Para conjuntos convexos no se produce esta situación.

Para definir el cierre O-convexo de un conjunto de puntos hay varias posibilidades. A continuación se muestran las tres alternativas introducidas en la literatura:

Alternativa 4.1.: *Dado un conjunto de puntos, su cierre O-convexo es el menor O-convexo que contiene al conjunto dado.* (Ottman, Soisalon-Soininen y Wood, 1983)

En la Imagen 4.1. las cruces indican los puntos originales del conjunto cuyos cierres {0, 90}-convexos están dibujados.

Obsérvese que esta definición de cierre O-convexo es equivalente a decir que: el cierre O-convexo de un conjunto de puntos es la intersección de todos los conjuntos O-convexos que contienen al conjunto dado.

Esta primera definición de cierre O-convexo da lugar a un único cierre O-convexo. Desafortunadamente el cierre O-convexo resultante podría ser un conjunto disconexo, mientras que el cierre convexo en sentido usual siempre es conexo.

Presumiblemente tanto Montuo y Fournier (Montuo y Fournier, 1982) como Nicholl (Nicholl et al., 1983) tuvieron esto en mente cuando pusieron como requisito que el cierre O-convexo fuera conexo, dando lugar a la segunda definición de cierre O-convexo (definición conexa):

Alternativa 4.2.: *Dado un conjunto de puntos, su cierre O-convexo es el menor O-convexo conexo que contiene el conjunto dado*.

Obsérvese que dichos cierres O-convexos no son necesariamente únicos, un hecho que inicialmente fue pasado por alto (Nicholl et al., 1983). En la Imagen 4.2 se muestran tres cierres {0, 90}-convexos diferentes de un conjunto de puntos según esta definición.

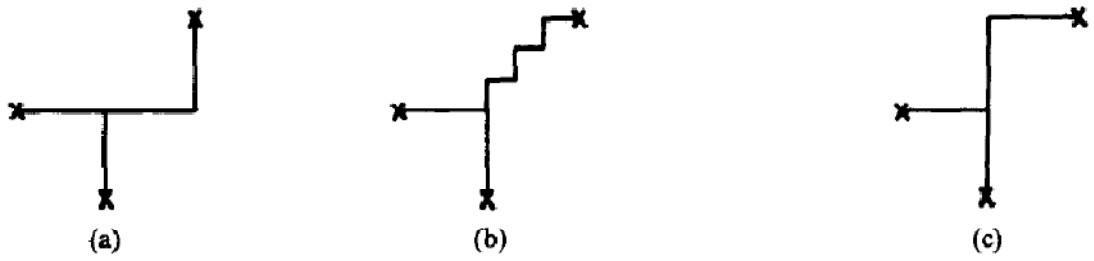


Imagen 4.2. Tres posibles cierres {0, 90}-convexos de un conjunto según la Alternativa 4.2.

Hay de hecho un número infinito de cierres O-convexos (según la definición de la Alternativa 4.2) distintos en el caso representado en la Imagen 4.2.

A pesar de que el mayor inconveniente de los cierres O-convexos según esta definición es que no son únicos, existen otros inconvenientes. Si se intenta hacer lo mismo que para la Alternativa 4.1. y se considera la intersección de todos los conjuntos O-convexos conexos que contienen al conjunto de puntos dado, el resultado no se ajusta a ninguna de las definiciones de las alternativas 4.1 y 4.2. Por ejemplo, los tres conjuntos de puntos de la Imagen 4.2. dan lugar a la Imagen 4.3.

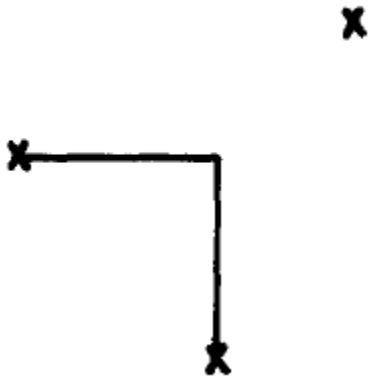


Imagen 4.3. Cierre {0, 90}-convexo según la definición 4.3.

Esto lleva a una tercera definición:

Definición 4.3.: *Se define **escalera** como:*

a) Un segmento cuya dirección está en O.

*b) Dadas dos direcciones d_1 y d_2, consecutivas en O, se dice que una secuencia de segmentos $l_1, l_2, \ldots, l_m$ es una **escalera** si:*

- *l_i se encuentra con l_{i+1} y l_{i-1} en sus puntos extremos.*
- *las direcciones de $l_1, l_2, \ldots, l_m$ alternan entre d_1 y d_2.*
- *la cadena es monótona en ambas direcciones d_1 y d_2.*

y se llama semiplano escalera a cualquiera de las dos partes del plano en que éste queda dividido por una escalera (Rawlins y Wood, 1987).

Dado un conjunto de puntos, su cierre O-convexo es la intersección de todos los semiplanos escalera cerrados que lo contienen.

En esta sección se han considerado tres posibles definiciones de cierre convexo rectilíneo de un conjunto de puntos, cada una de las cuales tiene sus ventajas e inconvenientes. La siguiente tabla resume sus propiedades:

	Cierre O-convexo (Alternativa 4.1)	**Cierre O-convexo (Alternativa 4.2)**	**Cierre O-convexo (Definición 4.3)**
Conexo	No necesariamente	Si	No necesariamente
Único	Si	No necesariamente	Si
Consistente	Si	No	Si
Estable	No	Si	Si

Tabla 4.1. Tabla resumen de los cierres O-convexos.

En la tabla anterior la palabra *Consistente* significa con definiciones alternativas.

Por otro lado, *Estable* hace referencia a la capacidad de calcular el cierre O-convexo de manera dinámica.

Observando esta tabla la definición de envolvente O-convexa que presenta la mejor relación de propiedades deseables es la Definición 4.3. Por lo tanto de aquí en adelante se trabajará sobre dicha definición.

Una vez elegida una definición es posible dar un algoritmo para el cálculo de las envolventes O-convexas presentadas en la Definición 4.3. El algoritmo se presenta a continuación:

1) Como entrada del mismo se tiene un conjunto de direcciones $O = \{o_1,...,o_i,[o_{i+1},o_{i+2}],o_{i+3}\ldots o_k,o_1+180,...,o_i+180,[o_{i+1}+180,o_{i+2}+180], o_{i+3}+180\ldots o_k+180\}$ con o_1, …, o_k en [0,180) y ordenadas de menor a mayor y un conjunto de puntos P.

2) Se calcula el conjunto $N = \{n_1,...,n_i,n_{i+1},n_{i+2},n_{i+3},\ldots,n_k, n_1+180,..., n_i+180,n_{i+1}+180,n_{i+2}+180,n_{i+3}+180,\ldots,n_k+180\}$ de las direcciones resultantes de sumar 90 a las de O (para los intervalos sólo se suma 90 a las direcciones extremas del intervalo) y se ordenan de menor a mayor.

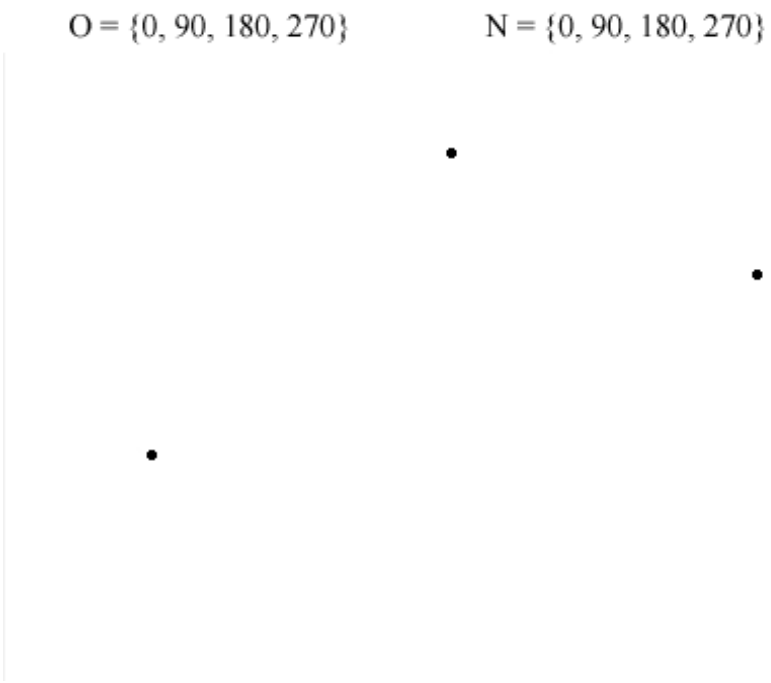


Imagen 4.4. Paso dos del algoritmo.

3) Para cada dirección en N encontrar el punto extremo en P correspondiente.

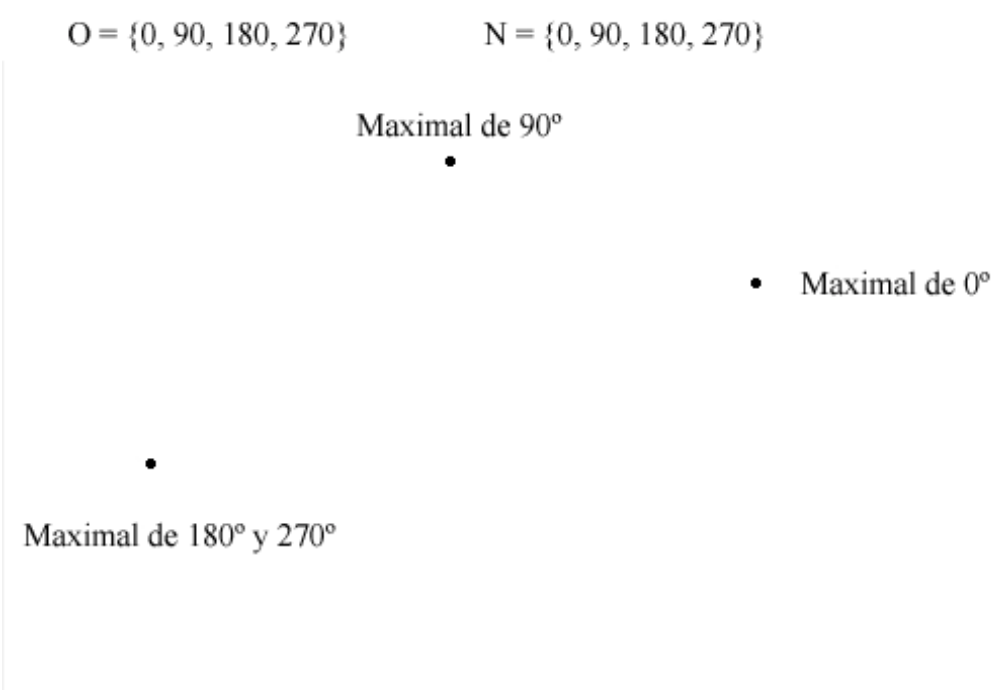


Imagen 4.5. Paso tres del algoritmo.

4) Para cada par de direcciones n_i, n_{i+1} (recuérdese que i+1 es la dirección consecutiva a i) construir la escalera correspondiente entre sus puntos extremos p_i, p_{i+1} de la siguiente forma:

    a. Si $p_i = p_{i+1}$ se devuelve p_i. Si no:

    b. Si la dirección del segmento que va de p_i a p_{i+1} está contenida en O, se devuelve el segmento que va de p_i a p_{i+1}. Si no:

    c. Considerar el rayo que sale de p_i y tiene dirección $n_{i+1}+90$.

    d. Sobre ese rayo deslizar el extremo de un segundo rayo con dirección n_i+90

    e. Cuando este segundo rayo se encuentre con un punto q del conjunto P:

        i. Si $q = p_{i+1}$, añadir a la escalera el segmento recorrido sobre el primer rayo, junto con el segmento del segundo rayo hasta q. Al llegar a p_{i+1} la escalera está acabada y pasamos al siguiente par de direcciones. Si $q \neq p_{i+1}$ entonces:

        ii. Añadir a la escalera el segmento recorrido sobre el primer rayo, junto con el segmento del segundo rayo hasta q.

        iii. Se actualiza el valor de p_i al de q, $p_i = q$, y se repite el proceso.

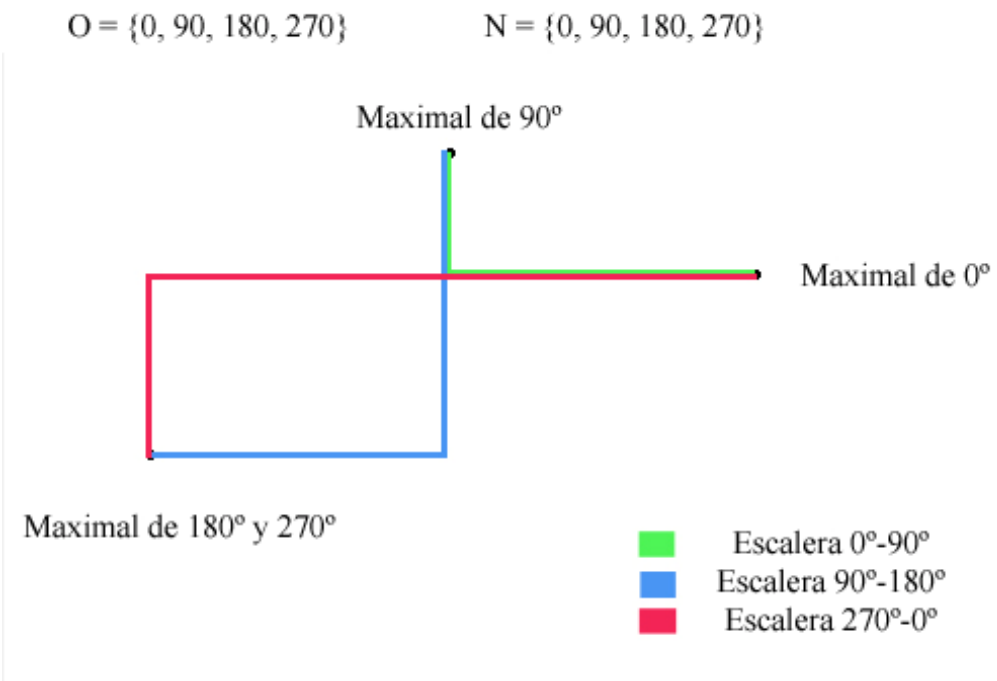


Imagen 4.6. Paso cuatro del algoritmo.

5) Una vez construidas todas las escaleras ya se pueden obtener los semiplanos escalera. Al hacer la intersección de los mismos se obtiene la envolvente O-convexa del conjunto P.

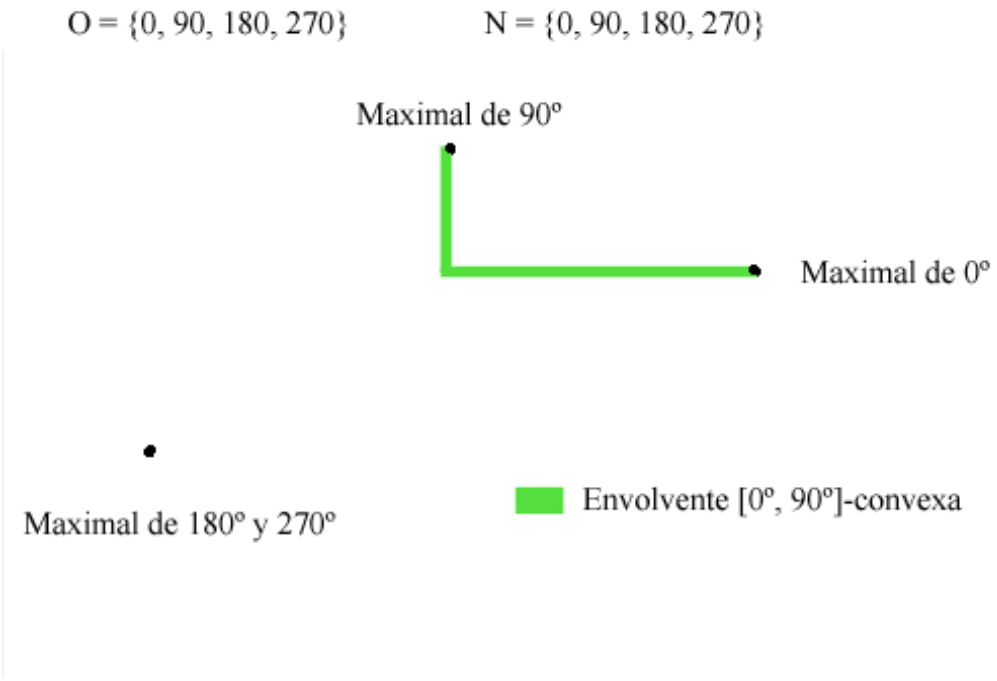


Imagen 4.7. Paso cinco del algoritmo.

Un posible pseudocódigo para este algoritmo es:

```
FUNCIÓN CH_InterseccionSemiplanosEscalera(conjunto de direcciones: O, conjunto de puntos: P)

	//Calcular conjunto N

	Conjunto de direcciones: N

	PARA CADA dirección D en O HACER

	Añadir(N, D + 90)

	FIN PARA CADA

	OrdenarMenorMayor(N)

	Conjunto de puntos: Extremos

	PARA CADA dirección D en N HACER

	//Calcular el punto de P que es extremo en la dirección D y añadirlo a Extremos

	Añadir(Extremos, CalcularPuntoExtremo(D, P))

	FIN PARA CADA

	//Para cada par de direcciones consecutivas, en sentido anti-horario, se construye una escalera entre sus puntos extremos

	Lista escaleras: Escaleras

	PARA CADA PAR DE direcciones d1 Y d2 (consecutivas) HACER

	//Donde CrearEscalera es una función que implementa lo descrito en 4)

	Añadir(Escaleras, CrearEscalera(Extremos[d1], Extremos[d2])

	FIN PARA CADA

	//Interseccion es una función que lleva a cabo la intersección de todos los semiplanos escalera.

	DEVOLVER Interseccion(Escaleras)

FIN FUNCIÓN
```

La esencia de este algoritmo ha sido extraída de uno de los algoritmos para el cálculo de envolventes {0º, 90º}-convexas presentados por Ottman (Ottman, 1984).

Uno de los problemas que surgen con este algoritmo es el cálculo de puntos maximales en una dirección. Este problema tiene una fácil solución aplicando una transformación afín: una rotación respecto al origen. Se llevará a cabo una rotación sobre el plano de tal forma que la dirección especificada pase a ser el eje X (o el eje Y), de manera que el problema se reduce a calcular el punto de mayor coordenada X (o el de mayor coordenada Y). Otro de los problemas, pero desde el punto de vista programático, puede ser el cálculo de la intersección de semiplanos escalera y, por lo tanto, será tratado en la sección correspondiente.

En cuanto al coste, sea n el número de puntos y sea k el número de direcciones, el coste en el peor caso (el peor caso es que haya que calcular intersecciones de semiplanos escalera) es $O(n \log n) + O(kn) + O\left(\frac{n^2}{4}\right)$.

4.1.2. Triangulaciones con direcciones restringidas

Como se estudió en el apartado tercero una triangulación está formada por un conjunto de triángulos. Un triángulo se puede definir como la envolvente convexa de un conjunto de tres puntos, como consecuencia, se puede definir un O-triángulo como la envolvente O-convexa de tres puntos.

Después de llevar a cabo un estudio de los O-triángulos se puede afirmar que existen tres posibles casos:

i. O-triángulos formados por una sola componente conexa: son aquellos en los que la intersección de los semiplanos escalera es vacía.

ii. O-triángulos formados por dos componentes conexas: son aquellos en los que hay una componente formada por un solo punto y otra componente formada por los dos restantes. Se debe a que hay una intersección.

iii. O-triángulos formados por tres componentes conexas: son aquellos en los que cada componente es uno de los vértices. Se debe a que la intersección de semiplanos escalera da como resultado el propio conjunto.

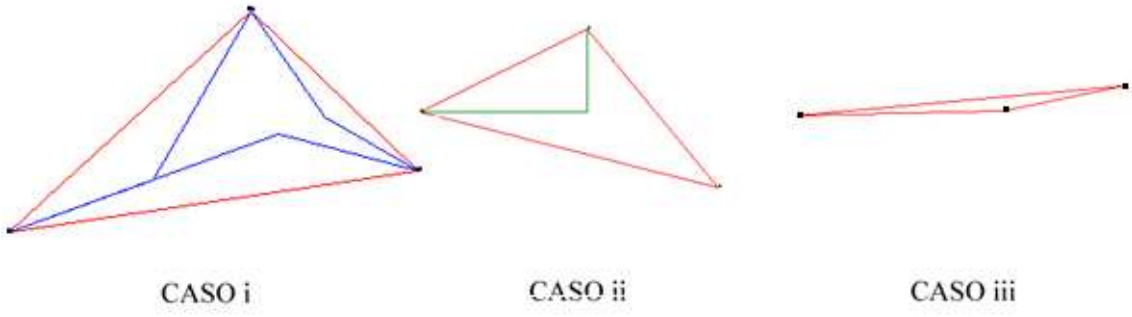


Imagen 4.8. Representación de los tres posibles casos de O-triángulos. Los triángulos originales aparecen en color rojo. En el caso ii O = {0º, 90º}. En el caso iii el O-triángulo está formado únicamente por los tres puntos.

Dado que una triangulación está formada por un conjunto de triángulos, para calcular una O-triangulación, bastará con calcular la envolvente O-convexa para cada triángulo de la misma. Hay que observar que el perímetro de una triangulación de un conjunto de puntos en el caso general es la envolvente convexa de dicho conjunto, cosa que no ocurre para el caso de direcciones restringidas. Esto se puede observar en la siguiente imagen:

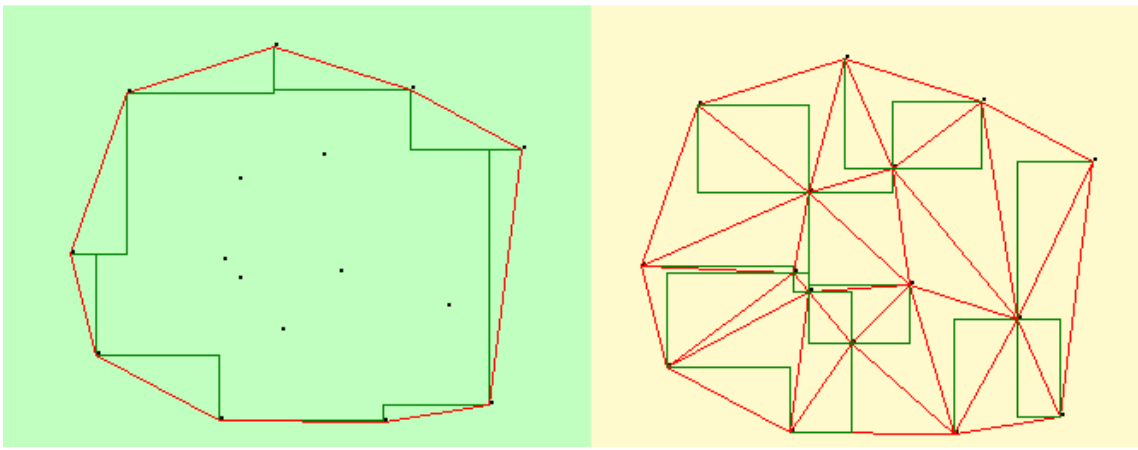


Imagen 4.9. Comparación entre envolvente O-convexa y O-triangulación.

En la parte izquierda de la Imagen 4.9 se puede ver la envolvente convexa (en rojo) y la envolvente {0, 90}-convexa de un conjunto de puntos. En la parte derecha se puede ver la triangulación de Delaunay de dicho conjunto de puntos (en rojo) y la O-triangulación correspondiente. Mientras que en el caso habitual de envolvente convexa y triangulación se cumple que el perímetro de la triangulación es la envolvente convexa, se puede apreciar que en el caso restringido esto no es así.

4.1.3.Cálculo de envolventes O-convexas mediante *Inflado*

Una vez expuesta la base teórica en lo referente a cierres O-convexos y O-triángulos se va a presentar un algoritmo que permite apreciar las envolventes O-convexas de una manera mucho más visual e intuitiva. Durante la labor de investigación de este algoritmo se ha podido observar una peculiaridad de las envolventes O-convexas que será mostrada a lo largo de los siguientes párrafos.

El algoritmo, denominado algoritmo de inflado de paralelepípedos es el siguiente:

1) Como entrada se tiene un conjunto de direcciones $O = \{o_1,...,o_i,[o_{i+1},o_{i+2}],o_{i+3}\ldots o_k,o_1+180,...,o_i+180,[o_{i+1}+180,o_{i+2}+180],$ $o_{i+3}+180\ldots o_k+180\}$ con o_1, …, o_k en [0,180) y ordenadas de menor a mayor y un conjunto de puntos P.

2) Hacer la envolvente convexa en el sentido general para el conjunto P.

3) Para cada arista del polígono (obtenido al hacer el paso 2) hacer:

   a. Comprobar si la dirección de la arista pertenece al conjunto O. Si pertenece se pasa a la siguiente arista. En caso contrario:

   b. Situar el origen de coordenadas en su punto medio, llamar *p* al extremo que está en el semiplano superior y *q* al otro y comprobar las direcciones del conjunto O (teniendo solo en cuenta las direcciones de O entre 0º y 180º) entre las que se encuentra la dirección, *d*, de la arista. Estas direcciones serán su $tope^+$ y su $tope^-$.

   c. Sustituir el segmento *pq* por el paralelepípedo con vértices *p* y *q* y con direcciones $d^+=d$+precisión, $d^-=d$-precisión, donde "precisión" es la distancia entre dos ángulos de inflado (para barrer todos los posibles ángulos "precisión" debería ser un valor infinitamente pequeño).

      Observación: desde *q* se usarán las direcciones con el sentido original (entre 0º y 180º), mientras que desde *p* se usarán con el sentido contrario.

   d. Sustituir el paralelepípedo anterior por el que se obtiene al hacer $d^+=d^+$+precisión, $d^-=d^-$-precisión. Si en algún paso d^+ alcanza $tope^+$ o d^- alcanza $tope^-$, en los pasos posteriores se mantiene ese valor (sin sumar o restar "precisión"). Se repite el paso d hasta que ambas direcciones alcanzan su tope.

e. Si en algún paso del inflado el paralelepípedo se topa con un punto, *i*, interior del polígono habrá que sustituir la arista que se está inflando por las aristas *pi* e *iq* y comenzar el proceso desde el paso 3 para estas nuevas aristas.

4) Una vez obtenidos todos los paralelepípedos se procederá a realizar la siguiente operación: para cada dos paralelepípedos se realiza su intersección con el objetivo de añadir a cada uno de los paralelepípedos los puntos en que ambos polígonos se intersecan como vértices extras. Dichos vértices no cambiarán la forma de los paralelepípedos pero proporcionan una información de gran importancia para obtener el resultado.

5) Para cada paralelepípedo considerar cada uno de los segmentos que lo forman (teniendo en cuenta los vértices extra añadidos en el paso 4). Si los dos extremos de un segmento están dentro del polígono derivado del original que da como resultado el paso tercero (hay que recordar que si en el proceso de inflado una arista se topa con un punto interior esta es sustituida por las aristas que quedan definidas por el punto interior y los dos extremos de dicha arista), entonces dicho segmento pertenece a la envolvente O-convexa.

Del paso quinto se ha obtenido la siguiente observación: *la envolvente O-convexa está contenida dentro del área del polígono derivado de la envolvente convexa original y que se obtiene al incluir en el perímetro del polígono los puntos interiores que pertenecerán a la envolvente O-convexa y que han sido hallados en el paso tercero del algoritmo*.

A continuación se muestran una serie de ilustraciones sobre el algoritmo de inflado:

Imagen 4.10. Conjunto de puntos sobre el que se calculará el cierre {0º, 90º}-convexo por inflado.

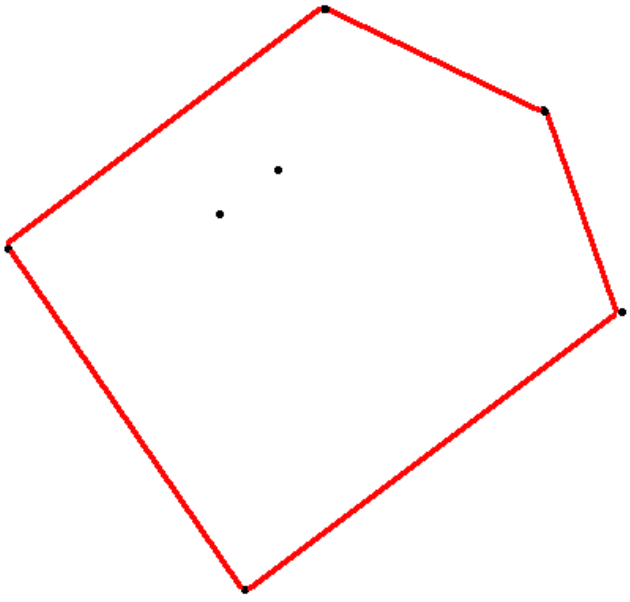


Imagen 4.11. Envolvente convexa del conjunto de puntos de la Imagen 4.10. en el caso habitual.

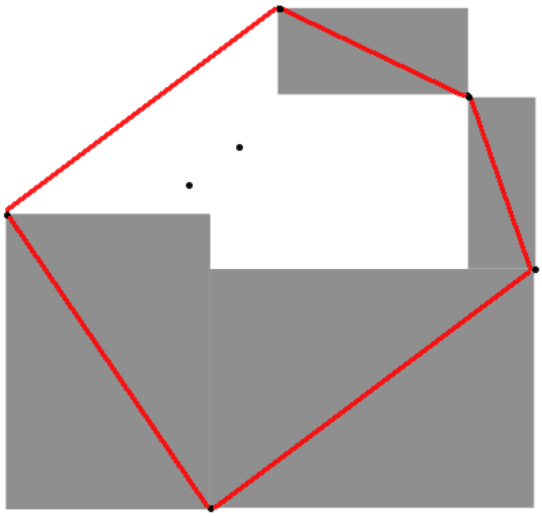


Imagen 4.12. Se aplica el paso tercero a las aristas del polígono para ir consiguiendo los paralelepípedos correspondientes.

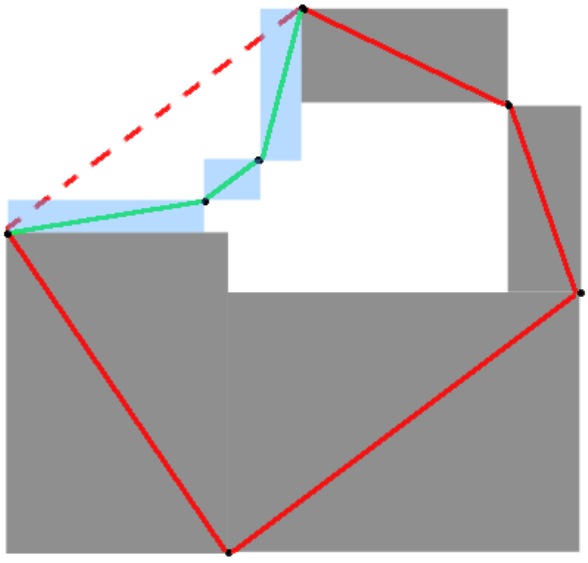


Imagen 4.13. Para la última arista a inflar (punteada en rojo) hay puntos intermedios por lo que es sustituida por las nuevas aristas (en verde). Sobre estas nuevas aristas se realiza el proceso de inflado.

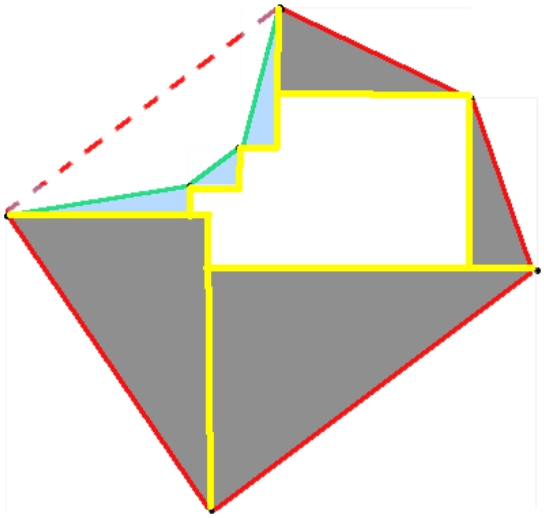


Imagen 4.14. Finalmente se aplican los pasos cuarto y quinto del algoritmo y se obtiene el resultado: la envolvente {0º, 90º}-convexa (en amarillo).

Este algoritmo no será implementado en el software resultante del proyecto actual pero en su lugar será implementado un caso particular del mismo explicado en la sección presentada a continuación.

4.1.4.Cálculo de O-triangulaciones mediante *Inflado*

Como ya se mencionó en el apartado 4.1.2. para calcular una O-triangulación basta con calcular la envolvente O-convexa de los triángulos que la compone. Esto puede ser aplicado también al algoritmo de inflado. Sin embargo para el caso de inflado en triángulos se ha observado que no es necesario contemplar el caso de encontrar puntos intermedios durante el inflado de una arista por lo cual se elimina uno de los pasos del algoritmo. Al eliminar este paso, la comprobación del paso quinto del algoritmo dado en la sección anterior se realiza sobre la envolvente convexa en el caso habitual.

El algoritmo de inflado de paralelepípedos, aplicado a triángulos, es el siguiente:

1) Como entrada se tiene un conjunto de direcciones $O = \{o_1,...,o_i,[o_{i+1},o_{i+2}],o_{i+3}\ldots o_k,o_1+180,...,o_i+180,[o_{i+1}+180,o_{i+2}+180], o_{i+3}+180\ldots o_k+180\}$ con o_1, …, o_k en [0,180) y ordenadas de menor a mayor y un conjunto de puntos P.

2) Hacer la envolvente convexa en el sentido general para el conjunto P.

3) Para cada arista del polígono (obtenido al hacer el paso 2) hacer:

   a. Comprobar si la dirección de la arista pertenece al conjunto O. Si pertenece se pasa a la siguiente arista. En caso contrario:

   b. Situar el origen de coordenadas en su punto medio, llamar *p* al extremo que está en el semiplano superior y *q* al otro y comprobar las direcciones del conjunto O (teniendo solo en cuenta las direcciones de O entre 0º y 180º) entre las que se encuentra la dirección, *d*, de la arista. Estas direcciones serán su $tope^+$ y su $tope^-$.

   c. Sustituir el segmento *pq* por el paralelepípedo con vértices *p* y *q* y con direcciones $d^+=d$+precisión, $d^-=d$-precisión, donde "precisión" es la distancia entre dos ángulos (para barrer todos los posibles ángulos "precisión" debería ser un valor infinitamente pequeño).
      Observación: desde *q* se usarán las direcciones con el sentido original (entre 0º y 180º), mientras que desde *p* se usarán con el sentido contrario.

   d. Sustituir el paralelepípedo anterior por el que se obtiene al hacer $d^+=d^+$+precisión, $d^-=d^-$-precisión. Si en algún paso d^+ alcanza $tope^+$ o d^- alcanza $tope^-$, en los pasos posteriores se mantiene ese valor (sin sumar o restar "precisión"). Se repite el paso d hasta que ambas direcciones alcanzan su tope.

4) Una vez obtenidos todos los paralelepípedos se procederá a realizar la siguiente operación: para cada dos paralelepípedos se realiza su intersección con el objetivo de añadir a cada uno de los paralelepípedos los puntos en que ambos polígonos se intersecan como vértices extras. Dichos vértices no cambiarán la forma de los paralelepípedos pero proporcionan una información de gran importancia para obtener el resultado.

5) Para cada paralelepípedo considerar cada uno de los segmentos que lo forman (teniendo en cuenta los vértices extra añadidos en el paso 4). Si los dos extremos de un segmento están dentro de la envolvente convexa original, entonces dicho segmento pertenece a la envolvente O-convexa.

Del paso quinto se ha obtenido la siguiente observación: *la envolvente O-convexa está contenida dentro del área de la envolvente convexa en el caso habitual (sólo para el caso de envolventes O-convexas sobre tres puntos).*

A continuación se muestra una secuencia de ilustraciones para un triángulo usando direcciones 0º y 90º:

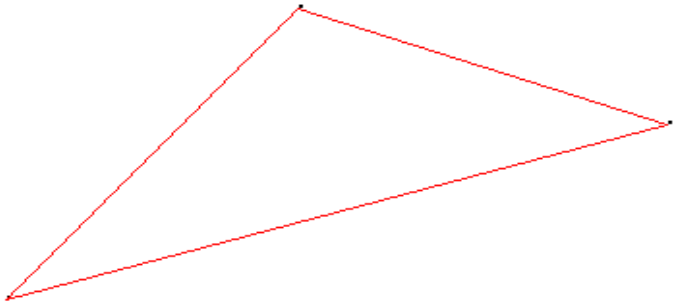


Imagen 4.15. Triángulo obtenido en el paso 2 del algoritmo.

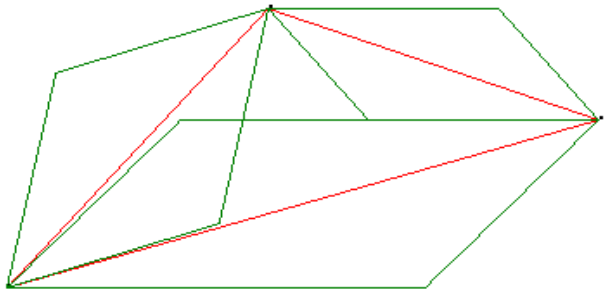


Imagen 4.16. En color verde oscuro se pueden apreciar los paralelepípedos en proceso de inflado para cada una de las aristas.

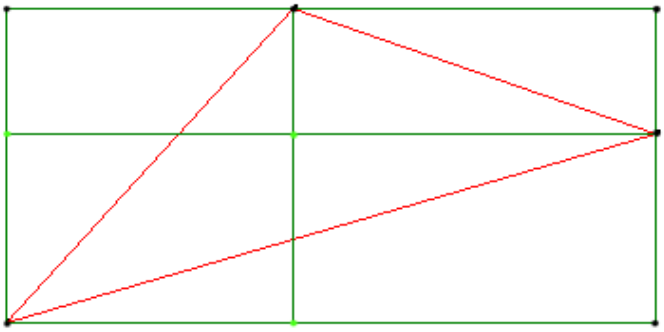


Imagen 4.17. En color verde oscuro se pueden ver los paralelepípedos totalmente inflados. Los puntos de color verde claro son aquellos que se añadirán a los

paralelepípedos correspondientes a la arista izquierda y a la arista inferior (debido a que los paralelepípedos correspondientes se intersecan en esos puntos) y los puntos negros son los puntos originales de los paralelepípedos.

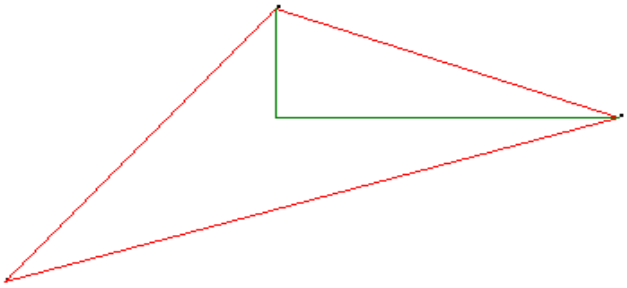


Imagen 4.18. Envolvente {0, 90}-convexa del triángulo. Se puede apreciar que de todos los segmentos de la figura 4.8, los únicos que se encuentran en su totalidad dentro de la envolvente convexa en el caso habitual, son aquellos que forman parte del resultado.

Un pseudocódigo del algoritmo de inflado de paralelepípedos (para triángulos) es el siguiente:

```
FUNCIÓN CH_Inflado(conjunto de direcciones: O, conjunto de puntos: P)
        resultado: Lista de segmentos
        //Calcular la envolvente convexa de P
        CH: Polígono
        CH = EnvolventeConvexa(P)
        lista_paralelepipedos: Lista Polígonos
        PARA CADA arista EN CH HACER
        Dir_arista: dirección
        Dir_arista = ObtenerDireccion(arista)
        SI Dir_arista NO PERTENECE A O ENTONCES
```

```
        tope_mas, tope_menos, d_mas, d_menos: dirección
        poligono_inf: Polígono
        tope_mas = ObtenerTopeMas(O, Dir_arista)
        tope_menos = ObtenerTopeMenos(O, Dir_arista)
        HACER
        d_mas = MIN(d_mas + PRECISIÓN, tope_mas)
        d_menos = MAX(d_menos – PRECISIÓN, tome_menos)
        poligono_inf = CrearParalelepipedo(arista, d_mas, d_menos)
        MIENTRAS (d_mas ≠ tope_mas) Y (d_menos ≠ tope_menos)
        Añadir(lista_paralelepipedos, polígono_inf);
        SI NO
        Añadir(resultado, arista)
        FIN SI
        FIN PARA CADA
        //Se añaden a los paralelepípedos correspondientes los puntos extra de la intersección de polígonos
        AñadirPuntosInterseccion(lista_paralelepipedos)
        PARA CADA paralelepipedo_extra EN lista_paralelepipedos HACER
        PARA CADA segmento EN paralelepípedo_extra HACER
        SI segmento PERTENECE A CH ENTONCES
        Añadir(resultado, segmento)
        FIN SI
        DEVOLVER resultado
```

```
FIN FUNCIÓN
```

4.1.5.Conclusión de la sección

En la sección cuatro se ha llevado a cabo un estudio de las diferentes definiciones, propuestas en la literatura, de envolvente convexa con direcciones restringidas. Estudiando las propiedades que ofrece cada definición se ha seleccionado una de ellas. Con un concepto claro de envolvente convexa con direcciones restringidas se ha llevado a cabo un estudio relativo a los triángulos, un caso particular de envolvente convexa. Además se han presentado dos algoritmos para el cálculo de envolventes convexas y soluciones para algunos sub-problemas interesantes derivados de dichos algoritmos.

Por todo ello se puede concluir que se han cumplido los objetivos que van del segundo al cuarto.

A continuación se procederá al desarrollo de un programa que aplique todos los conocimientos adquiridos hasta este punto.

5. Desarrollo de una aplicación para la visualización de envolventes convexas con direcciones restringidas

En esta sección se muestra la documentación técnica correspondiente al desarrollo de la aplicación que visualiza envolventes convexas y triangulaciones con direcciones restringidas. En esta aplicación se han implementado los algoritmos descritos en las secciones anteriores de tal manera que se puedan llevar a cabo ejemplos prácticos de todo lo explicado anteriormente.

Para llevar a cabo este producto software se ha utilizado una metodología Métrica v3 pero adaptada a las necesidades particulares de un desarrollo unipersonal. Para adaptar la metodología se han eliminado las fases innecesarias. Al ser la metodología utilizada una adaptación de Métrica v3 el ciclo de vida es un ciclo de vida en cascada.

5.1. Definición del sistema

El sistema a desarrollar servirá para la visualización de envolventes convexas y triangulaciones tanto con direcciones restringidas como sin ellas.

Para el desarrollo del sistema se ha elegido un enfoque orientado a objetos debido a que se aprecia una fuerte ligadura entre datos y operaciones sobre los mismos.

Además al ser orientado a objetos se podrán utilizar librerías matemáticas y gráficas orientadas a objetos que facilitarán el desarrollo del sistema. Por otro lado con ello se facilita el mantenimiento del sistema, al estar localizados los posibles cambios en los módulos desarrollados para la construcción del mismo.

5.1.1.Descripción y planteamiento del problema

Se omitirá esta fase de la definición del sistema ya que la descripción y el planteamiento del problema quedan reflejados en la introducción del presente documento

5.1.2.Descripción general del entorno tecnológico

El trabajo se enmarca dentro de la realización de un proyecto de fin de carrera de la titulación Ingeniería Informática de la Universidad de Alcalá fruto de la investigación acerca de las envolventes convexas con direcciones restringidas.

El sistema se organiza como una aplicación de ventana *standalone* en el cual se podrán introducir datos de entrada (conjuntos de puntos y direcciones) para obtener envolventes convexas o triangulaciones.

La aplicación será desarrollada utilizando la tecnología .NET debido a las facilidades que ofrece para la creación de aplicaciones de escritorio. Además el lenguaje seleccionado de entre todos los que soporta .NET es C++ debido a su eficiencia en problemas de geometría computacional y a que es el lenguaje en el que está escrita la librería matemática CGAL necesaria para el proyecto.

Los ordenadores en los que se ejecuta la aplicación deben tener instalado un sistema operativo Windows con el .NET Framework.

5.1.3.Identificación de usuarios

En el uso de la aplicación podemos identificar un único tipo de usuario: usuario con conocimientos del área de aplicación del Sistema de Información. La aplicación sólo debería ser usada por usuarios con un cierto nivel de conocimiento de Geometría Computacional ya que de lo contrario no se le podrá sacar todo el partido posible al sistema.

Además se recomienda que estos usuarios usen la aplicación con fines didácticos en el área de la Geometría Computacional.

En las sesiones de obtención de requisitos participarán el desarrollador del sistema y el tutor del TFC.

5.2. Catálogo de requisitos

En esta sección se recogen los requisitos del Sistema de Información clasificados según su tipo.

5.2.1.Requisitos funcionales

FUNC1 Visualización de envolventes convexas por el método de la intersección de semiplanos escalera. Permite visualizar envolventes convexas con direcciones restringidas o envolventes convexas en el caso general.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Alta

FUNC1.1 Selección de puntos por fichero o ratón. Permite elegir los puntos que constituyen la entrada del programa o bien mediante un fichero de entrada o bien haciendo click con el ratón.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Baja

FUNC1.1.1 Cargar puntos desde fichero Permite cargar una distribución de puntos desde fichero.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

FUNC1.1.2 Guardar puntos a fichero Permite guardar una distribución de puntos a fichero.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

FUNC1.2 Realización de envolvente convexa (mediante el método de *las escaleras*). Dado un conjunto de direcciones y puntos de entrada calcula la envolvente convexa con o sin direcciones restringidas, si es posible, usando el método de la intersección de semiplanos escalera (método de *las escaleras*). Hay casos en los que no se realizará el proceso completo, en cuyo caso se hará saber al usuario. Los casos que no tengan éxito mostrarán un mensaje de error.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Muy alta

FUNC1.3 Modo de visualización paso a paso Permite ver el resultado correspondiente a una entrada paso a paso. Se ofrece la posibilidad de ir hacia adelante y hacia atrás en cualquier momento.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

FUNC2 Conmutación entre visualización de envolventes convexas y triangulaciones (solo en método de *las escaleras*). Permite conmutar entre envolventes convexas y triangulaciones guardando la información de cada módulo del programa y pudiendo volver de uno a otro en cualquier momento.

Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

FUNC3 Visualización de triangulaciones (sólo en método de *las escaleras*). Permite visualizar triangulaciones tanto en el caso general como para direcciones restringidas.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Alta

FUNC3.1 Carga de triangulaciones desde fichero Permite cargar triangulaciones desde fichero.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Baja

FUNC3.2 Introducción de puntos a través de la interfaz y generación de la triangulación de Delaunay Permite al usuario introducir puntos en el programa a través del ratón y calcular la triangulación de Delaunay de estos puntos.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

FUNC3.3 Guardar triangulaciones de Delaunay Permite guardar en fichero la triangulación generada.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Baja

FUNC4 Visualización de envolventes convexas por el método del inflado. Permite visualizar triangulaciones con direcciones restringidas, así como una animación de su construcción.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Alta

FUNC4.1 Selección de puntos por fichero o ratón. Permite elegir los puntos que constituyen la entrada del programa o bien mediante un fichero de entrada o bien haciendo click con el ratón.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Baja

FUNC4.1.1 Cargar puntos desde fichero Permite cargar una distribución de puntos desde fichero.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

FUNC4.1.2 Guardar puntos a fichero Permite guardar una distribución de puntos a fichero.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

FUNC4.2 Realización de la envolvente convexa (mediante el método de inflado). Dado un conjunto de direcciones y puntos de entrada calcula la triangulación de Delaunay sin direcciones restringidas. Para cada triángulo se debe calcular la envolvente convexa con direcciones restringidas usando el método del inflado.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Alta

FUNC4.3 Modo de visualización con animación		
Permite ver el resultado correspondiente a una entrada como si fuera un *video*. Se ofrece la posibilidad de ir hacia adelante y hacia atrás en cualquier momento. También se ofrece la posibilidad de parar la animación y reanudarla desde ese punto		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Alta

FUNC5 Selección de direcciones.		
Permite seleccionar las direcciones que formarán la envolvente.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

FUNC5.1 Añadir direcciones.		
Permite añadir una dirección o un rango de ellas.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

FUNC5.2 Quitar direcciones.		
Permite quitar una dirección.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Baja

FUNC5.3 Reiniciar direcciones.		
Permite quitar todas las direcciones seleccionadas.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Muy baja

FUNC6 Selección de los modos de envolvente (sólo en método de *las escaleras*)		
Se permitirá seleccionar entre tres modos: ortogonal, direcciones seleccionadas y todas las direcciones.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Baja

FUNC7 Limpiar pantalla		
Permite limpiar el área de visualización para introducir una entrada nueva.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

FUNC8 Guardar área de visualización como imagen		
Permite guardar el área de visualización en una imagen.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

FUNC9 Preferencias de visualización		
Permite establecer las preferencias en cuanto a los colores de visualización de cada uno de los elementos que se podrán observar en el área de visualización.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Baja

5.2.2.Requisitos de datos

DAT1 Punto		
Contendrá los datos de un punto.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

DAT2 Dirección		
Contendrá los datos de una dirección.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

DAT3 Lista de direcciones		
Conjunto de direcciones.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Alta

DAT4 Lista de puntos		
Lista de puntos que puede representar o bien un segmento o bien un conjunto de ellos.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Alta

DAT5 Fichero de puntos		
Fichero que servirá para cargar puntos desde fichero. Estos ficheros no deben contener puntos alineados. El fichero deberá tener el siguiente formato: coordenada_x#coordenada_y#radio		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

DAT6 Fichero de triangulación		
Fichero que servirá para cargar triangulaciones desde fichero. Estos ficheros no deben contener puntos alineados. El fichero deberá tener el siguiente formato: coordenada_x1#coordenada_y1# coordenada_x2#coordenada_y2# coordenada_x3#coordenada_y3		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

DAT7 Estado en el proceso de inflado		
Elemento de datos que representa un paso cualquiera dentro del proceso de inflado. Debe contener información de los paralelepípedos resultado de un paso de inflado en cada una de las aristas que compone el polígono.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

DAT8 Lista de estados del proceso de inflado		
Elemento de datos que representa un conjunto con todos los pasos de un inflado.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

5.2.3.Requisitos de interfaz

I1 Interfaz de visualización Pantalla donde los usuarios podrán visualizar las envolventes convexas y las triangulaciones. La interfaz usada para los dos algoritmos de visualización debe ser equivalente.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

I2 Control selector de direcciones Control que permitirá a los usuarios seleccionar las direcciones que se usarán al hacer la envolvente convexa. Debe ser un elemento usable y con un cuidado diseño.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

I3 Formato visual Todas las pantallas de la aplicación seguirán el mismo formato visual que comprende colores, fuentes y organización de los elementos.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

I3.1: Colores agradables Para el diseño de las pantallas se utilizarán colores agradables en tonos claros. Además los colores deben ser configurables.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

I4: Teclas de función El uso del teclado permite accesos directos a funciones mediante teclas o combinaciones especiales de las mismas.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Media

5.2.4.Requisitos de seguridad

No se han detectado requisitos de seguridad en esta aplicación.

5.2.5.Requisitos de eficiencia

E1: Tiempos de respuesta aceptables Al tratarse de una aplicación de visualización deben conseguirse tiempos de respuesta apropiados. Se aplicarán en la medida de lo posible técnicas apropiadas de programación. Asimismo se aplicarán todas las optimizaciones de compilación posibles.		
Versión: 1.0	**Autor:** José Daniel Expósito	**Dificultad:** Alta

5.3. Análisis orientado a objetos

5.3.1.Modelo de casos de uso

5.3.1.1. Caso de uso del sistema

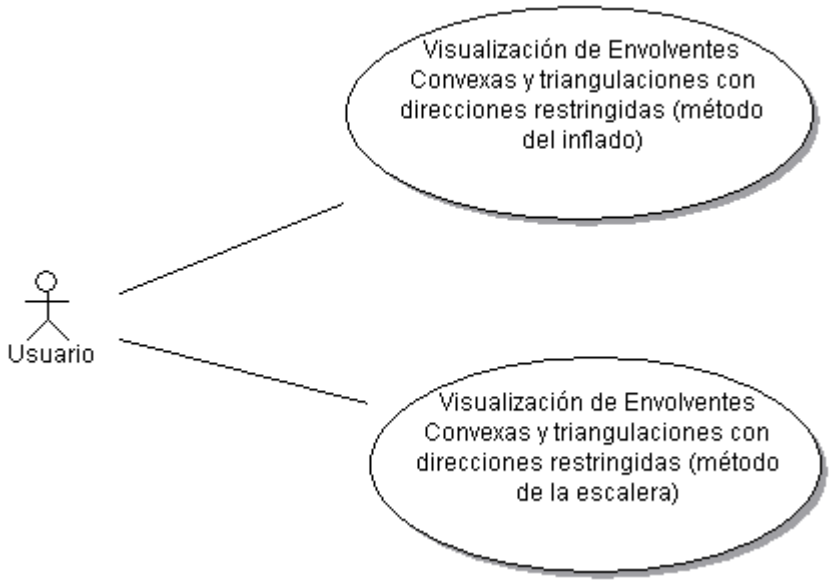


Diagrama 5.1. Caso de uso del sistema

5.3.1.1.1. Actores

- **Usuario:** el usuario será una persona del perfil descrito en el apartado *Identificación de usuarios*. Este usuario tendrá acceso a toda la funcionalidad de la aplicación.

5.3.1.1.2. Casos de uso

- Visualización de Envolventes Convexas y triangulaciones con direcciones restringidas (método de la escalera): este caso de uso recoge toda la funcionalidad de visualización relacionada con el algoritmo de intersección de semiplanos escalera. Entre esta funcionalidad destaca la posibilidad de alternar entre hacer la operación para una envolvente convexa o triangular la superficie y aplicar la operación. En ambos casos se puede seleccionar si se hará la envolvente con todas las direcciones, con las ortogonales o con las seleccionadas por el usuario.

- Visualización de Envolventes Convexas y triangulaciones con direcciones restringidas (método del inflado): este caso de uso recoge

toda la funcionalidad de visualización relacionada con el algoritmo de inflado de paralelepípedos. Permite seleccionar un conjunto de puntos a triangular y un conjunto de direcciones con los que se llevará a cabo el proceso de inflado.

5.3.1.1.3. Relaciones

El actor Usuario se relaciona con los dos casos de uso del sistema ya que tendrá acceso a toda la funcionalidad ofrecida por ambos casos de uso.

5.3.1.2. Caso de uso Visualización de Envolventes Convexas y triangulaciones con direcciones restringidas (método de la escalera)

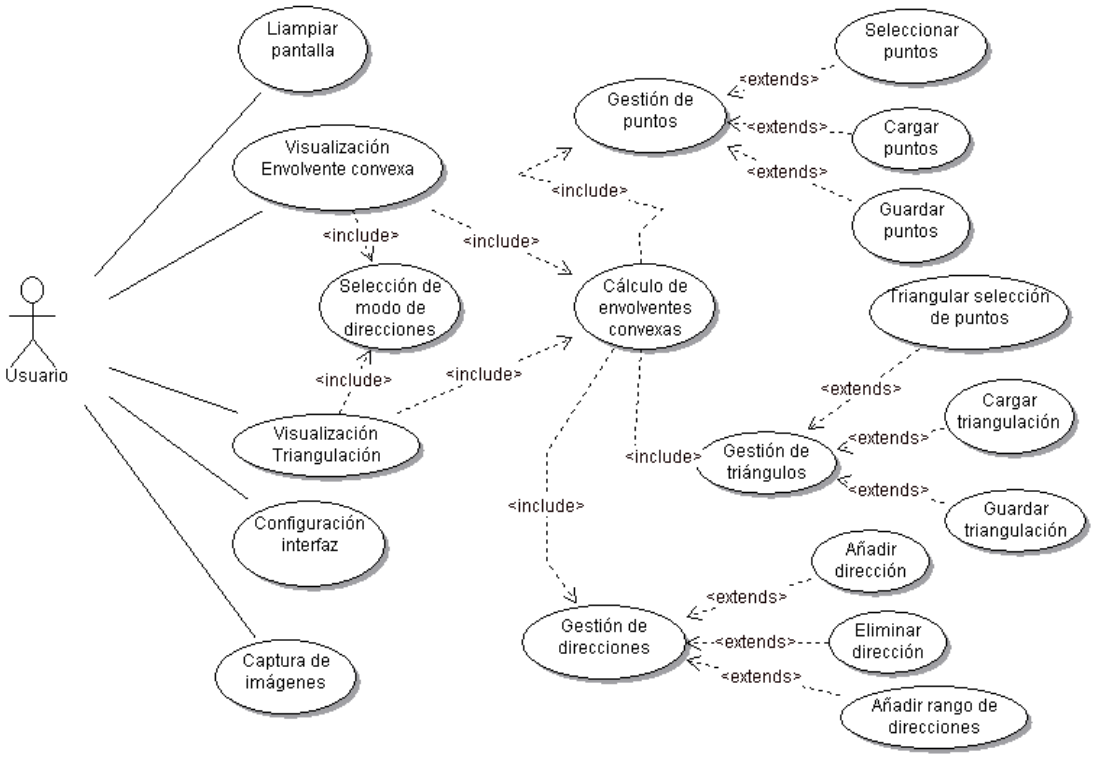


Diagrama 5.2. Caso de uso de Visualización de Envolventes Convexas y triangulaciones con direcciones restringidas (método de la escalera).

5.3.1.2.1. Actores

- **Usuario:** el usuario tendrá acceso a toda la funcionalidad ofrecida por los casos de uso en que se descompone este caso de uso. Entre dicha funcionalidad se encuentra: visualizar envolventes convexas con o sin direcciones restringidas usando un conjunto de puntos introducido desde la interfaz o cargado desde fichero y usando las direcciones que correspondan al modo elegido por el usuario (todas, ortogonales y/o seleccionadas por el usuario), visualizar triangulaciones con o sin direcciones restringidas usando un conjunto de triángulos introducido desde la interfaz o cargado desde fichero y usando las direcciones que correspondan al modo elegido por el usuario (todas, ortogonales y/o seleccionadas por el usuario), limpiar la pantalla para realizar nuevas operaciones, configurar la interfaz y realizar capturas del trabajo realizado con la aplicación.

5.3.1.2.2. Casos de uso

- **Limpiar pantalla:** permite a un usuario limpiar el área de dibujo de la aplicación para realizar una nueva operación.

- **Configuración interfaz:** permite a un usuario cambiar los colores de los elementos de la interfaz para elegir una combinación a su gusto.

- **Captura de imágenes:** permite a un usuario hacer una captura del área de dibujo.

- **Visualización Envolvente convexa:** permite a un usuario visualizar la envolvente convexa de un conjunto de puntos dados, con los modos seleccionados y las direcciones indicadas.

- **Visualización Triangulación:** permite a un usuario visualizar una triangulación dada con las direcciones especificadas.

- **Selección de modo de direcciones:** permite al usuario elegir entre tres modos para el conjunto de direcciones que actúa como entrada del algoritmo: modo todas direcciones (se usarán todas las direcciones), modo ortogonal (se usan las direcciones 0^o y 90^o) y modo direcciones seleccionadas (se usan las direcciones especificadas por el usuario).

- **Cálculo de envolventes convexas:** permite realizar el cálculo de la envolvente convexa dado un modo de operación, un conjunto de puntos y un conjunto de direcciones.

- **Gestión de puntos:** lleva a cabo la gestión de los puntos que actúan como entrada del algoritmo.
- **Seleccionar puntos:** permite al usuario seleccionar los puntos que van a formar parte de la entrada del algoritmo manualmente.
- **Cargar puntos:** permite al usuario cargar desde fichero los puntos que van a formar parte de la entrada del algoritmo.
- **Guardar puntos:** permite al usuario guardar a fichero los puntos introducidos.
- **Gestión de triángulos:** lleva a cabo la gestión y el cálculo de los triángulos que forman la entrada del algoritmo.
- **Triangular selección de puntos:** permite a un usuario calcular la triangulación de Delaunay de un conjunto de puntos dado. Dicha triangulación pasa a ser la entrada del algoritmo.
- **Cargar triangulación:** realiza la carga de una triangulación desde fichero.
- **Guardar triangulación:** guarda una triangulación en un fichero.
- **Gestión de direcciones:** lleva a cabo la gestión de las direcciones seleccionadas por el usuario. Le permite al mismo añadir y quitar direcciones individuales, añadir rangos de direcciones y reiniciar el conjunto de direcciones.
- **Añadir dirección:** permite al usuario añadir una dirección al conjunto. Esto se puede hacer tanto gráficamente como numéricamente.
- **Eliminar dirección:** permite al usuario eliminar una dirección del conjunto. Esto se puede hacer tanto gráficamente como numéricamente.
- **Añadir rango de direcciones:** permite al usuario añadir direcciones de cinco en cinco grados entre unos valores máximo y mínimo especificados.

5.3.1.2.3. Relaciones

El actor Usuario se relaciona con los casos de uso Limpiar pantalla, Visualización Envolvente Convexa, Visualización Triangulación, Configuración

interfaz y Captura de imágenes permitiéndole al mismo realizar la funcionalidad descrita para dichos casos de uso.

El caso de uso Visualización Envolvente Convexa y el caso de uso Visualización Triangulación incluyen al caso de uso Selección de modo de direcciones, ya que para realizar cualquiera de las funcionalidades que engloban los dos primeros es obligatorio hacer uso de la funcionalidad ofrecida por el tercero.

Además los casos de uso Visualización Envolvente Convexa y Visualización Triangulación incluyen al caso de uso Cálculo de envolventes convexas ya que para visualizar tanto las envolventes como las triangulaciones es necesario llevar a cabo su cálculo.

El caso de uso Cálculo de envolventes convexas incluye a los casos de uso Gestión de puntos, Gestión de triángulos y Gestión de direcciones ya que para llevar a cabo los cálculos es obligatorio haber realizado la funcionalidad descrita en Gestión de direcciones y haber realizado la funcionalidad descrita por Gestión de puntos o la funcionalidad descrita por Gestión de triángulos.

Gestión de puntos extiende de Seleccionar puntos, Cargar puntos y Guardar puntos debido a que al realizar la funcionalidad del mismo se puede realizar cualquiera de los tres casos de uso de los que extiende.

Gestión de triángulos extiende de Triangular selección puntos, Cargar triangulación y Guardar triangulación debido a que al realizar la funcionalidad del mismo se puede realizar cualquiera de los tres casos de uso de los que extiende.

Gestión de direcciones extiende de Añadir dirección, Eliminar dirección y Añadir rango de direcciones debido a que al realizar la funcionalidad del mismo se puede realizar cualquiera de los tres casos de uso de los que extiende.

5.3.1.3. Caso de uso Visualización de Envolventes Convexas y triangulaciones con direcciones restringidas (método del inflado)

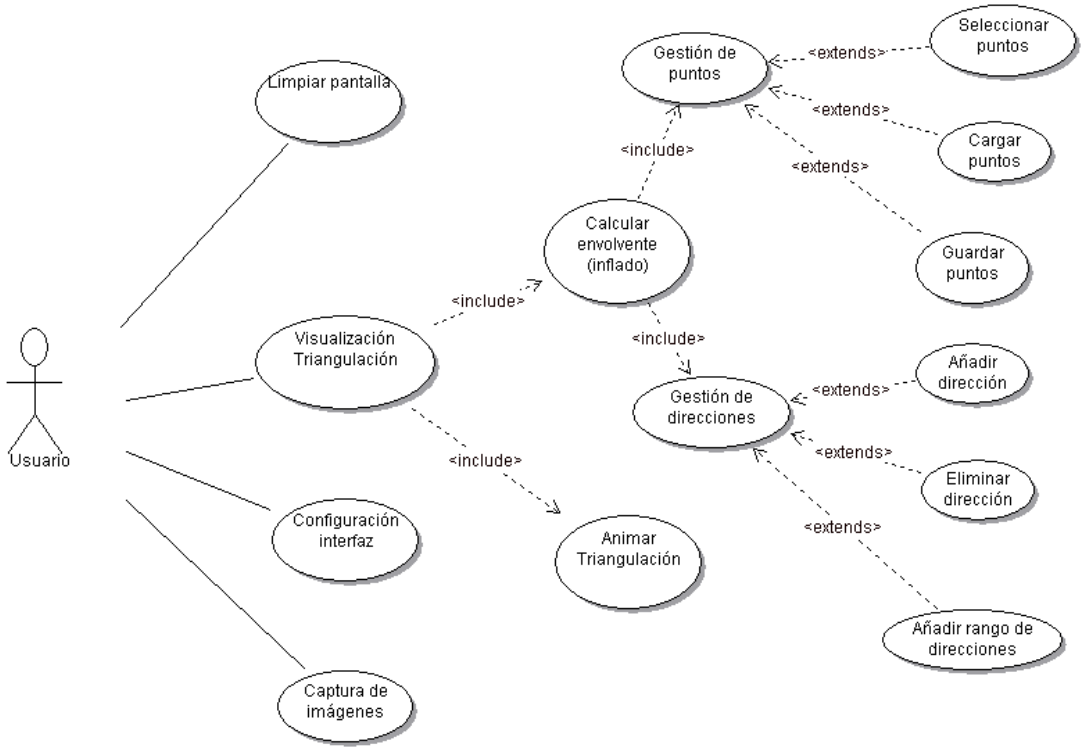


Diagrama 5.3. Caso de uso de Visualización de Envolventes Convexas y triangulaciones con direcciones restringidas (método del inflado).

5.3.1.3.1. Actores

- **Usuario:** el usuario tendrá acceso a toda la funcionalidad ofrecida por los casos de uso en que se descompone este caso de uso. Entre dicha funcionalidad se encuentra: visualizar triangulaciones sin direcciones restringidas (siempre se muestra la triangulación sin direcciones restringidas) y con direcciones restringidas (el resultado correspondiente a la triangulación con direcciones restringidas se visualizará mediante una animación) usando un conjunto de puntos introducido desde la interfaz o cargado desde fichero y usando las direcciones seleccionadas por el usuario, limpiar la pantalla para realizar nuevas operaciones, configurar la interfaz y realizar capturas del trabajo realizado con la aplicación.

5.3.1.3.2. Casos de uso

- **Limpiar pantalla:** permite a un usuario limpiar el área de dibujo de la aplicación para realizar una nueva operación.
- **Configuración interfaz:** permite a un usuario cambiar los colores de los elementos de la interfaz para elegir una combinación a su gusto.
- **Captura de imágenes:** permite a un usuario hacer una captura del área de dibujo.
- **Visualización Triangulación:** permite a un usuario visualizar una triangulación dada con las direcciones especificadas. La triangulación para todas las direcciones se muestra siempre.
- **Calcular envolvente (inflado):** permite realizar el cálculo de la envolvente convexa dado un conjunto de puntos y un conjunto de direcciones mediante el método del inflado. Este caso de uso además prepara los datos necesarios para la visualización animada de la envolvente
- **Gestión de puntos:** lleva a cabo la gestión de los puntos que actúan como entrada del algoritmo.
- **Seleccionar puntos:** permite al usuario seleccionar los puntos que van a formar parte de la entrada del algoritmo manualmente.
- **Cargar puntos:** permite al usuario cargar desde fichero los puntos que van a formar parte de la entrada del algoritmo.
- **Guardar puntos:** permite al usuario guardar a fichero los puntos introducidos.
- **Gestión de direcciones:** lleva a cabo la gestión de las direcciones seleccionadas por el usuario. Le permite al mismo añadir y quitar direcciones individuales, añadir rangos de direcciones y reiniciar el conjunto de direcciones.
- **Añadir dirección:** permite al usuario añadir una dirección al conjunto. Esto se puede hacer tanto gráficamente como numéricamente.
- **Eliminar dirección:** permite al usuario eliminar una dirección del conjunto. Esto se puede hacer tanto gráficamente como numéricamente.

- **Añadir rango de direcciones:** permite al usuario añadir direcciones de cinco en cinco grados entre unos valores máximo y mínimo especificados.

- **Animar Triangulación:** permite al usuario iniciar y parar la animación correspondiente al inflado de la triangulación, dando como resultado la triangulación con direcciones restringidas. Además cuando la animación está pausada se pueden avanzar o retroceder pasos.

5.3.1.3.3. Relaciones

El actor Usuario se relaciona con los casos de uso Limpiar pantalla, Visualización Triangulación, Configuración interfaz y Captura de imágenes permitiéndole al mismo realizar la funcionalidad descrita para dichos casos de uso.

El caso de uso Visualización Triangulación incluye al caso de uso Calcular envolvente (inflado) ya que para visualizar las triangulaciones es necesario su cálculo.

El caso de uso Calcular envolvente (inflado) convexa incluye a los casos de uso Gestión de puntos y Gestión de direcciones ya que para llevar a cabo los cálculos es obligatorio haber realizado la funcionalidad descrita en Gestión de direcciones y haber realizado la funcionalidad descrita por Gestión de puntos.

Gestión de puntos extiende de Seleccionar puntos, Cargar puntos y Guardar puntos debido a que al realizar la funcionalidad del mismo se puede realizar cualquiera de los tres casos de uso de los que extiende.

Gestión de direcciones extiende de Añadir dirección, Eliminar dirección y Añadir rango de direcciones debido a que al realizar la funcionalidad del mismo se puede realizar cualquiera de los tres casos de uso de los que extiende.

Además el caso de uso Visualización Triangulación incluye al caso de uso Animar Triangulación ya que para visualizar la triangulación con direcciones restringidas de la manera especificada en los requisitos es necesario llevar a cabo la funcionalidad descrita por Animar Triangulación.

5.3.2.Modelo de clases de Análisis

En este apartado se llevará a cabo el análisis de las clases que se estiman necesarias una vez realizado el análisis de casos de uso.

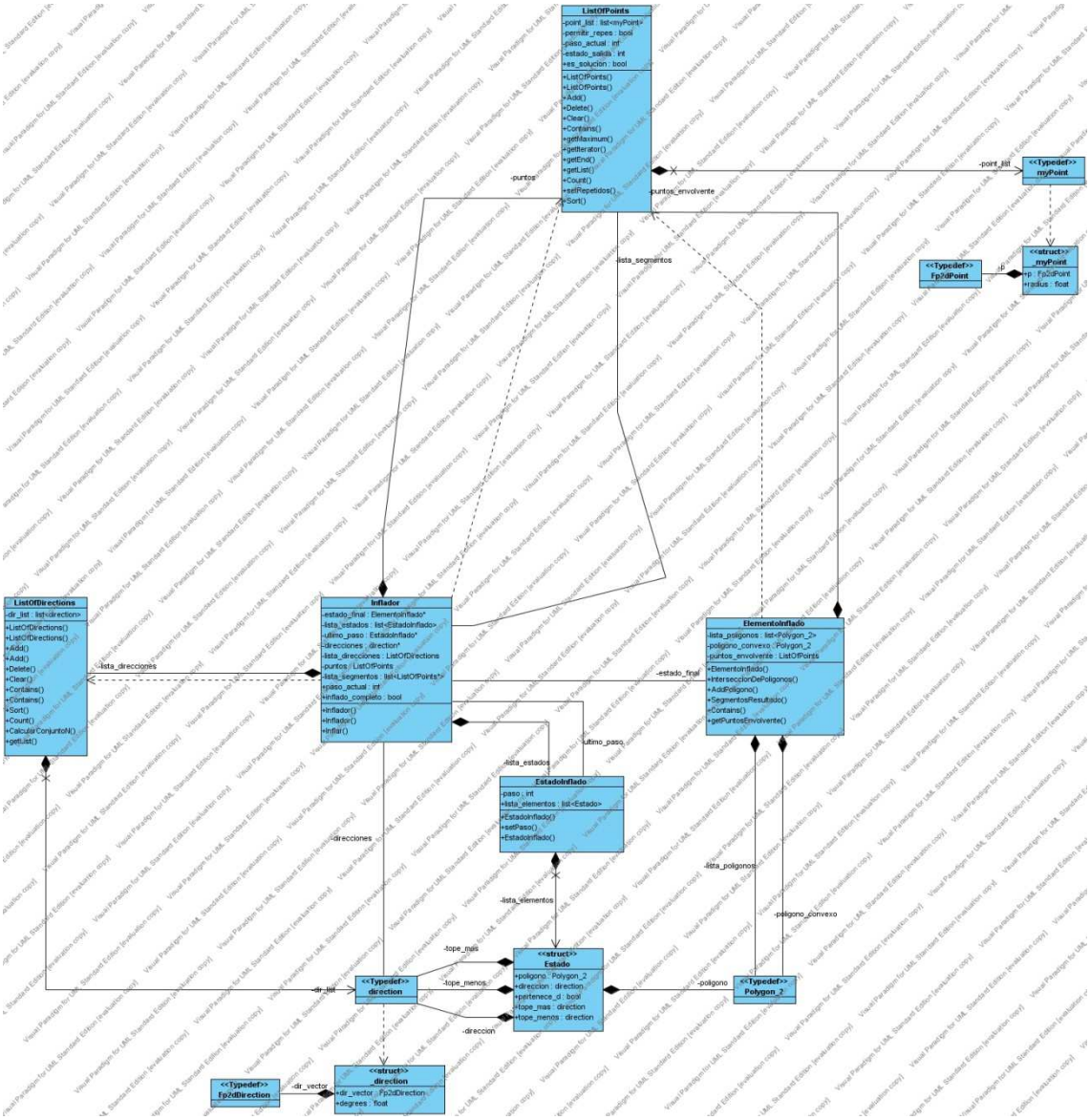


Diagrama 5.4. Diagrama de clases de análisis.

El diagrama de clases de análisis cuenta con 5 clases y diversas definiciones de tipos y estructuras. Con ello se está adelantando trabajo de la fase de diseño, lo cual se debe a que sólo con el análisis se ve que algunos elementos de datos no tienen entidad suficiente para constituir una clase dando lugar a estructuras. Las relaciones entre los elementos del diagrama se detallan a continuación.

Entre la estructura *myPoint* y el tipo *Fp2dPoint* existe una relación de asociación porque para cada instancia de *myPoint* existirá una de *Fp2dPoint.* A su vez existe una relación de asociación entre *myPoint* y la clase *ListOfPoints*. Esto se debe a que para cada instancia de *ListOfPoints* existirá cero, una o varias de *myPoint*.

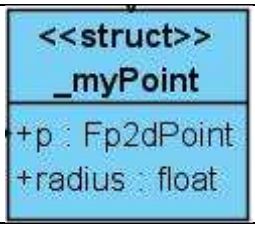
Entre la estructura *direction* y el tipo *Fp2dDirection* existe una relación de asociación ya que para cada instancia de *direction* existe una de *Fp2dDirection*. A su vez existe una relación de asociación entre *direction* y la clase *ListOfDirections*. Esto se debe a que para cada instancia de *ListOfDirections* existirá cero, una o varias de *direction*.

Entre la estructura *Estado* y los tipos *Polygon_*2 y *direction* existen sendas relaciones de asociación ya que para cada instancia de *Estado* existe una instancia de *Polygon_2* y de *direction*.

Entre la clase *EstadoInflado* y la estructura *Estado* existe una relación de asociación ya que para cada instancia de *EstadoInflado* existirán cero, una o más instancias de *Estado*.

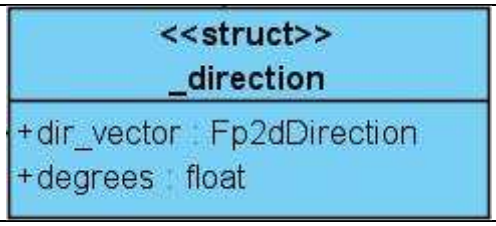
Entre la clase *ElementoInflado* y el tipo *Polygon_2* existe una relación de asociación ya que para cada instancia de *ElementoInflado* hay cero, una o más instancias de *Polygon_2*. Además *ElementoInflado* se relaciona con *ListOfPoints* mediante una relación de asociación ya que para cada *ElementoInflado* existe una instancia de *ListOfPoints*. Finalmente existe una relación de dependencia entre *ElementoInflado* y *ListOfPoints* ya que la primera no puede existir sin la segunda.

Entre la clase *Inflador* y *ListOfDirections* existe una relación de asociación ya que para cada *Inflador* existe un *ListOfDirections*. Además existe una relación de asociación entre *Inflador* y *ListOfPoints* debido a que para cada instancia del primero existe una del segundo. *Inflador* también se relaciona con *EstadoInflado* mediante asociación ya que para cada *Inflador* existen cero, uno o más elementos *EstadoInflado*. Por otra parte *Inflador* se relaciona con *ElementoInflado* mediante asociación ya que para cada instancia del primero hay una del segundo. Finalmente decir que existen relaciones de dependencia entre *Inflador* y *ListOfPoints* y *ListOfDirections* ya que *Inflador* no puede existir sin ninguna de dichas dos clases.

ESTRUCTURA: myPoint	
	Estructura que representa un punto mediante sus coordenadas y un radio
ATRIBUTOS	
• **p:** punto propiamente dicho representado por sus coordenadas. • **radius:** radio del punto.	

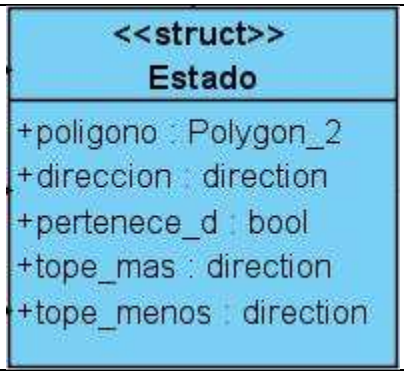
CLASE: ListOfPoints	
	Esta clase representa una lista de puntos.
ATRIBUTOS	
• **point_list:** representación interna de la lista de puntos. • **permitir_repes:** determina si se pueden introducir puntos repetidos o no. • **paso_actual:** determina en que punto de la lista se encuentra. • **estado_salida:** determina el estado de salida de ciertas operaciones con la lista. • **es_solucion:** dice si un conjunto de puntos forma solución de la	

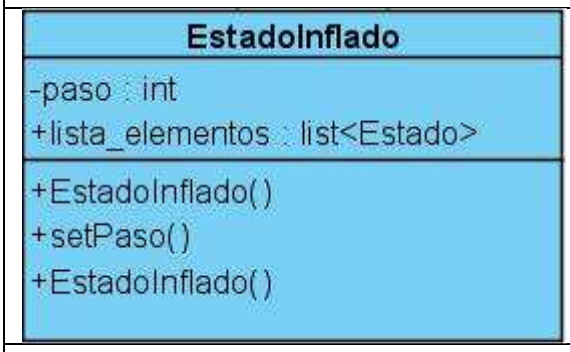
envolvente de un conjunto de puntos.
MÉTODOS
Los métodos de esta clase no son más que las operaciones básicas sobre una lista: añadir puntos, quitarlos, borrar la lista, ordenarla, contar el número de elementos, comprobar si contiene un punto… Además de la funcionalidad típica hay que proporcionar métodos de acceso a la lista a bajo nivel.

ESTRUCTURA: direction	
	Esta estructura representa una dirección mediante su vector director y los grados que forma con el eje X.
ATRIBUTOS	
• **dir_vector:** vector director de la dirección. • **degrees:** grados que forma la dirección con el eje X.	

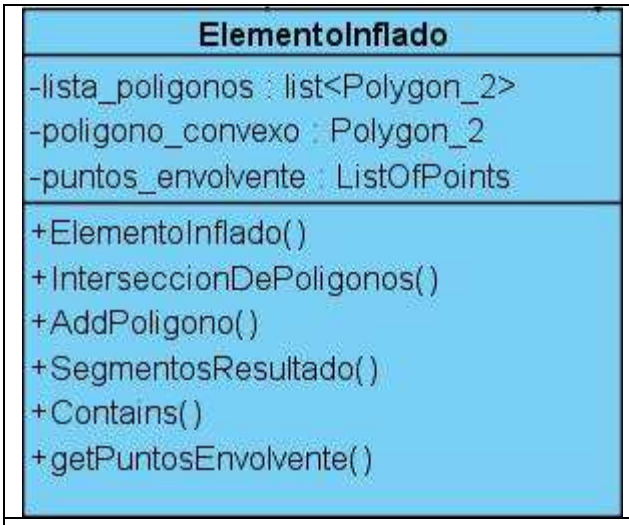
CLASE: ListOfDirections	
	Esta clase representa una lista de direcciones.
ATRIBUTOS	
• **dir_list:** representación interna de la lista de direcciones.	
MÉTODOS	
Los métodos de esta clase no son más que las operaciones básicas sobre una lista: añadir direcciones, quitarlas, borrar la lista, ordenarla, contar el número de elementos, comprobar si contiene un punto… Además es necesario que a partir	

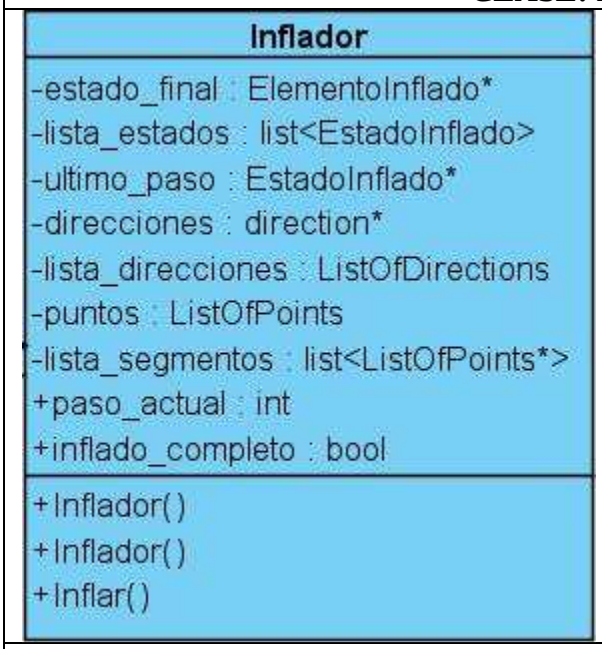
de una lista de direcciones se obtenga el conjunto de direcciones normales a las mismas (conjunto N). Aparte de la funcionalidad típica hay que proporcionar métodos de acceso a la lista a bajo nivel.

ESTRUCTURA: Estado	
	Esta estructura representa un paso en el inflado de la arista de un polígono.
ATRIBUTOS	
• **poligono:** polígono a inflar. • **direccion:** dirección de la arista en proceso de inflado. • **pertenece_d:** determina si la dirección pertenece al conjunto de direcciones de entrada. • **tope_mas:** tope positivo de inflado. • **tope_menos:** tope negativo de inflado.	

CLASE: EstadoInflado	
	Clase que representa un paso en el inflado de un polígono.
ATRIBUTOS	
• **paso:** número de paso que representa. • **lista_elementos:** lista que contiene el estado de inflado de cada arista del polígono.	
MÉTODOS	
La funcionalidad que ofrece esta clase se limita a almacenar los datos del estado y el número de paso.	

CLASE: ElementoInflado

	Esta clase representa un elemento completamente inflado.
ATRIBUTOS	
• **lista_poligonos:** lista de paralelepípedos resultantes del inflado de cada arista. • **poligono_convexo:** polígono original. • **puntos_envolvente:** puntos del polígono original.	
MÉTODOS	
Los métodos que ofrece esta clase sirven para tener acceso a los miembros, para añadir paralelepípedos a la clase, para realizar la intersección de polígonos y finalmente para obtener los segmentos que componen el resultado.	

CLASE: Inflador	
	Clase que infla los polígonos y sirve como almacén para todos y cada uno de los pasos de inflado
ATRIBUTOS	
• **estado_final:** estado final del inflado. • **lista_estados:** lista de estados del inflado. • **ultimo_paso:** último paso que se ha dado en el proceso de inflado.	

• **direcciones:** direcciones que se usarán para el proceso (en forma de vector). • **lista_direcciones:** direcciones que se usarán para el proceso. • **puntos:** lista de puntos de entrada. • **lista_segmentos:** lista de segmentos que conforman la solución. • **paso_actual:** número del paso actual. • **inflado_completo:** determina si el inflado está finalizado.
MÉTODOS
Los métodos de esta clase permiten avanzar un paso en el inflado.

5.3.3.Especificación de las interfaces de usuario

En la *Identificación de perfiles de usuario* sólo se ha detectado un perfil: usuario con conocimientos del área de aplicación del Sistema de Información el cual tendrá acceso a todos los procesos primitivos del sistema.

Además este acceso será totalmente libre ya que no se ha especificado ningún requisito de seguridad.

Definición de interfaces

Al entrar en la aplicación aparecerá una ventana con el icono de la aplicación en la parte superior izquierda. Esta ventana permitirá acceder a la ventana de visualización correspondiente a cada algoritmo.

Las principales interfaces usadas por el sistema son:

- **Ventana de visualización para el método de la intersección de semiplanos escalera:** esta ventana mostrará un control selector de direcciones, un área de dibujo, un menú con diferentes opciones, botones para limpiar la pantalla y calcular el resultado y botones para ver el resultado paso a paso.

- **Ventana de visualización para el método del inflado:** esta ventana mostrará un control selector de direcciones, un área de dibujo, un menú con diferentes opciones, botones para limpiar la pantalla y calcular el resultado y botones (play, pause, next, previous) para controlar la animación.

Relaciones de las interfaces con los casos de uso del sistema

A- Ventana de visualización para el método de la intersección de semiplanos escalera

B- Ventana de visualización para el método del inflado

	A	B
Caso de uso visualización de envolventes convexas y triangulaciones con direcciones restringidas (método de la escalera)	x	
Caso de uso visualización de envolventes convexas y triangulaciones con direcciones restringidas (método del inflado)		x

Tabla 5.1. Relación de interfaces con casos de uso

Pautas generales de diseño

Los colores a usar en todas las pantallas de la aplicación serán lo más agradables posibles para facilitar el uso de la aplicación y evitar que canse la vista así como evitar que desagrade a los usuarios. La combinación de colores elegida es:

- Fondos de ventana de color gris estándar de Windows.
- Fondos de área de dibujo de color verde claro o amarillo claro.
- Botones rectangulares del color estándar de Windows.
- Algunos botones serán imágenes que representen con claridad la funcionalidad que se desencadenará si son pulsados.
- En lo que se refiere a las fuentes:
    - Los textos de los botones y de las cabeceras de las ventanas tendrán una fuente *Microsoft Sans Serif* de un tamaño de 8,25 puntos.
    - Los textos de los menús tendrán una fuente *Segoe UI* de tamaño 9 puntos.

En general se ha de buscar una distribución cómoda y, en la medida de lo posible, jerárquica, para que al usuario le resulte intuitiva la utilización de la aplicación.

Al entrar en la aplicación aparecerá una ventana con dos botones: uno para entrar en la ventana de visualización del método de intersección de semiplanos escalera y otro para la visualización del método del inflado.

Al pinchar sobre el botón correspondiente al método de la intersección de semiplanos escalera se muestra una ventana correspondiente a la descrita en **Ventana de visualización para el método de la intersección de semiplanos escalera.**

Al pinchar sobre el botón correspondiente al método del inflado se muestra una ventana correspondiente a la descrita en **Ventana de visualización para el método del inflado.**

Elección de teclas

Se usarán combinaciones de teclas fáciles de recordar para el usuario de forma que no tenga que estar consultando la ayuda o los manuales de forma continua.

La distribución de comandos es la siguiente:

- Enter - pulsar el botón/menú seleccionado.
- Tabulador - viajar entre los campos en el formulario.
- Flechas de dirección - viajar por los controles que pertenecen a un mismo grupo.
- Alt + F4 - cerrar una ventana.

Además del teclado podremos usar el ratón facilitando aún más si cabe la navegabilidad entre los elementos de la página.

Prototipos de interfaces

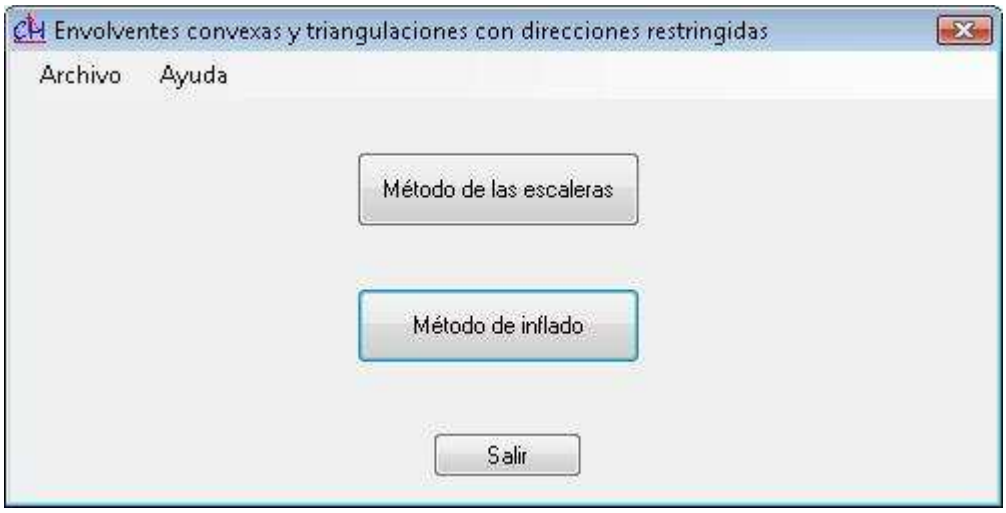


Imagen 5.1. Prototipo de ventana inicial de la aplicación

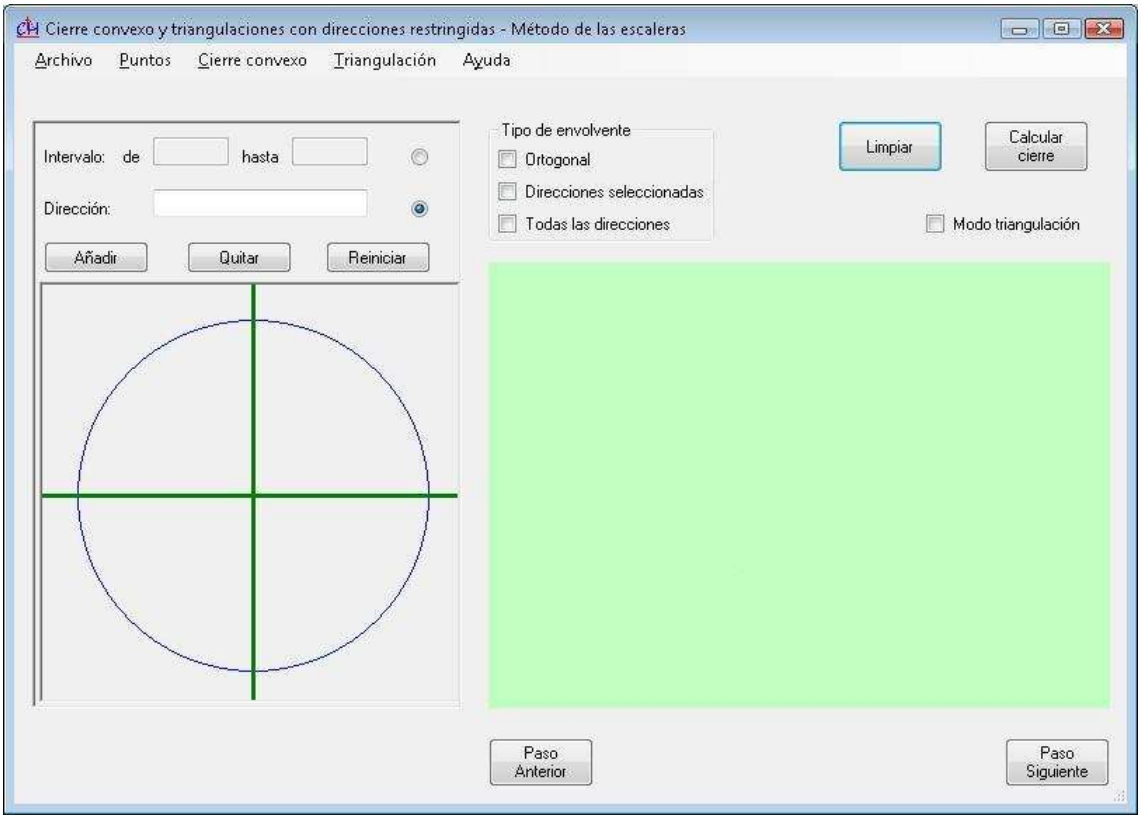


Imagen 5.2. Prototipo de interfaz para el método de las escaleras

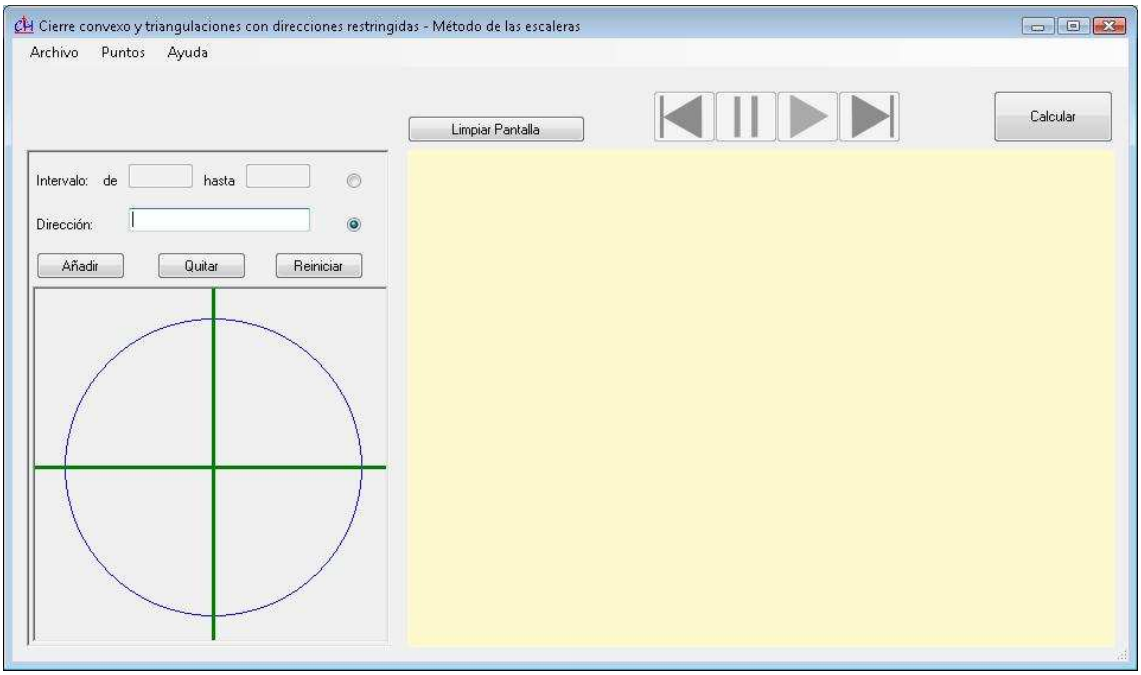


Imagen 5.3. Prototipo de interfaz para el método del inflado.

5.3.4.Análisis de consistencia

Requisitos funcionales	
Requisito	**Aparece reflejado en…**
FUNC1 Visualización de envolventes convexas por el método de la intersección de semiplanos escalera	Caso de uso Visualización de Envolventes Convexas y triangulaciones con direcciones restringidas (método de la escalera)
FUNC1.1 Selección de puntos por fichero o ratón	Caso de uso Gestión de puntos (en 5.3.1.2)
FUNC1.1.1 Cargar puntos desde fichero	Caso de uso Cargar puntos (en 5.3.1.2)
FUNC1.1.2 Guardar puntos a fichero	Caso de uso Guardar puntos (en 5.3.1.2)
FUNC1.2 Realización de envolvente convexa (mediante el método de las escaleras)	Caso de uso Cálculo de envolventes convexas (en 5.3.1.2)
FUNC1.3 Modo de visualización paso a paso	Caso de uso Visualización de Envolventes Convexas y triangulaciones con direcciones restringidas (método de la escalera)
FUNC2 Conmutación entre visualización de envolventes convexas y triangulaciones (sólo en método de las escaleras)	Caso de uso Visualización de Envolventes Convexas y triangulaciones con direcciones restringidas (método de la escalera)
FUNC3 Visualización de triangulaciones (sólo en método de las escaleras)	Caso de uso Visualización Triangulación (en 5.3.1.2)
FUNC3.1 Carga de triangulaciones desde fichero	Caso de uso Cargar triangulación (en 5.3.1.2)
FUNC3.2 Introducción de puntos a través de la interfaz y generación de la triangulación de Delaunay	Caso de uso Triangular selección de puntos (en 5.3.1.2)
FUNC3.3 Guardar triangulaciones de	Caso de uso Guardar Triangulación (en 5.3.1.2)

Delaunay	
FUNC4 Visualización de envolventes convexas por el método del inflado.	Caso de uso Visualización de Envolventes Convexas y triangulaciones con direcciones restringidas (método del inflado)
FUNC4.1 Selección de puntos por fichero o ratón	Caso de uso Gestión de puntos (en 5.3.1.3)
FUNC4.1.1 Cargar puntos desde fichero	Caso de uso Cargar puntos (en 5.3.1.3)
FUNC4.1.2 Guardar puntos a fichero	Caso de uso Guardar puntos (en 5.3.1.3)
FUNC4.2 Realización de la envolvente convexa (mediante el método de inflado)	Caso de uso Calcular envolvente (inflado) (en 5.3.1.3)
FUNC4.3 Modo de visualización con animación	Caso de uso Animar triangulación (en 5.3.1.3)
FUNC5 Selección de direcciones	Caso de uso Gestión de direcciones (en 5.3.1.2 y 5.3.1.3)
FUNC5.1 Añadir direcciones	Caso de uso Añadir dirección (en 5.3.1.2 y 5.3.1.3)
FUNC5.2 Quitar direcciones	Caso de uso Eliminar dirección (en 5.3.1.2 y 5.3.1.3)
FUNC5.3 Reiniciar direcciones	Caso de uso Gestión de direcciones (en 5.3.1.2 y 5.3.1.3)
FUNC6 Selección de los modos de envolvente (sólo en método de las escaleras)	Caso de uso Selección de modo de direcciones (en 5.3.1.2)
FUNC7 Limpiar pantalla	Caso de uso Limpiar pantalla (en 5.3.1.2 y 5.3.1.3)
FUNC8 Guardar área de visualización como imagen	Caso de uso Captura de imágenes (en 5.3.1.2 y 5.3.1.3)
FUNC9 Preferencias de visualización	Caso de uso Configuración interfaz (en 5.3.1.2 y 5.3.1.3)

Tabla 5.2. Requisitos funcionales

Requisitos de Datos	
Requisito	**Aparece reflejado en…**
DAT1 Punto	Estructura myPoint
DAT2 Dirección	Estructura direction
DAT3 Lista de direcciones	Clase ListOfDirections
DAT4 Lista de puntos	Clase ListOfPoints
DAT5 Fichero de puntos	Caso de uso guardar puntos (en 5.3.1.2 y 5.3.1.3)
DAT6 Fichero de triangulación	Caso de uso Guardar Triangulación (en 5.3.1.2)
DAT7 Estado en el proceso de inflado	Estructura Estado Clase EstadoInflado Clase ElementoInflado
DAT8 Lista de estados del proceso de inflado	Clase Inflador

Tabla 5.3. Requisitos de datos

Requisitos de Interfaz	
Requisito	**Aparece reflejado en…**
I1 Interfaz de visualización	Especificación de las interfaces de usuario (5.3.3)
I2 Control selector de direcciones	Especificación de las interfaces de usuario (5.3.3)
I3 Formato visual	Especificación de las interfaces de usuario (5.3.3)
I3.1 Colores agradables	Especificación de las interfaces de usuario (5.3.3)
I4 Teclas de función	Especificación de las interfaces de usuario (5.3.3)

Tabla 5.4. Requisitos de interfaz

Requisitos de Eficiencia	
Requisito	**Aparece reflejado en…**
E1 Tiempos de respuesta aceptables	Requisito orientado a la fase de diseño

Tabla 5.5. Requisitos de eficiencia

5.3.5.Especificación del plan de pruebas

En este apartado se llevará a cabo una especificación del plan de pruebas en la que se explicará brevemente que pruebas se van a realizar, sobre qué elementos del sistema, cómo se van a realizar, cuándo y quién interviene en ellas. Empezaremos por enumerar los tipos de pruebas que se van a realizar especificando para cada una sobre qué tipo de elemento se van a realizar y en qué momento.

Durante la codificación del sistema de información se realizarán los siguientes tipos de pruebas:

- **Pruebas unitarias**: se realizarán sobre cada método individual de cada clase nada más acabar la programación de la clase y sobre las funciones auxiliares. Serán llevadas a cabo por el autor del presente proyecto. Las pruebas se realizarán usando un enfoque de caja negra. Para cada elemento de código probado se observará que la salida sea la esperada. En caso de que no lo sea se modificará el módulo hasta que se obtengan los resultados esperados. Se omitirá la prueba de métodos que involucren clases ajenas y se dejarán para las pruebas de integración.

- **Pruebas de integración**: se realizarán una vez estén probadas individualmente las clases, sobre clases interdependientes. Estas pruebas serán llevadas a cabo por el autor del presente proyecto. Las pruebas se realizarán desde abajo hacia arriba, integrando las clases con menor nivel de acoplamiento y probándolas, así sucesivamente hasta llegar al mayor nivel de acoplo. Se usarán conjuntos de

datos de prueba que nos permitan probar la integración de las clases. No se dará una prueba por buena hasta que para cada entrada de datos la salida, resultado de probar los métodos de las clases que se integran, sea la esperada.

- **Pruebas del sistema**: se realizarán sobre el sistema de información una vez estén realizadas las pruebas anteriores. Estas pruebas serán realizadas por el autor del presente proyecto. Será una prueba de integración de todo el sistema en su totalidad, es decir, integrando todas las clases que componen el sistema de información. Las pruebas se darán por terminadas una vez que tanto el autor del proyecto como el tutor estimen el Sistema de Información acabado.

5.4. Diseño orientado a objetos

5.4.1. Definición de la arquitectura del sistema y de la arquitectura de soporte

Máquinas

Para hacer uso de este sistema de información es necesario que cada usuario que vaya a usarlo disponga de un ordenador con las siguientes características técnicas: procesador de al menos 1.5 GHz y 1 Gb de memoria RAM. No hay requerimientos específicos en cuanto a la cantidad de espacio en disco duro disponible.

El ordenador además debe contar con teclado y ratón como dispositivos de entrada.

Sistema operativo

La aplicación sólo funcionará sobre sistemas operativos de Microsoft debido a la utilización de la tecnología .NET. A pesar de que hay algunas versiones de .NET para Linux éstas no alcanzan aún la versión requerida por la presente aplicación.

Posibles módulos a utilizar de librerías generales

Los módulos o librerías comunes a utilizar se corresponden con las que ofrece .NET para el desarrollo de aplicaciones de ventana y las que ofrece CGAL para la realización de todo tipo de operaciones de Geometría Computacional.

5.4.2. Diseño de algoritmos

Se sustituye la fase de Diseño de casos de uso reales por la de Diseño de algoritmos con el objetivo de explicar detalladamente cómo se han diseñado los algoritmos de cálculo de la envolvente convexa con direcciones restringidas. Por lo tanto se omitirá el diseño, en este caso trivial, de la mayoría de los casos de uso, centrando este apartado únicamente en los casos de uso que están directamente relacionados con el cálculo de la envolvente convexa.

5.4.2.1. Diseño del algoritmo de la intersección de semiplanos escalera

El pseudocódigo de este algoritmo fue presentado con anterioridad en este mismo documento. En base a ello se realizará un diseño del mismo.

La entrada de este algoritmo consta de un conjunto de puntos, representado por una instancia de la clase *ListOfPoints*, y un conjunto de direcciones representado por una instancia de la clase *ListOfDirections* (cabe destacar que para el caso de intervalos de direcciones se ha optado por introducir direcciones de 5º en 5º entre ambos extremos del intervalo). Además se considera oportuno separar de manera clara los tres posibles casos: que el conjunto contenga todas las direcciones, que el conjunto de direcciones sea 0º y 90º o que el conjunto contenga cualquier otra combinación de direcciones.

El primer caso se corresponde con el caso habitual de la envolvente convexa por lo que se puede realizar esta operación aprovechando las funciones para el cálculo de la envolvente convexa ofrecidas por CGAL.

Los dos casos siguientes son análogos pero hay unos pequeños detalles que hacen que merezca la pena separar el paso de construcción de escaleras. De cualquier modo en ambos casos lo primero que se hace es construir el conjunto de direcciones normales, funcionalidad que pertenecerá a la lista de direcciones.

Posteriormente para cada dirección normal se calcula su punto maximal. Como se ha mencionado anteriormente para hacer esto hay que rotar el plano. Para ello se aplicará una matriz de rotación y la funcionalidad de CGAL para calcular puntos maximales.

Una vez hecho esto, para cada dos direcciones se calculará la escalera correspondiente. Para conseguirlo se crearán dos métodos distintos, uno para escaleras del caso ortogonal (entendiendo como ortogonal el caso de direcciones 0º y 90º) y otro para el resto de escaleras. La diferencia radica en la forma de deslizar una recta sobre otra y en la forma de encontrar los puntos intermedios de la escalera (es más precisa esta operación en el caso “ortogonal”).

Ambos métodos tendrán la misma estructura y harán uso de la funcionalidad de CGAL para el cálculo de intersecciones. Cabe destacar que en el algoritmo original la recta que se desliza hace un barrido en busca de puntos que pertenezcan a esta escalera. Para barrer este espacio desde código habría que hacer incrementos infinitamente pequeños, lo cual no es posible. Por lo tanto se hará lo siguiente: dotar a los puntos de un radio para que al comprobar si existe un punto nuevo en la escalera, se compruebe si la recta corta al radio del punto, con ello se puede determinar con precisión la recta real que forma la nueva escalera.

Además en la implementación del algoritmo antes de comenzar una escalera se filtrarán los puntos dejando sólo aquellos que pertenecen al paralelepípedo que forman los puntos maximales y las rectas con igual dirección a las dos direcciones de entrada pasando por dichos puntos maximales.

Una vez calculadas todas las escaleras se realiza la intersección de escaleras pertenecientes a direcciones opuestas. Para ello se comprobará segmento a segmento de la escalera si hay intersección. En caso de haberla se unirán las escaleras y se eliminarán los puntos oportunos, según lo expuesto en el apartado cuarto de la presente memoria. Debido a cuestiones de representación la intersección se obviará en caso de que las escaleras se corten en más de dos puntos.

Una vez hechas todas estas operaciones se retornará una instancia de *ListOfPoints* con la envolvente convexa. Para un mejor entendimiento del proceso se recurrirá a un diagrama de secuencia.

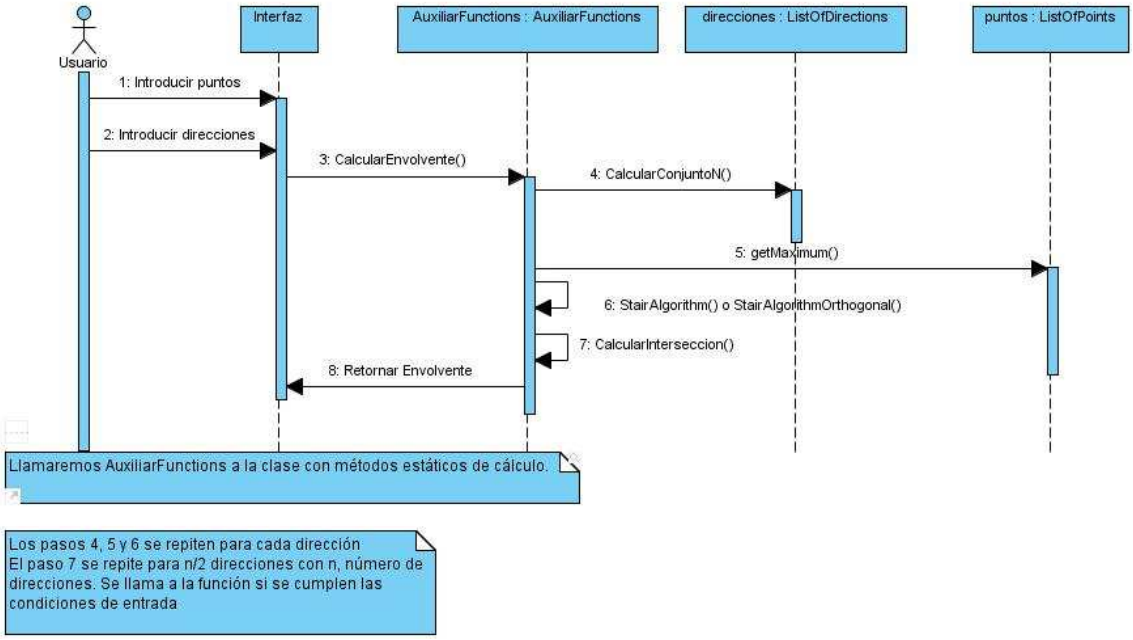


Diagrama 5.5. Diagrama de secuencia para el método de la escalera.

Para el caso de los triángulos el proceso es el mismo que el descrito pero aplicado a cada triángulo por separado.

Se estima oportuno implementar toda esta funcionalidad en forma de clase de métodos estáticos.

5.4.2.2. Diseño del algoritmo de inflado

El pseudocódigo de este algoritmo fue presentado con anterioridad en este mismo documento. En base a ello se realizará un diseño del mismo.

Al igual que en el caso anterior la entrada constará de un conjunto de puntos representado por una instancia de la clase *ListOfPoints* y un conjunto de direcciones representado por una instancia de la clase *ListOfDirections*. Sin embargo en esta

ocasión la lista de puntos será triangulada, aprovechando la funcionalidad de CGAL, y será almacenada en forma de una lista de triángulos, es decir, el algoritmo realiza la envolvente convexa para cada triángulo de la triangulación.

El proceso a seguir para cada triangulo será el siguiente: inflar las aristas hasta llegar al máximo y una vez llegado al máximo obtener los segmentos que forman parte del resultado.

Gran parte del proceso de diseño de este algoritmo se realizará en función de las estructuras de datos obtenidas en la fase de análisis. Estas son: *Estado*, *EstadoInflado*, *ElementoInflado* e *Inflador*.

Una instancia de *Inflador* llevará a cabo el inflado (propiamente dicho) de cada arista y un control global del proceso. Se observan dos estados bien diferenciados en el inflado: el estado inicial y los siguientes.

En el estado inicial de cada arista se desconoce si realmente es necesario inflar la arista (no será necesario si la dirección pertenece al conjunto) ni los topes entre los cuales hay que inflar las aristas del paralelepípedo. Por lo cual habrá que calcular si pertenece al conjunto de direcciones o, si no pertenece, los topes. Para ello se pueden aprovechar las funciones ofrecidas por CGAL.

En los estados intermedios ya se conoce toda la información con lo cual lo único que habrá que hacer será inflar. El inflado deberá hacerse de la siguiente forma: para cada una de las dos direcciones de las aristas del paralelepípedo comprobar si han alcanzado el tope correspondiente. Si no han alcanzado el tope correspondiente se calcula el nuevo vector dirección y el nuevo paralelepípedo usando las funciones de intersección de CGAL. Si se alcanzó la dirección tope no se hace nada.

Tanto el paso inicial como los intermedios se almacenarán en una lista de *EstadoInflado* que servirá para tener un histórico de todo el proceso y posteriormente poder mostrarlo de manera animada.

El proceso termina cuando no se puede inflar más ninguna arista. En ese momento habrá que calcular los segmentos que forman parte de la envolvente convexa. Para ello habrá que implementar el método de descomposición en aristas mencionado en el apartado cuarto y ello se hará en la clase *ElementoInflado*, que representa un elemento completamente inflado con un paralelepípedo por arista (si la dirección de dicha arista no pertenece al conjunto de entrada). Para implementar la intersección de paralelepípedos habrá que usar las funciones ofrecidas por CGAL en lo referente a polígonos e intersecciones obteniendo así un listado de los segmentos resultantes de hacer la intersección de todos los paralelepípedos. Una vez obtenida esta lista hay que ver cuáles de estos segmentos forman parte del polígono original, esto se podrá hacer comprobando con CGAL cuales parejas de puntos están dentro del polígono.

Una vez hecho esto ya se podrá mostrar al usuario una animación. Para ello se usará un temporizador que cada cierto tiempo cambiará el paso a dibujar por el siguiente en la lista.

Para comprender mejor el proceso se hará uso de un diagrama de secuencia.

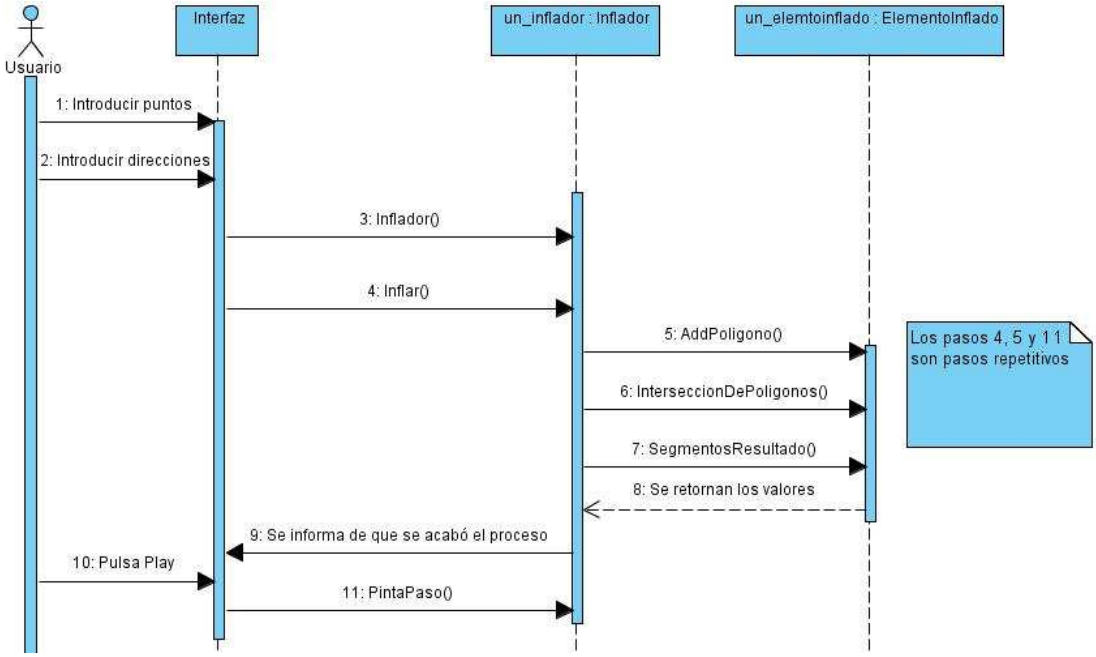


Diagrama 5.6. Diagrama de secuencia para el método del inflado.

5.4.3.Diagrama de Clases de diseño

En el diagrama presentado a continuación se incluyen todas las clases que conforman el sistema, desde las clases de datos hasta las clases relacionadas con la interfaz. Las clases añadidas son: *AuxiliarFunctions*, *FormularioEntrada*, *VentanaPrincipal* y *FormularioInflado*.

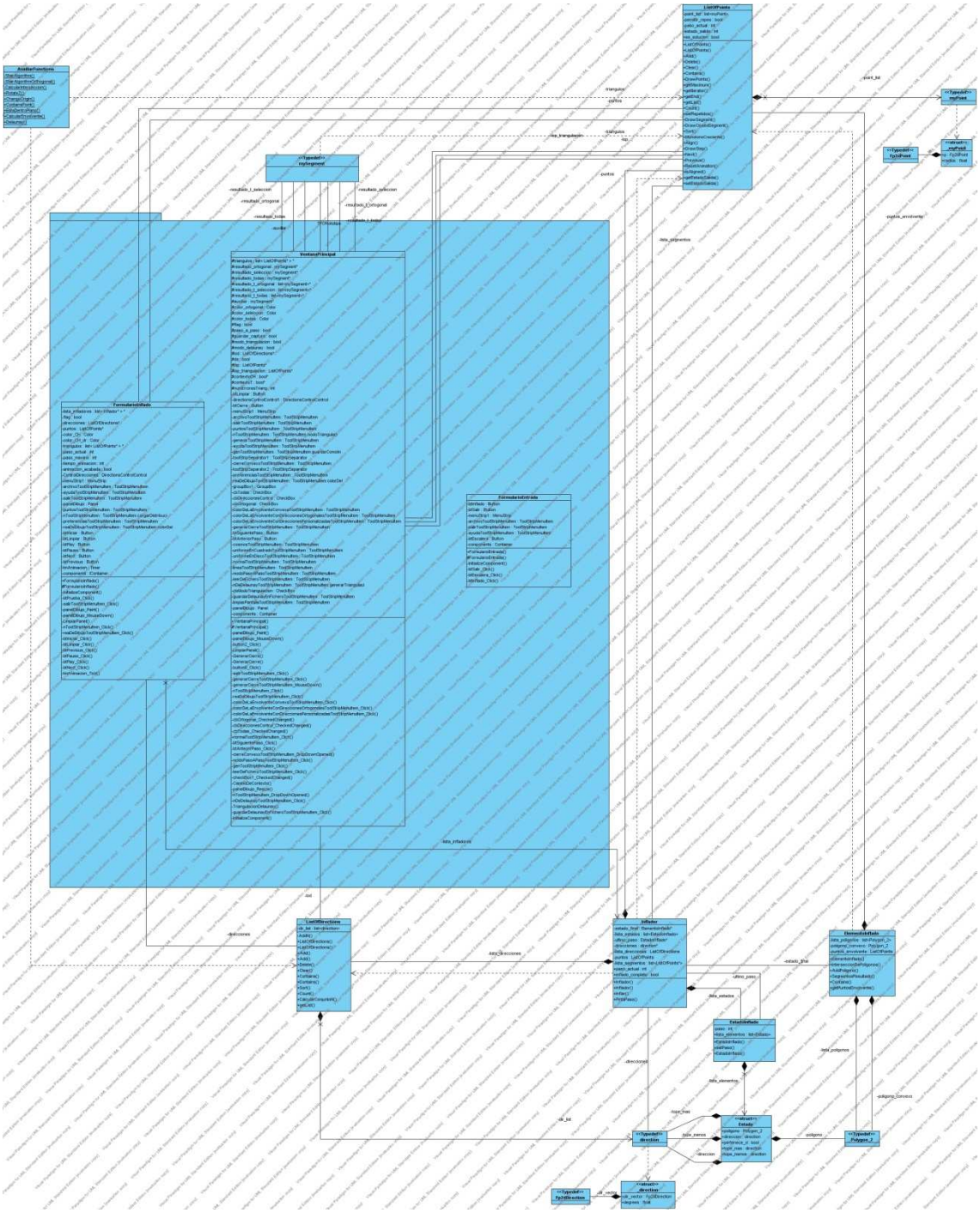


Diagrama 5.7. Diagrama de clases de diseño.

El diagrama de clases de diseño consta de 9 clases, 4 clases más respecto al diagrama de clases de análisis. Las relaciones resultantes de la inclusión de las nuevas clases se detallan a continuación.

ListOfDirections establece relaciones de asociación con *VentanaPrincipal* y con *FormularioInflado* ya que para cada instancia de *VentanaPrincipal* y *FormularioInflado* tiene que haber una instancia de *ListOfDirections*. Además *ListOfDirections* tiene una relación de dependencia con *AuxiliarFunctions* ya que esta última no puede existir si no existe la primera.

Por su parte *ListOfPoints* establece múltiples relaciones de asociación tanto con *VentanaPrincipal* como con *FormularioInflado* debido a que para cada instancia de éstas deben existir una o varias de *ListOfPoints*. Además *ListOfPoints* tiene una relación de dependencia con *AuxiliarFunctions* ya que esta última no puede existir si no existe la primera.

Finalmente *Inflador* establece una relación de asociación con *FormularioInflado* ya que para cada instancia de *FormularioInflado* deben existir una o más instancias de *Inflador*.

A continuación se presentará información detallada de las nuevas clases. No se repetirán las clases explicadas en la fase de análisis porque a las mismas sólo se les ha añadido algunos métodos de dibujo.

<table>
<tr><th colspan="2">CLASE: FormularioEntrada</th></tr>
<tr><td> 
</td><td>Esta clase representa el formulario que se inicia nada más lanzar el programa. Permitirá elegir entre el método del inflado y el método de las escaleras.</td></tr>
<tr><th colspan="2">ATRIBUTOS</th></tr>
<tr><td colspan="2">Los atributos de esta clase se corresponden con los controles de la interfaz de usuario. No hay, por lo tanto, ningún atributo que merezca un comentario especial.</td></tr>
<tr><th colspan="2">MÉTODOS</th></tr>
<tr><td colspan="2">Los métodos que ofrece esta clase sirven para iniciar la interfaz y para manejar los eventos que desencadenan los controles.</td></tr>
</table>

CLASE: FormularioInflado

Esta clase representa el formulario que sirve como interfaz al método del inflado.

ATRIBUTOS

- **lista_infladores:** lista de infladores que llevarán a cabo el inflado de cada uno de los triángulos de la triangulación.
- **flag:** bandera que indica si hay que borrar la pantalla.
- **direcciones:** lista de direcciones de entrada.
- **puntos:** lista de puntos de entrada.
- **triángulos:** lista de triángulos de la triangulación.
- **paso_actual:** número de paso actual en el inflado.
- **paso_maximo:** número de paso máximo en el inflado.
- **tiempo_animacion:** tiempo entre paso y paso de la animación de inflado.
- **animacion_acabada:** indica si la animación ha finalizado.

El resto de atributos corresponden a controles de la interfaz.

MÉTODOS

Los métodos que ofrece esta clase sirven para iniciar la interfaz y para manejar los eventos que desencadenan los controles.

CLASE: VentanaPrincipal	
	Esta clase representa el formulario que sirve como interfaz al método de las escaleras.
ATRIBUTOS	
• **triangulos:** lista de triángulos que forman una triangulación • **resultado_ortogonal:** resultado del cierre convexo con direcciones 0º y 90º. • **resultado_seleccion:** resultado del cierre convexo con direcciones seleccionadas por el usuario. • **resultado_todas:** resultado del cierre convexo en el caso habitual. • **resultado_t_ortogonal:** lista de resultados para cada triángulo usando direcciones 0º y 90º. • **resultado_t_seleccion:** lista de resultados para cada triángulo usando direcciones seleccionadas por el usuario. • **resultado_t_todas:** lista de resultados para cada triángulo usando todas las direcciones. • **flag:** bandera que indica si hay que borrar la pantalla. • **paso_a_paso:** indica si se está usando o no el modo paso a paso.	

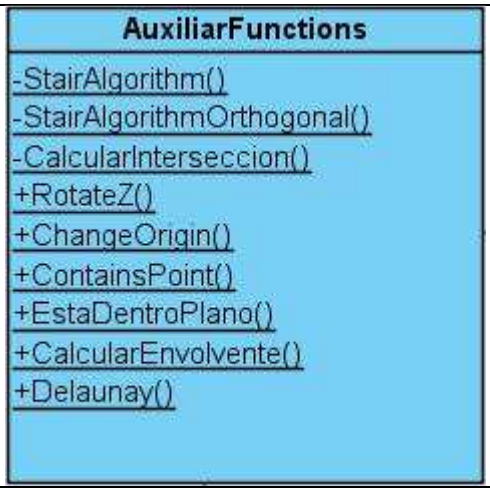
- **guardar_captura:** indica si hay que guardar una captura de pantalla o no.
- **modo_triangulación:** indica si el formulario está en modo envolvente o en modo triangulación.
- **modo_delaunay:** indica en el modo triangulación si los puntos se insertan a mano o mediante fichero.
- **lod:** lista de direcciones de entrada.
- **lop:** lista de puntos de entrada.
- **lop_triangulacion:** lista de puntos de entrada de la triangulación.

El resto de atributos corresponden a controles de la interfaz o atributos de gestión de dibujo.

MÉTODOS

Los métodos que ofrece esta clase sirven para iniciar la interfaz, para manejar los eventos que desencadenan los controles y para obtener los resultados a partir de los datos de entrada.

CLASE: AuxiliarFunctions



Esta clase sirve para agrupar las operaciones utilizadas en toda la aplicación.

ATRIBUTOS

No se presentan atributos.

MÉTODOS

- **StairAlgorithm:** lleva a cabo una escalera entre dos puntos maximales.
- **StairAlgorithmOrthogonal:** lleva a cabo una escalera rectilínea entre dos puntos maximales.
- **CalcularInterseccion:** calcula la intersección de dos escaleras.
- **RotateZ:** rota un punto sobre el eje Z.
- **ChangeOrigin:** realiza una transformación de cambio de origen.
- **ContainsPoint:** comprueba si un punto está en una línea con un cierto margen de error.
- **EstaDentroPlano:** comprueba si un punto está dentro de un plano.
- **CalcularEnvolvente:** lleva a cabo el cálculo de la envolvente.

- **Delaunay:** lleva a cabo la triangulación de Delaunay.

5.5. Establecimiento de requisitos de implantación

Las condiciones para el correcto funcionamiento de la aplicación son las siguientes:

- Disponer de un ordenador con un sistema operativo de los especificados.
- Entrega de los siguientes documentos:
  - Manual de explotación
  - Manual de usuario
  - Información sobre los requisitos mínimos de hardware y software

Una vez cumplidos todos estos requisitos se puede llevar a cabo la implantación del sistema con riesgos mínimos.

5.6. Código fuente de la aplicación

TFCPrototipo.cpp (punto de entrada de la aplicación)

```
#include "stdafx.h"

#include "FormularioEntrada.h"

using namespace TFCPrototipo;

/**
 * Punto de entrada de la aplicación.
 * Lanza una aplicación cuyo formulario de inicio es
FormularioEntrada.
*/

[STAThreadAttribute]
int main(array<System::String ^> ^args)
{
	// Habilitar los efectos visuales de Windows XP antes de
crear ningún control
	Application::EnableVisualStyles();
	Application::SetCompatibleTextRenderingDefault(false);

	// Crear la ventana principal y ejecutarla
	Application::Run(gcnew FormularioEntrada());
	return 0;
}
```

Types.h

```
#pragma once

#include <CGAL/Cartesian.h>

typedef   CGAL::Point_2<CGAL::Cartesian<float> >
Fp2dPoint;
typedef   CGAL::Direction_2<CGAL::Cartesian<float> >
Fp2dDirection;
typedef       CGAL::Line_2<CGAL::Cartesian<float>>
	Fp2dLine;
typedef CGAL::Plane_3<CGAL::Cartesian<float>>
	Fp3dPlane;

#define Y_AXIS 90
#define X_AXIS 0

typedef struct _myPoint{
	Fp2dPoint p;
	float radius;
}myPoint;

typedef struct _direction{
	Fp2dDirection dir_vector;
	float degrees;
}direction;
```

ListOfPoints.h

```
#pragma once
#include "Types.h"
#include <list>

class ListOfPoints{
private:
	std::list<myPoint> point_list;
	bool permitir_repes;
	int paso_actual;
	int estado_salida;
public:
	ListOfPoints();
	~ListOfPoints();
	bool es_solucion;
	void Add(float x, float y, float r);
	void Delete(myPoint p);
	void Clear();
	bool Contains(float x, float y);
	void DrawPoints(System::Drawing::Graphics ^g);
	myPoint getMaximum(direction d);
	std::list<myPoint>::iterator getIterator();
	std::list<myPoint>::iterator getEnd();
	std::list<myPoint> getList();
	int Count();
	void setRepetidos(bool repe);
	void DrawSegment(System::Drawing::Graphics ^g,
System::Drawing::Color color);
	void
ListOfPoints::DrawClosedSegment(System::Drawing::Graphics ^g,
System::Drawing::Color color);
	void Sort(int cuadrante);
	void MonotonoCreciente(int cuadrante);
	bool ListOfPoints::Align(float x, float y);
	void DrawStep(System::Drawing::Graphics ^g,
System::Drawing::Color color);
	void Next();
	void Previous();
	void ResetAnimation();
	bool isAligned(int x, int y);
	int getEstadoSalida();
	void setEstadoSalida(int s);
};
```

ListOfPoints.cpp

```
#include "stdafx.h"
#pragma once
#include "ListOfPoints.h"
#include <CGAL/ch_selected_extreme_points_2.h>
#include <CGAL/Cartesian.h>
#include <CGAL/convex_hull_traits_2.h>
#include <fstream>
#include <stack>
#include <set>
#include <string>
#include <list>

myPoint default_p = { Fp2dPoint(-1000,-1000), -1000};
#define DEFAULT_POINT default_p

/**
 * Constructor de la clase lista de puntos.
 */
ListOfPoints::ListOfPoints(){
	point_list.clear();
	permitir_repes = false;
	paso_actual = 0;
	estado_salida = 0;
}

/**
 * Destructor de la clase lista de puntos.
 */
ListOfPoints::~ListOfPoints(){
	point_list.clear();
}

/**
 * Vacia la lista de puntos.
 */
void ListOfPoints::Clear(){
	point_list.clear();
}

/**
 * Permite establecer si la lista admite repetidos o no.
 * @param repe Si se permiten repetidos o no.
 */
void ListOfPoints::setRepetidos(bool repe){
	permitir_repes = repe;
}

/**
 * Mueve la animación un paso adelante.
 */
void ListOfPoints::Next()
{
	if(paso_actual < (point_list.size() - 1))
		paso_actual++;
}
```

```
/**
 * Mueve la animación un paso atrás.
 */
void ListOfPoints::Previous()
{
	if(paso_actual > 0)
	{
		paso_actual--;
	}
}

/*
 * Reinicia la animación
 */
void ListOfPoints::ResetAnimation(){
	paso_actual = 0;
}

/**
 * Pinta un paso en la animación.
 * @param g Objeto graphics sobre el que se dibuja.
 * @param color Color de dibujado.
 */
void ListOfPoints::DrawStep(System::Drawing::Graphics ^g,
System::Drawing::Color color)
{
	//Máximo número de pasos
	//int max_pasos = point_list.size();
	//paso_actual++;
	std::list<myPoint>::iterator it;
	int tam = point_list.size();
	if(tam <= 0) return;
	int *x = new int[tam];
	int *y = new int[tam];
	int i = 0;
	System::Drawing::Pen ^p = gcnew
System::Drawing::Pen(color);
	for(it = point_list.begin(); it != point_list.end();
it++,i++ ){
		x[i] = it->p.x();
		y[i] = it->p.y();
	}
	for(int i = 0; i < tam && i < paso_actual; i++){
		g->DrawLine(p, x[i], y[i], x[i+1],y[i+1]);
		//i++;
		//if(i == (tam-1)) break;
	}


}

/**
 * Añade un punto a la lista.
 * @param x Coordenada X del punto.
 * @param y Coordenada Y del punto.
 * @param r Radio del punto.
 */
void ListOfPoints::Add(float x, float y, float r){
```

```
	if((!Contains(x, y) || permitir_repes)){
		myPoint new_p;
		new_p.p = Fp2dPoint(x, y);
		new_p.radius = r;
		point_list.push_back(new_p);
	}
}

/**
 * Dibuja los puntos.
 * @param g Objeto graphics sobre el que dibujar.
 */
void ListOfPoints::DrawPoints(System::Drawing::Graphics ^g){
	std::list<myPoint>::iterator it;
	System::Drawing::SolidBrush ^sd = gcnew
System::Drawing::SolidBrush(System::Drawing::Color::Black);
	for(it = point_list.begin(); it != point_list.end(); it++
){
		myPoint point_to_draw = *it;
		g->FillEllipse(sd,
point_to_draw.p.x(),point_to_draw.p.y(), point_to_draw.radius,
point_to_draw.radius);
	}

}

/**
 * Dibuja un segmento.
 * @param g Objeto graphics sobre el que se dibuja.
 * @param color Color de dibujado.
 */
void ListOfPoints::DrawSegment(System::Drawing::Graphics ^g,
System::Drawing::Color color){
	std::list<myPoint>::iterator it;
	int tam = point_list.size();
	if(tam <= 0) return;
	int *x = new int[tam];
	int *y = new int[tam];
	int i = 0;
	System::Drawing::Pen ^p = gcnew
System::Drawing::Pen(color);
	for(it = point_list.begin(); it != point_list.end();
it++,i++ ){
		x[i] = it->p.x();
		y[i] = it->p.y();
		//g->DrawLine(p,
p1.p.x(),p1.p.y(),p2.p.x(),p2.p.y());
	}
	i = 0;
	for(;;){
		g->DrawLine(p, x[i], y[i], x[i+1],y[i+1]);
		i++;
		if(i == (tam-1)) break;
	}

}

/**
```

```
 * Dibuja un segemento cerrado.
 * @param g Objeto graphics sobre el que se dibuja.
 * @param color Color de dibujado.
 */
void ListOfPoints::DrawClosedSegment(System::Drawing::Graphics
^g, System::Drawing::Color color){
      std::list<myPoint>::iterator it;
      int tam = point_list.size();
      if(tam <= 0) return;
      int *x = new int[tam];
      int *y = new int[tam];
      int i = 0;
      System::Drawing::Pen ^p = gcnew
System::Drawing::Pen(color);
      for(it = point_list.begin(); it != point_list.end();
it++,i++ ){
            x[i] = it->p.x();
            y[i] = it->p.y();
            //g->DrawLine(p,
p1.p.x(),p1.p.y(),p2.p.x(),p2.p.y());
      }
      i = 0;
      for(;;){
            int actual = i;
            int siguiente = (i + 1) % tam;
            g->DrawLine(p, x[actual], y[actual],
x[siguiente],y[siguiente]);
            i++;
            if(i == (tam)) break;
      }

}

/**
 * Cuenta el número de puntos.
 * @return Devuelve el número de puntos.
 */
int ListOfPoints::Count(){
      return point_list.size();
}

/*
 * Borra un punto de la lista.
 * @param p Punto a borrar.
 */
void ListOfPoints::Delete(myPoint p){
      if(!Contains(p.p.x(),p.p.y())) return;
      std::list<myPoint>::iterator it;

      for ( it=point_list.begin() ; it != point_list.end(); it++
){
            myPoint aux = *it;
            if(aux.p == p.p) break;
      }
      point_list.erase(it);
}

/**
```

```
 * Comprueba si un punto está alineado con alguno de la lista.
 * @param x Coordenada X.
 * @param y Coordenada Y.
 * @return Devuelve si está alineado o no.
 */
bool ListOfPoints::isAligned(int x, int y)
{
	std::list<myPoint>::iterator it;

	for ( it=point_list.begin() ; it != point_list.end(); it++
){
		if(it->p.x() == x && it->p.y() == y) continue;
		if(it->p.x() == x || it->p.y() == y) return true;
	}

	return false;
}

/**
 * Obtiene la lista de puntos.
 * @return Devuelve la lista de puntos.
 */
std::list<myPoint> ListOfPoints::getList(){
	return point_list;
}

/**
 * Comprueba si la lista contiene un punto.
 * @param x Coordenada X del punto a comprobar.
 * @param y Coordenada Y del punto a comprobar.
 * @return Devuelve si la lista contiene o no el punto.
 */
bool ListOfPoints::Contains(float x, float y){
	std::list<myPoint>::iterator it;
	if(point_list.size() < 1) return false;
	for ( it=point_list.begin() ; it != point_list.end(); it++
){
		myPoint aux = *it;
		if(aux.p.x() == x && aux.p.y() == y) return true;
	}

	return false;
}

/**
 * Comprueba si un punto está alineado vertical u
horizontalmente con alguno de la lista.
 * @param x Coordenada X.
 * @param y Coordenada Y.
 * @return Devuelve si está alineado o no.
 */
bool ListOfPoints::Align(float x, float y){
	std::list<myPoint>::iterator it;

	for ( it=point_list.begin() ; it != point_list.end(); it++
){
		myPoint aux = *it;
		if(aux.p.x() == x || aux.p.y() == y) return true;
```

```
		}

		return false;
	}

	/**
	 * Compara dos puntos.
	 * @param p1 Punto primero.
	 * @param p2 Punto segundo.
	 * @return Devuelve verdadero si el segundo está más abajo que
	el primero en caso de que estén alineados o en caso contrario si
	está más a la derecha.
	 */
	bool compareA(myPoint p1, myPoint p2){
		if(p1.p.x() == p2.p.x()){
			return p1.p.y() < p2.p.y();
		}else{
			return p1.p.x() > p2.p.x();
		}
	}

	/**
	 * Compara dos puntos.
	 * @param p1 Punto primero.
	 * @param p2 Punto segundo.
	 * @return Devuelve falso si el segundo está más abajo que el
	primero en caso de que estén alineados o en caso contrario si
	está más a la derecha.
	 */
	bool compareB(myPoint p1, myPoint p2){
		if(p1.p.x() == p2.p.x()){
			return p1.p.y() > p2.p.y();
		}else{
			return p1.p.x() < p2.p.x();
		}
	}

	/**
	 * Ordena los puntos según el cuadrante en que se esté.
	 * @param cuadrante Cuadrante en el que se desea hacer la
	ordenación.
	 */
	void ListOfPoints::Sort(int cuadrante){
		if(cuadrante == 1 || cuadrante == 4){
			point_list.sort(compareA);
		}else{
			point_list.sort(compareB);
		}
	}

	/**
	 * Ordena los puntos en una sucesión monótona creciente (método
	en desuso).
	 * @param cuadrante Cuadrante en el que se desean ordenar los
	puntos.
	 */
	void ListOfPoints::MonotonoCreciente(int cuadrante){
		std::list<myPoint> puntos_borrar;
```

```
	std::list<myPoint>::iterator it;
	std::list<myPoint>::iterator it2;

	myPoint anterior;
	if(cuadrante > 2){
		anterior.p = Fp2dPoint(-65000,-6500);
	}else{
		anterior.p = Fp2dPoint(-65000,-0);
	}
	for(it = point_list.begin(); it != point_list.end(); it++)
	{
		if(cuadrante > 2){
			if(abs(it->p.y()) < abs(anterior.p.y())){
				anterior = *it;
			}else{
				puntos_borrar.push_back(*it);
			}
		}else{
			if(abs(it->p.y()) > abs(anterior.p.y())){
				anterior = *it;
			}else{
				puntos_borrar.push_back(*it);
			}
		}
	}
	for(it2 = puntos_borrar.begin(); it2 !=
puntos_borrar.end(); it2++)
	{
		this->Delete(*it2);
	}
}

/**
 * Obtiene el estado de salida.
 * @return Devuelve el estado de salida.
 */
int ListOfPoints::getEstadoSalida()
{
	return estado_salida;
}

/**
 * Establece el estado de salida.
 * @param s Estado de salida.
 */
void ListOfPoints::setEstadoSalida(int s)
{
	estado_salida = s;
}

/**
 * Obtiene el punto maximal en una dirección.
 * @param d Dirección a comprobar.
 * @return Punto maximal en la dirección.
 */
myPoint ListOfPoints::getMaximum(direction d){
	if(d.degrees < 0) return DEFAULT_POINT;
	std::list<Fp2dPoint> lista_puntos;
```

```
	std::list<Fp2dPoint>    out_l;
	std::list<Fp2dPoint>::iterator it = out_l.begin();

	float rot_angle = Y_AXIS-d.degrees;

	lista_puntos.clear();
	std::list<myPoint>::iterator it2;

	for ( it2=point_list.begin() ; it2 != point_list.end();
it2++ ){
		Fp2dPoint punto;
		myPoint punto_rotado = *it2;
		punto_rotado = AuxiliarFunctions::RotateZ(rot_angle,
punto_rotado);
		punto = punto_rotado.p;
		lista_puntos.push_back(punto);

	}
	/*for(int i = 0; i < num_elems; i++){

	}*/

	CGAL::ch_n_point(lista_puntos.begin(),
lista_puntos.end(),it);

	Fp2dPoint max_p;
	max_p = *it;
	/*Chapuza para recuperar el punto original*/
	int pos_m = 0;
	bool encontrado = false;
	std::list<Fp2dPoint>::iterator it_busca;
	for (it_busca = lista_puntos.begin(); it_busca !=
lista_puntos.end(); it_busca++){
		if(max_p == *it_busca){
			encontrado = true;
			break;
		}
		pos_m++;
	}
	std::list<myPoint>::iterator it_encuentra;
	int i = 0;
	for(it_encuentra = point_list.begin(); it_encuentra !=
point_list.end(); it_encuentra++){
		if(i == pos_m){
			max_p = it_encuentra->p;
			break;
		}
		i++;
	}



	myPoint ret_val;
	ret_val.p = max_p;
	//ret_val = AuxiliarFunctions::RotateZ(-rot_angle,
ret_val);
	ret_val.radius = 3;
```

```
	return ret_val;
}

/**
 * Obtiene el inicio del iterador de la lista de puntos.
 * @return Iterador inicial de la lista.
 */
std::list<myPoint>::iterator ListOfPoints::getIterator(){
	return point_list.begin();
}

/**
 * Obtiene el fin del iterador de la lista de puntos.
 * @return Fin del iterador.
 */
std::list<myPoint>::iterator ListOfPoints::getEnd(){
	return point_list.end();
}
```

ListOfDirectios.h

```
#pragma once
#include "Types.h"
#include <list>

/*
 * Clase que representa una lista de direcciones.
 * Ofrece la funcionalidad necesaria para añadir, borrar, contar
y comprobar elementos.
 * Además nos permite ordenar la lista de menor a mayor y
obtener el conjunto normal del actual.
 */
class ListOfDirections{
private:
	//int num_elems;
	//myPoint** list;
	std::list<direction> dir_list;
public:
	ListOfDirections();
	~ListOfDirections();
	void Add(float x1, float y1, float deg);
	//Sobrecargamos el método Add para meter un ángulo y que se
calcule el vector director
	void Add(float deg);
	void Delete(direction d);
	void Clear();
	bool Contains(float deg);
	bool Contains(Fp2dDirection dir);
	void Sort();
	int Count();
	ListOfDirections CalcularConjuntoN();
	std::list<direction> getList();
};
```

ListOfDirectios.cpp

```
#include "stdafx.h"
```

```
#include "ListOfDirections.h"
#include <CGAL/Cartesian.h>
#include <math.h>
#define PI (3.14159265358979323846)

/**
 * Constructor de la clase lista de direcciones.
 */
ListOfDirections::ListOfDirections(){
      dir_list.clear();
}

/**
 * Destructor de la clase lista de direcciones.
 */
ListOfDirections::~ListOfDirections(){
      dir_list.clear();
      //dir_list.~list();
}

/**
 * Obtiene la lista de direcciones.
 * @return Devuelve la lista.
 */
std::list<direction> ListOfDirections::getList(){
      return dir_list;
}

/**
 * Función que compara dos direcciones.
 * @param d1 Dirección primera.
 * @param d2 Dirección segunda.
 * @return Devuelve si d1 es menor que d2.
 */
bool compare_dir(direction d1, direction d2){
      return d1.degrees < d2.degrees;
}

/**
 * Ordena la lista de direcciones.
 */
void ListOfDirections::Sort(){
      dir_list.sort(compare_dir);
}

/**
 * Cuenta los elementos de la lista.
 * @return Devuelve el número de elementos que tiene la lista.
 */
int ListOfDirections::Count(){
      return dir_list.size();
}


/**
 * Añade una dirección al conjunto (conjunto O).
 * Al añadir una dirección se comprueba que d € [0, 180).
 * @param x1 Coordenada x del vector dirección.
```

```
 * @param y1 Coordenada y del vector dirección.
 * @param deg Ángulo en grados de la dirección.
 */
void ListOfDirections::Add(float x1, float y1, float deg){
	if(deg >= 0 && deg < 180){
		Fp2dDirection dir = Fp2dDirection(x1, y1);

		direction new_dir = {dir, deg};

		if(!Contains(deg))	dir_list.push_back(new_dir);
		direction new_dir2 = {-dir, deg + 180};

		if(!Contains(deg+180))  dir_list.push_back(new_dir2);
	}
}

/**
 * Añade una dirección al conjunto (conjunto O).
 * Al añadir una dirección se comprueba que d € [0, 180).
 * @param deg Ángulo en grados de la dirección.
 */
void ListOfDirections::Add(float deg){
	if(deg >= 0 && deg < 180){
		float x, y;
		double radianes;
		//Pasamos de grados a radianes
		radianes = (deg * 2*PI)/360;
		//Obtenemos la 'x' y la 'y' de la recta
		y = (float)sin(radianes);
		x = (float)cos(radianes);
		//Apaño debido a los decimales
		if((float)abs(x) < 0.0000000001) x = 0;
		if((float)abs(y) < 0.0000000001) y = 0;
		//Creamos la linea que pasa por el punto
		Fp2dPoint punto = Fp2dPoint(x, y);
		Fp2dLine linea_deg = Fp2dLine(CGAL::ORIGIN, punto);
		//Creamos las nuevas variables para la direccion deg
y para la direccion deg+180
		direction direccion_deg, direccion_deg180;
		direccion_deg.degrees = deg;
		direccion_deg180.degrees = deg + 180;
		direccion_deg.dir_vector = linea_deg.direction();
		direccion_deg180.dir_vector = -linea_deg.direction();
		if(!Contains(deg) && !Contains(deg+180)){
			dir_list.push_back(direccion_deg);
			dir_list.push_back(direccion_deg180);
		}

	}
}

/**
 * Método que calcula el conjunto de direcciones N (normal) a
partir del conjunto actual (el conjunto O).
 * @return Devuelve el conjunto N.
 */
ListOfDirections ListOfDirections::CalcularConjuntoN(){
	ListOfDirections ret_val;
```

```
	std::list<direction>::iterator iterador;
	for(iterador = dir_list.begin(); iterador != 
dir_list.end(); iterador++){
		Fp2dLine linea = Fp2dLine(CGAL::ORIGIN, iterador-
>dir_vector);
		Fp2dDirection direccion_perpendicular = 
linea.perpendicular(CGAL::ORIGIN).direction();
		ret_val.Add(direccion_perpendicular.dx(), 
direccion_perpendicular.dy(), (int)(iterador->degrees + 90) % 
360);
	}

	ret_val.Sort();

	return ret_val;
}

/**
 * Borra todas las direcciones del conjunto.
 */
void ListOfDirections::Clear(){
	dir_list.clear();
}

/**
 * Comprueba si el conjunto contiene una dirección.
 * @param dir Dirección a comprobar.
 * @return Devuelve si la dirección está o no en el cojunto.
 */
bool ListOfDirections::Contains(Fp2dDirection dir){
	std::list<direction>::iterator it;

	for ( it=dir_list.begin() ; it != dir_list.end(); it++ ){
		direction aux = *it;
		if(aux.dir_vector == dir) return true;
	}

	return false;
}

/**
 * Comprueba si el conjunto contiene una dirección.
 * @param deg Ángulo en grados de la dirección a comprobar.
 * @return Devuelve si la dirección está o no en el cojunto.
 */
bool ListOfDirections::Contains(float deg){
	std::list<direction>::iterator it;

	for ( it=dir_list.begin() ; it != dir_list.end(); it++ ){
		direction aux = *it;
		if(aux.degrees == deg) return true;
	}

	return false;
}

/**
 * Borra una dirección del conjunto.
```

```
 * @param d Dirección a borrar.
 */
void ListOfDirections::Delete(direction d){
	if(!Contains(d.degrees)) return;
	std::list<direction>::iterator it;

	for ( it=dir_list.begin() ; it != dir_list.end(); it++ ){
		direction aux = *it;
		if(aux.degrees == d.degrees) break;
	}
	dir_list.erase(it);
}
```

AuxiliarFunctions.h

```
#pragma once
#include "Types.h"
#include "ListOfPoints.h"
#include "ListOfDirections.h"
//Definición de los tipos de modos de operación
typedef enum{
	ortogonal, todas, medio
}TCierre;

typedef ListOfPoints mySegment;

/*
 * Esta clase ofrece algunos métodos estáticos con operaciones
utilizadas en el cálculo de la envolvente convexa por el método
de la escalera.
 * También se ofrecen métodos para adecuar la entrada del
usuario a la representación usada en el programa.
*/
class AuxiliarFunctions{
private:
	static mySegment* StairAlgorithm(direction d1, direction
d2, myPoint maximal1, myPoint maximal2, ListOfPoints points);
	static mySegment* StairAlgorithmOrthogonal(direction d1,
direction d2, myPoint maximal1, myPoint maximal2, ListOfPoints
points);
	static mySegment* CalcularInterseccion(mySegment s1,
mySegment s2, bool *solucion);
	//static int estado_salida_sa;
public:
	static myPoint RotateZ(double angle,myPoint p);
	static myPoint ChangeOrigin(float matrix[3][3], double x,
double y);

	static bool ContainsPoint(Fp2dLine l, Fp2dPoint p, float r,
bool modo, Fp2dLine ll);
	static bool EstaDentroPlano(Fp2dLine line1, Fp2dLine line2,
Fp2dPoint point);
	static mySegment* CalcularEnvolvente(ListOfPoints
lista_puntos, ListOfDirections lista_direcciones, TCierre modo,
int *estado_salida);
```

```
	//static void CalcularEnvolventeV2(ListOfPoints
lista_puntos, ListOfDirections lista_direcciones, TCierre modo,
std::list<mySegment*>* lista_r);

	//static void DrawSegment(System::Drawing::Graphics ^g,
mySegment s);

	static void Delaunay(ListOfPoints lop,
std::list<ListOfPoints*> *triangulos);
	/*
	* Tipo enumerado que representa el estado de salida de
CalcularEnvolvente.
	* @see CalcularEnvolvente()
	*/
	typedef enum
	{
		correcto, /**< valor de salida correcto*/
		con_puntos_alineados, /**< valor de salida que indica
que hay puntos alineados*/
		error /**< valor de salida de error*/
	} TEstadoSalida;


};
```

AuxiliarFunctions.cpp

```
#pragma once
#include "stdafx.h"
#include <math.h>
#include <CGAL/ch_graham_andrew.h>
#include <CGAL/Polygon_2.h>
#include <iostream>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_euclidean_traits_xy_3.h>
#include <CGAL/Triangulation_euclidean_traits_2.h>
#include <CGAL/Delaunay_triangulation_2.h>




//#include <CGAL/Cartesian.h>
#define PI (3.141592653589793)
using namespace System;

//int estado_salida_sa;

/**
 * Tipo enumerado que representa las 8 direcciones del espacio.
 */
typedef enum
{
	arriba,
	arriba_izda,
	izda,
	abajo_izda,
```

```
	abajo,
	abajo_der,
	der,
	arriba_der
} TDirecciones;

//Constantes que indican si una dirección es menor, igual o
menor
#define IGUAL 1
#define MENOR 0
#define MAYOR 2


/**
 * Función que indica si d1 es IGUAL, MAYOR o MENOR que d2m
siendo d1 y d2 direcciones consecutivas.
 * @param d1 Dirección primera.
 * @param d2 Dirección segunda.
 * @return Devuelve IGUAL, MAYOR o MENOR.
 */
int CompararDirecciones(TDirecciones d1, TDirecciones d2)
{
	if(d1 == d2) return IGUAL;
	if(d1 == TDirecciones::arriba && d2 ==
TDirecciones::arriba_der) return MAYOR;
	if(d2 == TDirecciones::arriba && d1 ==
TDirecciones::arriba_der) return MENOR;
	if(d1 == TDirecciones::arriba && d2 == TDirecciones::der)
return MAYOR;
	if(d2 == TDirecciones::arriba && d1 == TDirecciones::der)
return MENOR;
	if(d1 < d2) return MENOR;
	else return MAYOR;
}

/**
 * Función que obtiene una dirección dada una coordenada.
 * @param x Coordenada X.
 * @param y Coordenada Y.
 * @return Devuelve la dirección.
 */
TDirecciones getDireccion(float x, float y)
{
	if(x == 1.0 && y == 0.0){
		return TDirecciones::der;
	}else if(x == 0.0 && y == 1.0){
		return TDirecciones::arriba;
	}else if(x == -1.0 && y == 0.0){
		return TDirecciones::izda;
	}else if(x == 0.0 && y == -1.0){
		return TDirecciones::abajo;
	}
	//Izquierda
	if(x < 0){
		if(y > 0){
			return TDirecciones::arriba_izda;
		}else if(y < 0){
			return TDirecciones::abajo_izda;
```

```
			}
		}else if(x > 0) //Derecha
		{
			if(y > 0){
				return TDirecciones::arriba_der;
			}else if(y < 0){
				return TDirecciones::abajo_der;
			}
		}
}

//struct K : CGAL::Exact_predicates_inexact_constructions_kernel
{};
/**Definición de los "traits" de la triangulación.*/
typedef
CGAL::Triangulation_euclidean_traits_2<CGAL::Cartesian<float>>
Gt;

/**
 * Método que calcula la triangulación de Delaunay  de una lista
de puntos dada.
 * @param lop Lista de puntos.
 * @param triangulos Lista de triangulos.
 */
void AuxiliarFunctions::Delaunay(ListOfPoints lop,
std::list<ListOfPoints*> *triangulos){
		CGAL::Delaunay_triangulation_2<Gt> dt;
		std::list<myPoint>::iterator it;
		//Se crea la triangulación
		for(it = lop.getIterator(); it != lop.getEnd(); it++){
			dt.push_back(it->p);
		}
		//Una vez creada la triangulación extraemos los triangulos
		CGAL::Delaunay_triangulation_2<Gt>::Finite_faces_iterator
f_it;

		for(f_it = dt.finite_faces_begin (); f_it !=
dt.finite_faces_end(); f_it++){
			ListOfPoints *triangulo = new ListOfPoints();
			CGAL::Delaunay_triangulation_2<Gt>::Face cara =
*f_it;
			for(int i = 0; i < 3; i++){
				//Obtenemos el i-esimo vertice

		CGAL::Delaunay_triangulation_2<Gt>::Vertex_handle v_handle
= cara.vertex(i);
				Fp2dPoint punto = v_handle->point();
				triangulo->Add(punto.x(), punto.y(), 3);
			}
			triangulos->push_back(triangulo);
		}
}

/**
 * Método que rota un punto una cierta cantidad de grados.
 * @param angle Grados a rotar.
 * @param origin Punto original a rotar.
 * @return Punto rotado.
```

```
 */
myPoint AuxiliarFunctions::RotateZ(double angle,myPoint origin){
	myPoint ret_val;
	angle = (angle * 2 * PI)/360;
	double tmpsin = sin(angle);
	double tmpcos = cos(angle);
	/*if(tmpsin < 0.00000000001) tmpsin = 0;
	if(tmpcos < 0.00000000001) tmpcos = 0;*/

	double point[4] = {origin.p.x(), origin.p.y(), 0, 1};
	double ret_point[2];
	double matrix[4][4] = {
		{tmpcos, tmpsin, 0, 0},
		{-tmpsin, tmpcos, 0, 0},
		{0, 0, 1, 0},
		{0, 0, 0, 1}
	};

	//Multiplicamos filas por columnas
	for(int i = 0; i < 2; i++){
		ret_point[i] =  point[0]*matrix[0][i] +
point[1]*matrix[1][i] + point[2]*matrix[2][i] +
point[3]*matrix[3][i];
		//if(ret_point[i] < powf(1,-10)) ret_point[0] = 0;
	}

	ret_val.p = Fp2dPoint((float)ret_point[0],
(float)ret_point[1]);

	return ret_val;
}

/**
 * Método que realiza un cambio de origen de un punto.
 * @param matrix Matriz de cambio de origen.
 * @param x Coordenada X.
 * @param y Coordenada Y.
 * @return Punto modificado.
*/
myPoint AuxiliarFunctions::ChangeOrigin(float matrix[3][3],
double x, double y){
	double point[3] = {x, y,1};
	myPoint ret_val;
	double ret_point[2];
	for(int i = 0; i < 2; i++){
		ret_point[i] =  point[0]*matrix[0][i] +
point[1]*matrix[1][i] + point[2]*matrix[2][i];
	}
	ret_val.p = Fp2dPoint((float)ret_point[0],
(float)ret_point[1]);

	return ret_val;
}


/**
 * Función que comprueba si un punto esta dentro de un polígono.
 * @param l1 Primer segmento del polígono.
```

```
 * @param l2 Segundo segmento del polígono.
 * @param l3 Tercero segmento del polígono.
 * @param l4 Cuarto segmento del polígono.
 * @param punto_check Punto a comprobar.
 * @return Si el punto esta dentro o no.
 */
bool EstaDentroDeUnPoligono(Fp2dLine l1, Fp2dLine l2, Fp2dLine
l3, Fp2dLine l4, Fp2dPoint punto_check){
      bool ret_val;
      bool ret_val2;
      ret_val = l1.has_on_negative_side(punto_check) &&
            l2.has_on_negative_side(punto_check) &&
            l3.has_on_negative_side(punto_check) &&
            l4.has_on_negative_side(punto_check);
      ret_val2 = l1.has_on(punto_check) ||
            l2.has_on(punto_check) ||
            l3.has_on(punto_check) ||
            l4.has_on(punto_check);
      return ret_val || ret_val2;
}

/**
 * Función que comprueba si un punto esta dentro de un polígono.
 * @param pt Punto a comprobar.
 * @param pgn_begin Inicio del vector de puntos que determina el
polígono.
 * @param pgn_end Final del vector de puntos que determina el
polígono.
 * @param traits Traits necesarios para CGAL.
 * @return Si el punto esta dentro o no.
 */
bool EstaDentroDeUnPoligono(Fp2dPoint pt, Fp2dPoint *pgn_begin,
Fp2dPoint *pgn_end, CGAL::Cartesian<float> traits)
{
  switch(CGAL::bounded_side_2(pgn_begin, pgn_end,pt, traits)) {
    case CGAL::ON_BOUNDED_SIDE :
      return true;
    case CGAL::ON_BOUNDARY:
      return true;
    case CGAL::ON_UNBOUNDED_SIDE:
      return false;
  }
}


/**
 * Método que lleva a cabo el algoritmo de la escalera entre dos
direcciones.
 * @param d1 Dirección de origen de la escalera.
 * @param d2 Dirección de destino de la escalera.
 * @param maximal1 Punto maximal de origen.
 * @param maximal2 Punto maximal de destino.
 * @param points Conjunto de puntos.
 * @return Devuelve la escalera resultante.
 */
mySegment* AuxiliarFunctions::StairAlgorithm(direction d1,
direction d2, myPoint maximal1, myPoint maximal2, ListOfPoints
points){
```

```
      //Si p_1=p_2, devolver ese punto
      if(maximal1.p == maximal2.p) return NULL;

      //Ahora YO incluiría aquí un pequeño paso intermedio:
filtrar los puntos que pertenecen al paralelepipedo.
      direction n1_90, n2_90; //Sumaremos 90 a las dos
direcciones de entrada
      n1_90.degrees = d1.degrees + 90;
      n2_90.degrees = d2.degrees + 90;
      Fp2dLine lin_n1, lin_n2;
      lin_n1 = Fp2dLine(CGAL::ORIGIN, d1.dir_vector);
      lin_n2 = Fp2dLine(CGAL::ORIGIN, d2.dir_vector);
      //Obtenemos los vectores perpendiculares
      n1_90.dir_vector =
lin_n1.perpendicular(CGAL::ORIGIN).direction();
      n2_90.dir_vector =
lin_n2.perpendicular(CGAL::ORIGIN).direction();
      //Construimos los 4 lados del paralelepipedo que contendrá
los puntos de trabajo.
      Fp2dLine lin_n1_90, lin_n2_90, lin_n1_menos_90,
lin_n2_menos_90;
      lin_n1_90 = Fp2dLine(maximal2.p, n1_90.dir_vector);
      lin_n2_90 = Fp2dLine(maximal1.p, n2_90.dir_vector);
      lin_n1_menos_90 = Fp2dLine(maximal1.p, -n1_90.dir_vector);
      lin_n2_menos_90 = Fp2dLine(maximal2.p, -n2_90.dir_vector);

      //bool caso_especial =
!lin_n2_menos_90.has_on_negative_side(maximal2.p);

      ListOfPoints nueva_lista;
      std::list<myPoint>::iterator iterador_filtro;
      CGAL::Object l1_i_l2;
      l1_i_l2 = CGAL::intersection(lin_n1_90,lin_n2_90);
      const Fp2dPoint * inter1 =
CGAL::object_cast<Fp2dPoint>(&l1_i_l2);
      CGAL::Object l1_i_l2_menos_90;
      l1_i_l2_menos_90 =
CGAL::intersection(lin_n1_menos_90,lin_n2_menos_90);
      const Fp2dPoint * inter2 =
CGAL::object_cast<Fp2dPoint>(&l1_i_l2_menos_90);
      Fp2dPoint p1, p2;
      p1 = (*inter1);
      p2 = (*inter2);
      Fp2dPoint puntos[] = {maximal1.p,p1,maximal2.p,p2};
      //VIGILAR SI INCLUYE SIEMPRE O NO LOS PUNTOS MAXIMALES -
hay algunos casos que se los salta...
      for(iterador_filtro = points.getIterator(); iterador_filtro
!= points.getEnd(); iterador_filtro++){
            //if(EstaDentroDeUnPoligono(lin_n2_90, lin_n1_90,
lin_n2_menos_90, lin_n1_menos_90, iterador_filtro->p)){
            if(EstaDentroDeUnPoligono((*iterador_filtro).p,
puntos, puntos+4, CGAL::Cartesian<float>())){
                  if(nueva_lista.isAligned(iterador_filtro-
>p.x(), iterador_filtro->p.y()))
nueva_lista.setEstadoSalida(TEstadoSalida::con_puntos_alineados)
;
                  nueva_lista.Add(iterador_filtro->p.x(),
iterador_filtro->p.y(), 3);
```

```
                }
        }
        nueva_lista.Add(maximal1.p.x(), maximal1.p.y(), 3);
        nueva_lista.Add(maximal2.p.x(), maximal2.p.y(), 3);
        //Sustituimos la lista original por la lista filtrada
        points = nueva_lista;
        mySegment* escalera = new mySegment();
        //Al añadir un punto podemos quitarlo del conjunto inicial
        escalera->setRepetidos(false);
        escalera->Add(maximal1.p.x(), maximal1.p.y(),
maximal1.radius);
        points.Delete(maximal1);
        //Trabajaremos SOLO con lin_n1_90 y lin_n2_90. Deslizaremos
n1 SOBRE n2
        bool salir = false; //Variable que controla si se sale del
algoritmo o no
        float x, y = maximal1.p.y();
        float y_antes = maximal1.p.y();
        float x_antes = maximal1.p.x();
        int incremento = 1; //El incremento siempre será de 1.
        int signo = 1;
        int ar_ab;
        int iz_der;
        if(maximal1.p.y() - maximal2.p.y() < 0) ar_ab = 1;
        else ar_ab = -1;
        if(maximal1.p.x() - maximal2.p.x() < 0) iz_der = 1;
        else iz_der = -1;
        if(n2_90.dir_vector.dy() < 0) signo = -1;
        incremento = signo * incremento;
        double MAX_PASOS = 1000000;
        double pasos = 0;
        //Declaramos el vértice de la recta de arrastre.
        //El punto de comienzo es el punto1
        Fp2dPoint vertice = Fp2dPoint(maximal1.p.x(),
maximal1.p.y());
        while(!salir){
                /*Considerar el rayo que sale de p_1 y tiene
dirección n_2+90.
                Sobre ese rayo deslizar el extremo de un segundo rayo
con dirección n_1+90.
                Cuando este segundo rayo se encuentre con un punto q
del conjunto:
                Si q=p_2, devolver el segmento recorrido sobre el
primer rayo,
                junto con el segmento del segundo rayo hasta q.
                FIN.
                Si no:

                Añadir a la escalera el segmento recorrido sobre el
primer rayo,
                junto con el segmento del segundo rayo hasta q.
                Actualizar p_1<--q y repetir el proceso para p_1 y
p_2.*/
                //Movemos la linea n1 sobre n2. Considerando como
casos especiales las lineas horizantales y verticales
                if(lin_n2_90.is_vertical() ||
lin_n2_90.is_horizontal()){
```

```
                  //La X y la Y de los vectores directores de las
lineas H y V SIEMPRE valdrán +-0 o +-1
                  //y =
                  vertice = Fp2dPoint(vertice.x() +
n2_90.dir_vector.dx(), vertice.y() + n2_90.dir_vector.dy());
            }else{
                  //Nos movemos a lo largo del eje Y de 1 en 1
                  //Obtenemos la x correspondiente a la Y de cada
paso
                  y += incremento;
                  x = lin_n2_90.x_at_y(y);
                  vertice = Fp2dPoint(x, y);
            }
            //Una vez desplazado el vértice creamos la recta con
dirección n1_90
            lin_n1_90 = Fp2dLine(vertice, n1_90.dir_vector);
            pasos++;
            if(pasos == MAX_PASOS){
                  escalera->Clear();
                  escalera-
>setEstadoSalida(TEstadoSalida::error);
                  return escalera;
            }
            //Para cada punto del conjunto comprobamos si se
produce intersección
            std::list<myPoint>::iterator iterador_interseccion;
            for(iterador_interseccion =
points.getIterator();iterador_interseccion!=points.getEnd();iter
ador_interseccion++){
                  //Tomamos un punto de la lista
                  myPoint punto_comprobar =
*iterador_interseccion;
                  bool modo = ((lin_n1_90.is_vertical() ||
lin_n1_90.is_horizontal())
                        && (lin_n2_90.is_vertical() ||
lin_n2_90.is_horizontal()));
                  //Se comprueba si hay intersección
                  if(AuxiliarFunctions::ContainsPoint(lin_n1_90,
punto_comprobar.p, 3, modo, lin_n2_90)){
                        /*if(ar_ab < 0){
                              if(punto_comprobar.p.y() > y_antes)
                                    break;
                        }else if(ar_ab > 0){
                              if(punto_comprobar.p.y() < y_antes)
                                    break;
                        }
                        if(iz_der < 0){//Izda
                              if(punto_comprobar.p.x() > x_antes)
                                    break;
                        }else if(iz_der > 0){//Der
                              if(punto_comprobar.p.x() < x_antes)
                                    break;
                        }*/

      if(lin_n2_90.has_on_positive_side(punto_comprobar.p))
                              break;
                        //Calculamos la intersección de manera
precisa
```

```
                        Fp2dLine linea_n1_precisa =
Fp2dLine(punto_comprobar.p, n1_90.dir_vector);
                        CGAL::Object result;
                        result =
CGAL::intersection(linea_n1_precisa,lin_n2_90);
                        const Fp2dPoint * interseccion =
CGAL::object_cast<Fp2dPoint>(&result);
                        //Añadimos el punto al "segmento"
                        escalera->Add(interseccion->x(),
interseccion->y(), 3);
                        pasos = 0;
                        if(punto_comprobar.p == maximal2.p){
                              //Si el punto al que hemos llegado
es el de destino
                              //Lo añadimos al resultado
                              escalera->Add(maximal2.p.x(),
maximal2.p.y(), 3);
                              //... y salimos
                              salir = true;
                              break;
                        }else{
                              //Si el punto al que llegamos es un
punto intermedio
                              //Añadimos el punto intermedio al
resultado
                              escalera-
>Add(punto_comprobar.p.x(), punto_comprobar.p.y(), 3);
                              //Quitamos el punto del conjunto de
puntos a comprobar
                              points.Delete(punto_comprobar);
                              //Se actualiza lin_n2_90 y el nuevo
vértice pasa a ser el punto recién comprobado
                              lin_n2_90 =
Fp2dLine(punto_comprobar.p, n2_90.dir_vector);
                              vertice = punto_comprobar.p;
                              y = punto_comprobar.p.y();
                              y_antes = y;
                              x_antes = punto_comprobar.p.x();
                              break;
                        }
                  }
            }
      }

      return escalera;

}

/**
 * Método que lleva a cabo el algoritmo de la escalera entre dos
direcciones (caso ortogonal).
 * @param d1 Dirección de origen de la escalera.
 * @param d2 Dirección de destino de la escalera.
 * @param maximal1 Punto maximal de origen.
 * @param maximal2 Punto maximal de destino.
 * @param points Conjunto de puntos.
 * @return Devuelve la escalera resultante.
 */
```

```
mySegment* AuxiliarFunctions::StairAlgorithmOrthogonal(direction
d1, direction d2, myPoint maximal1, myPoint maximal2,
ListOfPoints points){
      //Si p_1=p_2, devolver ese punto
      if(maximal1.p == maximal2.p) return NULL;

      //Ahora YO incluiría aquí un pequeño paso intermedio:
filtrar los puntos que pertenecen al paralelepipedo.
      direction n1_90, n2_90; //Sumaremos 90 a las dos
direcciones de entrada
      n1_90.degrees = d1.degrees + 90;
      n2_90.degrees = d2.degrees + 90;
      Fp2dLine lin_n1, lin_n2;
      lin_n1 = Fp2dLine(CGAL::ORIGIN, d1.dir_vector);
      lin_n2 = Fp2dLine(CGAL::ORIGIN, d2.dir_vector);
      //Obtenemos los vectores perpendiculares
      n1_90.dir_vector =
lin_n1.perpendicular(CGAL::ORIGIN).direction();
      n2_90.dir_vector =
lin_n2.perpendicular(CGAL::ORIGIN).direction();
      //Construimos los 4 lados del paralelepipedo que contendrá
los puntos de trabajo.
      Fp2dLine lin_n1_90, lin_n2_90, lin_n1_menos_90,
lin_n2_menos_90;
      lin_n1_90 = Fp2dLine(maximal2.p, n1_90.dir_vector);
      lin_n2_90 = Fp2dLine(maximal1.p, n2_90.dir_vector);
      lin_n1_menos_90 = Fp2dLine(maximal1.p, -n1_90.dir_vector);
      lin_n2_menos_90 = Fp2dLine(maximal2.p, -n2_90.dir_vector);

      ListOfPoints nueva_lista;
      std::list<myPoint>::iterator iterador_filtro;
      //VIGILAR SI INCLUYE SIEMPRE O NO LOS PUNTOS MAXIMALES -
hay algunos casos que se los salta...
      for(iterador_filtro = points.getIterator(); iterador_filtro
!= points.getEnd(); iterador_filtro++){
            if(EstaDentroDeUnPoligono(lin_n2_90, lin_n1_90,
lin_n2_menos_90, lin_n1_menos_90, iterador_filtro->p)){
                  nueva_lista.Add(iterador_filtro->p.x(),
iterador_filtro->p.y(), 3);
            }
      }
      nueva_lista.Add(maximal1.p.x(), maximal1.p.y(), 3);
      nueva_lista.Add(maximal2.p.x(), maximal2.p.y(), 3);
      //Sustituimos la lista original por la lista filtrada
      points = nueva_lista;
      mySegment* escalera = new mySegment();
      //Al añadir un punto podemos quitarlo del conjunto inicial
      escalera->setRepetidos(false);
      escalera->Add(maximal1.p.x(), maximal1.p.y(),
maximal1.radius);
      points.Delete(maximal1);
      //Trabajaremos SOLO con lin_n1_90 y lin_n2_90. Deslizaremos
n1 SOBRE n2
      bool salir = false; //Variable que controla si se sale del
algoritmo o no
      //Declaramos el vértice de la recta de arrastre.
      //El punto de comienzo es el punto1
```

```
	Fp2dPoint vertice = Fp2dPoint(maximal1.p.x(),
maximal1.p.y());
	double MAX_PASOS = 1000000;
	double pasos = 0;

	while(!salir){
		/*Considerar el rayo que sale de p_1 y tiene
dirección n_2+90.
		Sobre ese rayo deslizar el extremo de un segundo rayo
con dirección n_1+90.
		Cuando este segundo rayo se encuentre con un punto q
del conjunto:
		Si q=p_2, devolver el segmento recorrido sobre el
primer rayo,
		junto con el segmento del segundo rayo hasta q.
		FIN.
		Si no:

		Añadir a la escalera el segmento recorrido sobre el
primer rayo,
		junto con el segmento del segundo rayo hasta q.
		Actualizar p_1<--q y repetir el proceso para p_1 y
p_2.*/
		//Movemos la linea n1 sobre n2. Considerando como
casos especiales las lineas horizantales y verticales
		if(lin_n2_90.is_vertical() ||
lin_n2_90.is_horizontal()){
			//La X y la Y de los vectores directores de las
lineas H y V SIEMPRE valdrán +-0 o +-1
			vertice = Fp2dPoint(vertice.x() +
n2_90.dir_vector.dx(), vertice.y() + n2_90.dir_vector.dy());
		}
		//Una vez desplazado el vértice creamos la recta con
dirección n1_90
		lin_n1_90 = Fp2dLine(vertice, n1_90.dir_vector);
		pasos++;
		if(pasos == MAX_PASOS){
			escalera->Clear();
			escalera-
>setEstadoSalida(TEstadoSalida::error);
			return escalera;
		}
		//Para cada punto del conjunto comprobamos si se
produce intersección
		std::list<myPoint>::iterator iterador_interseccion;
		for(iterador_interseccion =
points.getIterator();iterador_interseccion!=points.getEnd();iter
ador_interseccion++){
			//Tomamos un punto de la lista
			myPoint punto_comprobar =
*iterador_interseccion;
			//Se comprueba si hay intersección
			if(AuxiliarFunctions::ContainsPoint(lin_n1_90,
punto_comprobar.p, 3, true, lin_n2_90)
				&&
!lin_n2_90.has_on_positive_side(punto_comprobar.p)){
				//Calculamos la intersección de manera
precisa
```

```
                        Fp2dLine linea_n1_precisa =
Fp2dLine(punto_comprobar.p, n1_90.dir_vector);
                        CGAL::Object result;
                        result =
CGAL::intersection(linea_n1_precisa,lin_n2_90);
                        const Fp2dPoint * interseccion =
CGAL::object_cast<Fp2dPoint>(&result);
                        //Añadimos el punto al "segmento"
                        escalera->Add(interseccion->x(),
interseccion->y(), 3);
                        pasos = 0;
                        if(punto_comprobar.p == maximal2.p){
                              //Si el punto al que hemos llegado
es el de destino
                              //Lo añadimos al resultado
                              escalera->Add(maximal2.p.x(),
maximal2.p.y(), 3);
                              //... y salimos
                              salir = true;
                              break;
                        }else{
                              //Si el punto al que llegamos es un
punto intermedio
                              //Añadimos el punto intermedio al
resultado
                              escalera-
>Add(punto_comprobar.p.x(), punto_comprobar.p.y(), 3);
                              //Quitamos el punto del conjunto de
puntos a comprobar
                              points.Delete(punto_comprobar);
                              //Se actualiza lin_n2_90 y el nuevo
vértice pasa a ser el punto recién comprobado
                              lin_n2_90 =
Fp2dLine(punto_comprobar.p, n2_90.dir_vector);
                              vertice = punto_comprobar.p;
                              break;
                        }
                  }
            }
      }

      return escalera;

}


/**
 * Función que comprueba si un punto está en una recta.
 * @param l Linea a comprobar.
 * @param p Punto a comprobar.
 * @param r Radio de error.
 * @param modo Modo de la comprobación.
 * @param ll Linea auxiliar.
 * @return Devuelve si el punto está contenido o no.
 */
bool AuxiliarFunctions::ContainsPoint(Fp2dLine l, Fp2dPoint p,
float r, bool modo, Fp2dLine ll){
      int x = 0;
```

```
	int y = 0;

	//if(!ll.has_on_negative_side(p) || !l.has_on(p)) return
false;
	if(modo && (l.is_vertical() || l.is_horizontal())){
		return l.has_on(p);
	}else{
		if(l.is_horizontal()){
			y = l.y_at_x(p.x());
			return (y < p.y()+r && p.y()-r < y);
		}else if(l.is_vertical()){
			x = l.x_at_y(p.y());
			return (x < p.x()+r && p.x()-r < x);
		}else{
			y = l.y_at_x(p.x());
			x = l.x_at_y(p.y());
			return (y < p.y()+r && p.y()-r < y) || (x <
p.x()+r && p.x()-r < x);
		}
	}
}



/**
 * Método que comprueba si un punto está dentro de un plano.
 * @param line1 Primera linea que define el plano.
 * @param line2 Segunda linea que define el plano.
 * @param point Punto a comprobar.
 * @return Devuelve si el punto está en el plano.
 */
bool AuxiliarFunctions::EstaDentroPlano(Fp2dLine line1, Fp2dLine
line2, Fp2dPoint point){

	return (line1.has_on_positive_side(point) &&
line2.has_on_positive_side(point))/* ||
		line1.has_on(point) || line2.has_on(point)*/;
}

/**
 * Función que comprueba si una dirección es ortogonal (a los
ejes).
 * @param d Dirección a comprobar.
 * @return Devuelve si la dirección es ortogonal.
 */
bool EsDirOrto(direction d){
	return (d.degrees == 0) || (d.degrees == 90) || (d.degrees
== 180) || (d.degrees == 270);
}

/**
 * Método que calcula la intersección de las escaleras.
 * @param s1 Primera escalera.
 * @param s2 Segunda escalera.
 * @param solucion Determina si se ha obtenido solución.
 * @return Devuelve la escalera con la intersección realizada.
 */
```

```
mySegment* AuxiliarFunctions::CalcularInterseccion(mySegment s1,
mySegment s2, bool *solucion)
{
	int num_i = 0;
	//Para cada sub-segmento de cada segmento miramos si hay
interseccion
	//Pasamos los segmentos a vectores
#pragma region "Pasar las listas a vectores"
	std::list<myPoint>::iterator it_seg1;
	myPoint* segmento1 = new myPoint[s1.Count()];
	int i = 0;
	for(it_seg1 = s1.getIterator(); it_seg1 != s1.getEnd();
it_seg1++)
	{
		segmento1[i] = *it_seg1;
		i++;
	}
	std::list<myPoint>::iterator it_seg2;
	myPoint* segmento2 = new myPoint[s2.Count()];
	i = 0;
	for(it_seg2 = s2.getIterator(); it_seg2 != s2.getEnd();
it_seg2++)
	{
		segmento2[i] = *it_seg2;
		i++;
	}
#pragma endregion
	mySegment* resultado = NULL;
	ListOfPoints puntos_sobrantes;
	//ListOfPoints puntos_interseccion;
	//Revisar que pasa con las intersecciones
	for(int j = 1; j < s1.Count(); j++){
		CGAL::Segment_2<CGAL::Cartesian<float>> l1 =
CGAL::Segment_2<CGAL::Cartesian<float>>(segmento1[j-1].p,
segmento1[j].p);
		Fp2dLine ll1 = Fp2dLine(segmento1[j-1].p,
segmento1[j].p);
		for(int k = 1; k < s2.Count(); k++){
			CGAL::Segment_2<CGAL::Cartesian<float>> l2 =
CGAL::Segment_2<CGAL::Cartesian<float>>(segmento2[k-1].p,
segmento2[k].p);
			Fp2dLine ll2 = Fp2dLine(segmento2[k-1].p,
segmento2[k].p);
			if(CGAL::do_intersect(l1,l2)){
				CGAL::Object result;
				result = CGAL::intersection(l1,l2);
				const Fp2dPoint * interseccion =
CGAL::object_cast<Fp2dPoint>(&result);
				if(interseccion == NULL) continue;
				//La mayoria de los puntos de mi conjunto
son enteros.
				int x_comprobar, y_comprobar;
				x_comprobar = (int)interseccion->x();
				y_comprobar = (int)interseccion->y();
				Fp2dPoint interseccion_entera =
Fp2dPoint(x_comprobar, y_comprobar);
				//Si la intersección se produce en un
punto perteneciente a la envolvente no se realiza
```

```
                        //la intersección de semiplanos escalera
                        if(segmento2[k-1].p == segmento1[j-1].p
|| segmento2[k-1].p == segmento1[j].p
                              || segmento2[k].p == segmento1[j-
1].p || segmento2[k].p == segmento1[j].p)
                              continue;
                        //Si se da UNA intersección devolvemos el
nuevo segmento
                        //ListOfPoints puntos_borrar1,
puntos_borrar2;
                        if(resultado == NULL){
                              resultado = new mySegment();
                              resultado->setRepetidos(false);
                        }
                        num_i++;
                        TDirecciones d1, d2;
                        d1 = getDireccion(ll1.direction().dx(),
ll1.direction().dy());
                        d2 = getDireccion(ll2.direction().dx(),
ll2.direction().dy());
                        bool ordenado;
                        int valor_c = CompararDirecciones(d1,
d2);
                        if(valor_c == MENOR)
                        {
                              ordenado = true;
                        }else if(valor_c == MAYOR)
                        {
                              ordenado = false;
                        }

                        //Comprobar como se mueve la comprobación
                        if(!ordenado){
                              for(int b = 0; b < s2.Count();
b++){

      if(!ll1.has_on_negative_side(segmento2[b].p) &&
!puntos_sobrantes.Contains(segmento2[b].p.x(),
segmento2[b].p.y())){
                                          resultado-
>Add(segmento2[b].p.x(), segmento2[b].p.y(),
segmento2[b].radius);
                                    }else{

      puntos_sobrantes.Add(segmento2[b].p.x(),
segmento2[b].p.y(), segmento2[b].radius);
                                          resultado-
>Delete(segmento2[b]);
                                    }
                              }

                              resultado->Add(interseccion->x(),
interseccion->y(), 3);

                              for(int a = 0; a < s1.Count();
a++){

      if(!ll2.has_on_negative_side(segmento1[a].p) &&
```

```
!puntos_sobrantes.Contains(segmento1[a].p.x(),
segmento1[a].p.y())){
                                        resultado-
>Add(segmento1[a].p.x(), segmento1[a].p.y(),
segmento1[a].radius);
                                    }else{

      puntos_sobrantes.Add(segmento1[a].p.x(),
segmento1[a].p.y(), segmento1[a].radius);
                                        resultado-
>Delete(segmento1[a]);
                                    }
                                }
                            }else{
                                for(int a = 0; a < s1.Count();
a++){

      if(!ll2.has_on_negative_side(segmento1[a].p) &&
!puntos_sobrantes.Contains(segmento1[a].p.x(),
segmento1[a].p.y())){
                                        resultado-
>Add(segmento1[a].p.x(), segmento1[a].p.y(),
segmento1[a].radius);
                                    }else{

      puntos_sobrantes.Add(segmento1[a].p.x(),
segmento1[a].p.y(), segmento1[a].radius);
                                        resultado-
>Delete(segmento1[a]);
                                    }
                                }

                                resultado->Add(interseccion->x(),
interseccion->y(), 3);

                                for(int b = 0; b < s2.Count();
b++){

      if(!ll1.has_on_negative_side(segmento2[b].p) &&
!puntos_sobrantes.Contains(segmento2[b].p.x(),
segmento2[b].p.y())){
                                        resultado-
>Add(segmento2[b].p.x(), segmento2[b].p.y(),
segmento2[b].radius);
                                    }else{

      puntos_sobrantes.Add(segmento2[b].p.x(),
segmento2[b].p.y(), segmento2[b].radius);
                                        resultado-
>Delete(segmento2[b]);
                                    }
                                }
                            }
                            //Comprobamos los puntos que sobran


                            //return resultado;
```

```
                        }
                }
        }
        if(num_i > 2){
                *(solucion) = false;
                return NULL;
        }
        //Pequeño parche para intersecciones dobles en la misma
"linea"
        if(resultado != NULL && resultado->Count() >= 4 && num_i >
1){
                resultado->setRepetidos(true);
                std::list<myPoint>::iterator it_r = resultado-
>getIterator();
                resultado->Add(it_r->p.x(), it_r->p.y(), it_r-
>radius);
        }
        return resultado;
}

/**
 * Función que comprueba si dos escaleras son "inversas".
 * @param ms1 Primera escalera.
 * @param ms2 Segunda escalera.
 * @param lp Lista de puntos.
 * @return Devuelve si son inversos o no.
 */
bool Inversos(mySegment *ms1, mySegment *ms2, ListOfPoints lp)
{
        if(ms1->Count() != ms2->Count()) return false;

        std::list<myPoint>::iterator contiene_p;

        for(contiene_p = lp.getIterator(); contiene_p !=
lp.getEnd(); contiene_p++)
        {
                if(!ms1->Contains(contiene_p->p.x(), contiene_p-
>p.y()) || !ms2->Contains(contiene_p->p.x(), contiene_p->p.y()))
return false;
        }

        std::list<myPoint>::iterator it1 = ms1->getIterator();
        std::list<myPoint>::iterator it2 = ms2->getEnd();
        it2--;

        bool ret_val = it1->p == it2->p;

        it1 = ms1->getEnd();
        it1--;
        it2 = ms2->getIterator();

        ret_val = ret_val && (it1->p == it2->p);



        return ret_val;

        /*myPoint *puntos1 = new myPoint[ms1->Count()];
```

```
	myPoint *puntos2 = new myPoint[ms2->Count()];

	std::list<myPoint>::iterator it1;
	int i = 0;
	for(it1 = ms1->getIterator(); it1 != ms1->getEnd(); it1++){
		puntos1[i] = *it1;
		i++;
	}

	std::list<myPoint>::iterator it2;
	i = 0;
	for(it2 = ms2->getIterator(); it2 != ms2->getEnd(); it2++){
		puntos2[i] = *it2;
		i++;
	}

	int j;
	for(i = 0,j = ms1->Count(); i < ms1->Count(); i++, j--)
	{
		if((puntos1[i].p.x() != puntos2[j].p.x()) ||
(puntos1[i].p.y() != puntos2[j].p.y())) return false;
	}*/

	//return true;
}


/**
 * Método que calcula la envolvente convexa.
 * @param lista_puntos Lista de puntos de los que se calculará
la envolvente.
 * @param lista_direcciones Lista de direcciones que se usarán
en la envolvente.
 * @param modo Modo de operación del método(ortogonal, todas las
direcciones, seleccionadas por el usuario).
 * @param estado_salida Estado de finalización de la operación
(se usa para cuando hay un error).
 * @return Devuelve si el punto está en el plano.
*/
mySegment* AuxiliarFunctions::CalcularEnvolvente(ListOfPoints
lista_puntos, ListOfDirections lista_direcciones, TCierre modo,
int *estado_salida){
	//Si la envolvente que piden es la normal
	if(modo == TCierre::todas){
		mySegment* res = new ListOfPoints();
		std::list<Fp2dPoint> lista_puntos_fp2d;
		std::list<myPoint>::iterator ptos;
		for(ptos = lista_puntos.getIterator(); ptos !=
lista_puntos.getEnd(); ptos++){
			Fp2dPoint p = Fp2dPoint(ptos->p.x(),ptos-
>p.y());
			lista_puntos_fp2d.push_back(p);
		}
		std::list<Fp2dPoint>	out_l;

		CGAL::ch_graham_andrew(lista_puntos_fp2d.begin(),
lista_puntos_fp2d.end(), std::back_inserter(out_l));
		//lista_puntos.Clear();
```

```
            bool primero = true;
            Fp2dPoint primero_p;
            std::list<Fp2dPoint>::iterator ptos_perimetro;
            res->setRepetidos(true);
            for(ptos_perimetro = out_l.begin(); ptos_perimetro !=
out_l.end(); ptos_perimetro++){
                  if(primero){
                        primero_p = *ptos_perimetro;
                        primero = false;
                  }
                  res->Add(ptos_perimetro->x(), ptos_perimetro-
>y(), 4);
            }

            res->Add(primero_p.x(), primero_p.y(), 4);

            return res;
      }

      //El paso1 del algoritmo va implicito en la inclusión de
las direcciones
      //Paso2: Calculamos el conjunto N={n_1,...,n_k,
n_1+180,...,n_k+180}
      //de las direcciones resultantes de sumar 90 a las de O y
ordenarlas de menor a mayor.
      ListOfDirections listaN =
lista_direcciones.CalcularConjuntoN();
      //Paso3:Para cada dirección en N, encontrar el punto
extremo correspondiente
      //(podrían ser varios; de momento podemos limitarnos a
conjuntos de puntos en los que no hay dos alineados en ninguna
de las direcciones de O, lo que garantiza que esto no pasa).
      ListOfPoints maximales;
      std::list<direction>::iterator it_dir;
      std::list<direction> lista_dir = listaN.getList();
      int max_direcciones = listaN.Count();
      direction* direcciones_v = new direction[max_direcciones];
      int i_dir = 0;
      maximales.setRepetidos(true);
      //Calculamos los máximales y a la vez vamos pasando las
direcciones a un array por comodidad
      for(it_dir = lista_dir.begin(); it_dir != lista_dir.end();
it_dir++){
            myPoint un_maximal =
lista_puntos.getMaximum(*it_dir);
            maximales.Add(un_maximal.p.x(), un_maximal.p.y(),
un_maximal.radius);
            direcciones_v[i_dir] = *it_dir;
            i_dir++;
      }
      //Se pasan los maximales a un vector por comodidad también.
      int max_puntos = maximales.Count();
      myPoint* maximales_v = new myPoint[max_puntos];
      std::list<myPoint> lp = maximales.getList();
      int i = 0;
      std::list<myPoint>::iterator it_max;
      for(it_max = lp.begin(); it_max!=lp.end(); it_max++){
            maximales_v[i] = *it_max;
```

```
			i++;
		}
		//Paso4:Para el par de direcciones n_1,n_2 (con lo que
vamos a trabajar en sentido antihorario)
		//, construir la escalera entre sus correspondientes puntos
extremos p_1 y p_2
		std::list<mySegment*> lista_segmentos;
		bool es_ortogonal = (modo == ortogonal);
		//Se harán tantas iteraciones como direcciones hay en N
		for(int j = 0; j < max_direcciones; j++){
			mySegment* resultado_un_paso;
			int origen, destino; //Calculamos la posición en el
vector de las direcciones y puntos origen y destino
			origen = j % max_direcciones;
			destino = (j + 1) % max_direcciones;
			//Si las direcciones son ortogonales o bien, en algún
momento intervienen éstas
			//nos quitamos de problemas
			//resultado_un_paso-
>setEstadoSalida(TEstadoSalida::correcto);
			if(es_ortogonal || (EsDirOrto(direcciones_v[origen])
&&EsDirOrto(direcciones_v[destino]))){
				resultado_un_paso =
AuxiliarFunctions::StairAlgorithmOrthogonal(direcciones_v[origen
], direcciones_v[destino],
				maximales_v[origen], maximales_v[destino],
lista_puntos);
			}else{
				resultado_un_paso =
AuxiliarFunctions::StairAlgorithm(direcciones_v[origen],
direcciones_v[destino],
				maximales_v[origen], maximales_v[destino],
lista_puntos);
			}
			if(resultado_un_paso != NULL)
			{
				int es = resultado_un_paso->getEstadoSalida();
				if(es == TEstadoSalida::error)
				{
					*(estado_salida) = TEstadoSalida::error;
					mySegment* ms = new mySegment();
					ms->Clear();
					return ms;
				}else if(es ==
TEstadoSalida::con_puntos_alineados){
					*(estado_salida) =
TEstadoSalida::con_puntos_alineados;
				}
			}
			//if(resultado_un_paso == NULL) continue;
			lista_segmentos.push_back(resultado_un_paso);
		}

		std::list<mySegment*>::iterator it_segmentos;
		mySegment** vector = new
mySegment*[lista_segmentos.size()];
		//Lleva la cuenta de la posición actual en el vector
		int num_i = 0;
```

```
	//Determina el numero de elementos no nulos
	int elementos_no_nulos = 0;
	System::Collections::Generic::List<int> ^lista_posiciones =
gcnew System::Collections::Generic::List<int>();
	for(it_segmentos = lista_segmentos.begin();
it_segmentos!=lista_segmentos.end(); it_segmentos++)
	{
		vector[num_i] = *it_segmentos;
		if((*it_segmentos != NULL)){
			elementos_no_nulos++;
			lista_posiciones->Add(num_i);
		}
		num_i++;
	}

	//Código añadido para el caso especial de tres o cuatro
puntos
	if((lista_puntos.Count() == 3 || lista_puntos.Count() == 4)
&& elementos_no_nulos == 2)
	{
		bool es_inverso =
Inversos(vector[lista_posiciones[0]],
vector[lista_posiciones[1]], lista_puntos);
		if(es_inverso)
		{
			(*estado_salida) = TEstadoSalida::correcto;
			mySegment *res_especial = new mySegment();
			res_especial->Clear();
			if(lista_puntos.Count() == 4)
			{
				res_especial->setRepetidos(true);

				std::list<myPoint>::iterator
it_fragmento1 = vector[lista_posiciones[0]]->getIterator();
				std::list<myPoint>::iterator
it_fragmento2 = vector[lista_posiciones[1]]->getIterator();
				it_fragmento1++;
				it_fragmento1++;
				res_especial->Add(it_fragmento1->p.x(),
it_fragmento1->p.y(), 3);
				it_fragmento1++;
				res_especial->Add(it_fragmento1->p.x(),
it_fragmento1->p.y(), 3);
				it_fragmento1++;
				res_especial->Add(it_fragmento1->p.x(),
it_fragmento1->p.y(), 3);
				it_fragmento2++;
				it_fragmento2++;
				it_fragmento2++;
				//res_especial->Add(it_fragmento2->p.x(),
it_fragmento2->p.y(), 3);
				it_fragmento2++;
				//res_especial->Add(it_fragmento2->p.x(),
it_fragmento2->p.y(), 3);
			}
			return res_especial;
		}
	}
```

```
	int tam_max_ls = (lista_segmentos.size()/2);
	lista_segmentos.clear();
	//bool ha_pasado = true;
	bool es_sol = true;
	std::list<mySegment*> lista_segmentos_atras;
	for(int ii = 0; ii < tam_max_ls; ii++){
		mySegment *s1, *s2;
		s1 = vector[ii];
		s2 = vector[ii + tam_max_ls];
		if(s1 != NULL && s2 != NULL){
			//ha_pasado = false;
			mySegment *resul;
			//resul = CalcularInterseccionNuevo(s1, s2);
			//if(modo == TCierre::ortogonal){
			//	resul = CalcularInterseccionOrto(*s1,
*s2);
			//}else

	resul=AuxiliarFunctions::CalcularInterseccion(*s1, *s2,
&es_sol);
			if(resul == NULL)
			{
				//Hay que poner la lista de puntos
ordenada
				lista_segmentos.push_back(s1);
				lista_segmentos_atras.push_back(s2);
			}else{
				lista_segmentos.push_back(resul);
			}
		}else
		{
			lista_segmentos.push_back(s1);
			lista_segmentos_atras.push_back(s2);
		}
	}
	std::list<mySegment*>::iterator it_pegar;
	for(it_pegar = lista_segmentos_atras.begin(); it_pegar !=
lista_segmentos_atras.end();it_pegar++){
		if(*it_pegar != NULL)
			lista_segmentos.push_back(*it_pegar);
	}

	mySegment *resultado = new mySegment();
	resultado->setRepetidos(true);
	std::list<mySegment*>::iterator it3;
	for(it3 = lista_segmentos.begin();
it3!=lista_segmentos.end(); it3++){
		std::list<myPoint>::iterator it4;
		if((*it3) == NULL) continue;
		std::list<myPoint> lis_pun = (*it3)->getList();
		for(it4 = lis_pun.begin(); it4!=lis_pun.end();
it4++){
			resultado->Add(it4->p.x(),it4->p.y(),it4-
>radius);
		}
		//return (resultado);
	}
```

```
	resultado->es_solucion = es_sol;
	//A lo mejor hay que comprobar que el primero y el ultimo
sean inguales.
	return (resultado);
}
```

EstadoInflado.h

```
#include "ElementoInflado.h"
#include "Types.h"

/**
 * Estructura que representa un estado de inflado de una arista.
 */
struct Estado{
	Polygon_2 poligono;/**<Polígono en proceso de inflado.*/
	direction direccion;/**<Dirección de la arista original.*/
	bool pertenece_d;/**<Determina si la arista pertenece a las
direcciones definidas por el usuario.*/
	direction tope_mas;/**<Tope positivo de inflado.*/
	direction tope_menos;/**<Tope negativo de inflado.*/
};

/*
 * Esta clase representa un paso en el inflado de un polígono.
*/
class EstadoInflado
{
private:
	int paso;
public:
	std::list<Estado> lista_elementos;
	EstadoInflado(std::list<Estado> lista_elems);
	void setPaso(int p);
	~EstadoInflado();
};
```

EstadoInflado.cpp

```
#pragma once
#include "stdafx.h"
#include "EstadoInflado.h"

/**
 * Constructor de un EstadoInflado.
 * @param lista_elems Elementos del estado.
 */
EstadoInflado::EstadoInflado(std::list<Estado> lista_elems)
{
	lista_elementos = lista_elems;
}

/**
 * Destructor de un EstadoInflado.
 */
EstadoInflado::~EstadoInflado(){
```

```
}

/**
 * Establece el paso.
 * @param p Número del paso.
 */
void EstadoInflado::setPaso(int p)
{
	this->paso = p;
}
```

ElementoInflado.h

```
#pragma once
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include "ListOfPoints.h"
#include <CGAL/Boolean_set_operations_2.h>

//typedef CGAL::Cartesian<float> K;
//Definición del Kernel usado por CGAL
struct Kernel : public CGAL::Cartesian<float> {};
//Definición de tipo polígono bidimensional
typedef CGAL::Polygon_2<Kernel> Polygon_2;
//Definición del tipo polígono con huecos
typedef CGAL::Polygon_with_holes_2<Kernel>
Polygon_with_holes_2;


/*
 * ElementoInflado - Esta clase representa un elemento con el
inflado realizado.
 * Ofrece la funcionalidad necesaria para obtener la envolvente
convexa con direcciones restringidas
 * partiendo de los paralelepípedos inflados.
*/
class ElementoInflado{
private:
	std::list<Polygon_2> lista_poligonos;
	Polygon_2 poligono_convexo;
	ListOfPoints puntos_envolvente;
public:
	ElementoInflado(ListOfPoints lop);
	void InterseccionDePoligonos();
	void AddPoligono(Polygon_2 p);
	std::list<ListOfPoints*> SegmentosResultado();
	bool Contains(Polygon_2 p);
	ListOfPoints getPuntosEnvolvente();
};
```

ElementoInflado.cpp

```
#pragma once
#include "stdafx.h"
#include "ElementoInflado.h"
#include "AuxiliarFunctions.h"
```

```
#include <CGAL/ch_graham_andrew.h>


/**
 * Constructor de la clase ElementoInflado.
 * @param lop Lista de puntos.
 */
ElementoInflado::ElementoInflado(ListOfPoints lop){
      std::list<Fp2dPoint> lista_puntos_fp2d;
      std::list<myPoint>::iterator ptos;
      for(ptos = lop.getIterator(); ptos != lop.getEnd();
ptos++){
            Fp2dPoint p = Fp2dPoint(ptos->p.x(),ptos->p.y());
            lista_puntos_fp2d.push_back(p);
      }
      std::list<Fp2dPoint>    out_l;
      //Se realiza la envolvente convexa
      CGAL::ch_graham_andrew(lista_puntos_fp2d.begin(),
lista_puntos_fp2d.end(), std::back_inserter(out_l));
      //lista_puntos.Clear();
      Fp2dPoint primero_p;
      std::list<Fp2dPoint>::iterator ptos_perimetro;
      puntos_envolvente.setRepetidos(true);
      Fp2dPoint *vector = new Fp2dPoint[out_l.size()];
      int i = 0;
      for(ptos_perimetro = out_l.begin(); ptos_perimetro !=
out_l.end(); ptos_perimetro++, i++){
            puntos_envolvente.Add(ptos_perimetro->x(),
ptos_perimetro->y(), 4);
            vector[i] = *ptos_perimetro;
      }
      //Se almacenan los puntos de la envolvente en un polígono
      poligono_convexo = Polygon_2(vector, vector +
out_l.size());
}

/**
 * Método para añadir un polígono al elemento inflado.
 * @param p Polígono a añadir.
 */
void ElementoInflado::AddPoligono(Polygon_2 p){
      if(!this->Contains(p)){
            ElementoInflado::lista_poligonos.push_back(p);
      }
}

/**
 * Método que comprueba si el elemento inflado contiene un
poliógno.
 * @param p Polígono a comprobar.
 * @return Devuelve si el polígono está o no contenido.
 */
bool ElementoInflado::Contains(Polygon_2 p){
      std::list<Polygon_2>::iterator it_contains;

      for(it_contains = lista_poligonos.begin(); it_contains !=
lista_poligonos.end(); it_contains++){
            if((*it_contains) == p) return true;
```

```
		}

		return false;
	}

	/**
	 * Método que comprueba si un polígono contiene un punto.
	 * @param pt Punto a comprobar.
	 * @param pgn_begin Inicio del vector de puntos que representa
	el polígono a comprobar.
	 * @param pgn_end Fin del vector de puntos que representa el
	polígono a comprobar.
	 * @param traits Traits usado por CGAL.
	 * @return Devuelve si el punto está o no dentro del polígono.
	 */
	bool PoligonoContienePunto(Fp2dPoint pt, Fp2dPoint *pgn_begin,
	Fp2dPoint *pgn_end, CGAL::Cartesian<float> traits)
	{
	  switch(CGAL::bounded_side_2(pgn_begin, pgn_end,pt, traits)) {
	    case CGAL::ON_BOUNDED_SIDE :
	      return true;
	    case CGAL::ON_BOUNDARY:
	      return true;
	    case CGAL::ON_UNBOUNDED_SIDE:
	      return false;
	  }
	}

	/**
	 * Método que obtiene los segmentos que pertenecen al resultado
	de la envolvente con direcciones restringidas.
	 * @return Devuelve los segmentos resultantes.
	 */
	std::list<ListOfPoints*> ElementoInflado::SegmentosResultado()
	{
		std::list<ListOfPoints*> li;
		Fp2dPoint *puntos = new Fp2dPoint[poligono_convexo.size()];
		for(int i = 0; i < poligono_convexo.size(); i++)
		{
			puntos[i] = poligono_convexo[i];
		}

		/*Fp2dPoint *puntos2 = new Fp2dPoint[union_poligonos];
		for(int i = 0; i < 33; i++)
		{
			puntos2[i] = union_poligonos[i];
		}*/
		std::list<Polygon_2>::iterator it_poligono;
		//Para cada polígono "modificado" del ElementoInflado se
	comprueba que segmentos del mismo
		//pertenecen al polígono original. Si pertenecen a dicho
	polígono están en el resultado
		for(it_poligono = lista_poligonos.begin(); it_poligono !=
	lista_poligonos.end(); it_poligono++)
		{
			Polygon_2 P = (*it_poligono);
			int tam_max = P.size();
			for(int i = 0; i < tam_max; i++)
```

```
		{
			int next_i = (i + 1) % tam_max;
			bool es_resultado = PoligonoContienePunto(P[i],
puntos, puntos+poligono_convexo.size(),CGAL::Cartesian<float>())
&&
				PoligonoContienePunto(P[next_i], puntos,
puntos+poligono_convexo.size(),CGAL::Cartesian<float>())/* &&
				PoligonoContienePunto(P[i], puntos2,
puntos2+union_poligonos.size(),CGAL::Cartesian<float>()) &&
				PoligonoContienePunto(P[next_i], puntos2,
puntos2+union_poligonos.size(),CGAL::Cartesian<float>())*/;
			if(!es_resultado) continue;
			ListOfPoints *lop = new ListOfPoints();
			lop->setRepetidos(true);
			lop->Add(P[i].x(), P[i].y(), 3);
			lop->Add(P[next_i].x(), P[next_i].y(), 3);
			/*if(lop->Count() == 1)
				printf("");*/
			li.push_back(lop);
		}
	}
	return li;
}


/**
 * Método que modifica los polígonos del ElementoInflado
añadiendo los puntos de intersección.
 */
void ElementoInflado::InterseccionDePoligonos()
{
	//Para cada par de poligonos se comprueba la interseccion y
se actualiza el poligono.
	std::list<Polygon_2> nueva_lista_pol;
	std::list<Polygon_2>::iterator it_poligono;
	//System::Collections::Generic::List<int>
^poligonos_sustituir = gcnew
System::Collections::Generic::List<int>();
	int i = 0, j = 0;
	for(it_poligono = lista_poligonos.begin(); it_poligono !=
lista_poligonos.end(); it_poligono++,i++)
	{
		Polygon_2 P = *it_poligono;
		std::list<Polygon_2>::iterator it_poligono2;
		for(it_poligono2 = lista_poligonos.begin();
it_poligono2 != lista_poligonos.end(); it_poligono2++, j++)
		{
			Polygon_2 Q = *it_poligono2;
			if(i <= j) continue;
			if(P == Q) continue;
			if(!CGAL::do_intersect(P, Q)) continue;

			Polygon_2 nuevoP;
			it_poligono->clear();
			for(int i = 0; i < P.container().size(); i++){
				int next_i = (i + 1) %
P.container().size();
```

```
                        CGAL::Segment_2<CGAL::Cartesian<float>>
sP = CGAL::Segment_2<CGAL::Cartesian<float>>(P[i], P[next_i]);
                        bool hay_interseccion = false;
                        nuevoP.push_back(P[i]);
                        it_poligono->push_back(P[i]);
                        for(int j = 0; j < Q.container().size();
j++){
                              int next_j = (j + 1) %
Q.container().size();

      CGAL::Segment_2<CGAL::Cartesian<float>> sQ =
CGAL::Segment_2<CGAL::Cartesian<float>>(Q[j], Q[next_j]);
                              if(CGAL::do_intersect(sP, sQ)){
                                    CGAL::Object result;
                                    result =
CGAL::intersection(sP,sQ);
                                    const Fp2dPoint *
interseccion = CGAL::object_cast<Fp2dPoint>(&result);
                                    if(interseccion == NULL ||
(*interseccion) == P[i] || (*interseccion) == P[next_i])
continue;

      nuevoP.push_back(*interseccion);
                                    it_poligono-
>push_back(*interseccion);
                              }
                        }
                  }

                  Polygon_2 nuevoQ;
                  it_poligono2->clear();
                  for(int i = 0; i < Q.container().size(); i++){
                        int next_i = (i + 1) %
Q.container().size();
                        CGAL::Segment_2<CGAL::Cartesian<float>>
sQ = CGAL::Segment_2<CGAL::Cartesian<float>>(Q[i], Q[next_i]);
                        bool hay_interseccion = false;
                        nuevoQ.push_back(Q[i]);
                        it_poligono2->push_back(Q[i]);
                        for(int j = 0; j < P.container().size();
j++){
                              int next_j = (j + 1) %
P.container().size();

      CGAL::Segment_2<CGAL::Cartesian<float>> sP =
CGAL::Segment_2<CGAL::Cartesian<float>>(P[j], P[next_j]);
                              if(CGAL::do_intersect(sP, sQ)){
                                    CGAL::Object result;
                                    result =
CGAL::intersection(sP,sQ);
                                    const Fp2dPoint *
interseccion = CGAL::object_cast<Fp2dPoint>(&result);
                                    if(interseccion == NULL ||
(*interseccion) == Q[i] || (*interseccion) == Q[next_i])
continue;

      nuevoQ.push_back(*interseccion);
```

```
                                        it_poligono2-
>push_back(*interseccion);
                                  }
                            }
                      }

                      /*nueva_lista_pol.push_back(nuevoP);
                      if(!poligonos_sustituir->Contains(i))
                            poligonos_sustituir->Add(i);
                      nueva_lista_pol.push_back(nuevoQ);
                      if(!poligonos_sustituir->Contains(j))
                            poligonos_sustituir->Add(j);*/
                }
          }
          /*int k;
          for(it_poligono = lista_poligonos.begin(), k = 0;
it_poligono != lista_poligonos.end(); it_poligono++, k++)
          {
          }*/

          //lista_poligonos = nueva_lista_pol;
}

/**
 * Método que obtiene los puntos que forman la envolvente del
polígono original.
 * @return Devuelve los puntos de la envolvente.
 */
ListOfPoints ElementoInflado::getPuntosEnvolvente(){
          return puntos_envolvente;
}
```

Inflador.h

```
#pragma once
#include "EstadoInflado.h"
#include "ElementoInflado.h"
#include "ListOfDirections.h"
#include "ListOfPoints.h"

#define PRECISION 5

/*
 * Esta clase representa un "inflador" de polígonos. Dado un
conjunto de puntos calcula el polígono que forma.
 * Usando ese polígono y las direcciones de entrada se van
inflando las aristas del polígono hasta obtener el resultado
 */
class Inflador
{
private:
          ElementoInflado *estado_final;
          std::list<EstadoInflado> lista_estados;
          EstadoInflado *ultimo_paso;
          direction *direcciones;
```

```
	ListOfDirections lista_direcciones;
	ListOfPoints puntos;

	std::list<ListOfPoints*> lista_segmentos;
public:
	int paso_actual;
	Inflador(ListOfDirections lod, ListOfPoints lop);
	~Inflador();
	void Inflar();
	//void ObtenerResultado();
	void PintaPaso(int paso, System::Drawing::Graphics ^g,
System::Drawing::Color color, System::Drawing::Color colorCH);
	bool inflado_completo;
};
```

Inflador.cpp

```
#pragma once
#include "stdafx.h"
#include "Inflador.h"
#include "stdio.h"
#define PI (3.14159265358979323846)

/**
 * Función que redondea un número al entero más próximo.
 * @param numero Número a redondear.
 * @return Número redondeado.
 */
float RedondearAlEnteroMasProximo(float numero){
	double d = (double)numero;
	System::String ^s = d.ToString();
	int coma = s->IndexOf(',');
	if(coma == -1) return numero;
	System::String ^ss = "0," + s->Substring(coma + 1);
	float f = float::Parse(ss);
	int modificador;
	if(f < 0.0005) modificador = 0;
	else if(f > 0.999) modificador = 1;
	float ff = float::Parse(s->Substring(0, coma));
	ff += modificador;
	return ff;
}

/**
 * Constructor de la clase Inflador.
 * @param lod Lista de direcciones entrada del problema.
 * @param lop Lista de puntos entrada del problema.
 */
Inflador::Inflador(ListOfDirections lod, ListOfPoints lop)
{
	this->lista_direcciones = lod;
	direcciones = new direction[lod.Count()];
	std::list<direction>::iterator it_dir;
	std::list<direction> lista_d_aux =
lista_direcciones.getList();
	int i = 0;
```

```
	for(it_dir = lista_d_aux.begin(); it_dir != 
lista_d_aux.end(); it_dir++, i++)
	{
		direcciones[i] = *it_dir;
	}
	this->puntos = lop;
	this->estado_final = new ElementoInflado(puntos);
	this->ultimo_paso = NULL;
	paso_actual = 0;
	inflado_completo = false;
}

/**
 * Método que pinta un paso del inflado.
 * @param paso Paso a dibujar.
 * @param g Objeto Graphics en el que dibujar.
 * @param color Color de dibujo de los polígonos o del
resultado.
 * @param colorCH Color de dibujo del polígono original.
 */
void Inflador::PintaPaso(int paso, System::Drawing::Graphics ^g,
System::Drawing::Color color, System::Drawing::Color colorCH)
{
	//Se crea una brocha de dibujo de colorCH
	System::Drawing::Pen ^pCH = gcnew
System::Drawing::Pen(colorCH);
	std::list<myPoint>::iterator it_puntos;
	int j = 0;
	int max_pj = puntos.Count();
	//Lo primero que se hace es dibujar la envolvente convexa
	for(it_puntos = puntos.getIterator(); it_puntos != 
puntos.getEnd(); it_puntos++, j++)
	{
		int next_j = (j + 1) % max_pj;
		std::list<myPoint>::iterator sig;
		if(next_j == 0)
		{
			sig = puntos.getIterator();
		}else{
			it_puntos++;
			sig = it_puntos;
			it_puntos--;
		}

		g->DrawLine(pCH, it_puntos->p.x(), it_puntos->p.y(),
sig->p.x(), sig->p.y());
	}
	//El paso -1 indica un paso inicial sin nada que dibujar
	if(paso == -1) return;
	//Si el paso no es el "final" se dibujan los polígonos del
estado correspondiente
	if(paso != lista_estados.size() - 1)
	{
		std::list<EstadoInflado>::iterator it_estados;
		int c_paso = 0;
		EstadoInflado *e_inflado;
		for(it_estados = lista_estados.begin(); it_estados !=
lista_estados.end(); it_estados++, c_paso++)
```

```
		{
			e_inflado = &(*it_estados);
			if(c_paso == paso) break;
		}
		std::list<Estado>::iterator it_estado;
		System::Drawing::Pen ^p = gcnew
System::Drawing::Pen(color);
		for(it_estado = e_inflado->lista_elementos.begin();
it_estado != e_inflado->lista_elementos.end(); it_estado++)
		{
			for(int i = 0; i < it_estado->poligono.size();
i++)
			{
				int next_i = (i + 1) % it_estado-
>poligono.size();
				g->DrawLine(p, it_estado-
>poligono[i].x(),it_estado->poligono[i].y(), it_estado-
>poligono[next_i].x(),it_estado->poligono[next_i].y());
			}
		}

	}else
	{
		//Si es el estado final se dibujan los segmentos del
resultado
		std::list<ListOfPoints*>::iterator it_segmentos;
		for(it_segmentos = lista_segmentos.begin();
it_segmentos != lista_segmentos.end(); it_segmentos++)
		{
			std::list<myPoint>::iterator it_p =
(*it_segmentos)->getIterator();
			myPoint p1, p2;
			p1 = *it_p;
			it_p++;
			p2 = *it_p;
			System::Drawing::Pen ^p = gcnew
System::Drawing::Pen(color);
			g->DrawLine(p, p1.p.x(),p1.p.y(),
p2.p.x(),p2.p.y());
		}
	}
}

/**
 * Destructor del inflador
 */
Inflador::~Inflador()
{
	ultimo_paso->~EstadoInflado();
}

/**
 * Obtiene el ángulo de un vector de dirección dado.
 * @param dir Vector dirección.
 * @return Devuelve el ángulo.
 */
double ObtenerAnguloVector(Fp2dDirection dir)
{
```

```
	float x, y;
	x = dir.dx();
	y = dir.dy();

	double rad = System::Math::Atan2(y, x);
	double grados = (rad * 180.0) / PI;

	if(grados < 0) grados = 360 + grados;


	return grados;
}

/**
 * Obtiene un vector dirección dado un ángulo.
 * @param deg Ángulo en grados.
 * @return Vector dirección.
 */
direction ObtenerVectorAngulo(int deg)
{
	float x, y;
	double radianes;
	//Pasamos de grados a radianes
	radianes = (deg * 2* PI)/360;
	//Obtenemos la 'x' y la 'y' de la recta
	y = (float)sin(radianes);
	x = (float)cos(radianes);
	//Apaño debido a los decimales
	if((float)abs(x) < 0.0000000001) x = 0;
	if((float)abs(y) < 0.0000000001) y = 0;
	//Creamos la linea que pasa por el punto
	Fp2dPoint punto = Fp2dPoint(x, y);
	Fp2dLine linea_deg = Fp2dLine(CGAL::ORIGIN, punto);
	direction ret_dir;
	ret_dir.degrees = deg;
	ret_dir.dir_vector =linea_deg.direction();

	return ret_dir;
}


/**
 * Método que lleva a cabo un paso del inflado.
 */
void Inflador::Inflar()
{
	bool fin_inflado = true;
	//Primer paso del inflado
	if(ultimo_paso == NULL){
		//Obtenemos los puntos de la envolvente y los pasamos
a un vector
		ListOfPoints envolvente = estado_final-
>getPuntosEnvolvente();
		Fp2dPoint *vector_puntos = new
Fp2dPoint[envolvente.Count()];
		std::list<myPoint>::iterator it_puntos;
		int k = 0;
```

```
            for(it_puntos = envolvente.getIterator(); it_puntos
!= envolvente.getEnd(); it_puntos++, k++)
            {
                  vector_puntos[k] = it_puntos->p;
            }
            //Para cada direccione averiguar los topes (o mirar
si esta en la lista de direcciones [solo entre 0 y 180])
            //y hacer el primer paso
            int tam_max = envolvente.Count();
            std::list<Estado> lest;
            for(int i = 0; i < tam_max; i++)
            {
                  int next_i = (i+1) % tam_max;
                  Fp2dLine linea_auxiliar =
Fp2dLine(vector_puntos[i], vector_puntos[next_i]);
                  Fp2dDirection dir_poligono =
linea_auxiliar.direction();
                  //float angulo =
CGAL::compare_angle_with_x_axis(
                  if(lista_direcciones.Contains(dir_poligono))
                  {
                        //Estos segmentos tienen alguna de las
direcciones del conjunto y por lo tanto no hara falta inflarlos
mas
                        Estado est;
                        est.direccion.dir_vector = dir_poligono;
                        est.direccion.degrees =
ObtenerAnguloVector(dir_poligono);
                        est.pertenece_d = true;
                        est.poligono.push_back(vector_puntos[i]);

      est.poligono.push_back(vector_puntos[next_i]);
                        est.tope_mas.degrees = -1;
                        est.tope_menos.degrees = -1;
                        lest.push_back(est);
                  }else{
                        int j = 0;
                        for(;;j++)
                        {
                              int next_j = (j+1) %
lista_direcciones.Count();
                              //if(direcciones[j].degrees > 180)
break;
                              if(j > lista_direcciones.Count())
break;

      if(dir_poligono.counterclockwise_in_between(direcciones[j].
dir_vector, direcciones[next_j].dir_vector))
                              {
                                    //inflado_completo = false;
                                    Estado est;
                                    est.pertenece_d = false;
                                    est.direccion.dir_vector =
dir_poligono;
                                    est.direccion.degrees =
ObtenerAnguloVector(dir_poligono);
                                    est.tope_menos.degrees =
direcciones[j].degrees;
```

```
                                    est.tope_mas.degrees =
direcciones[next_j].degrees;
                                    est.tope_menos.dir_vector =
direcciones[j].dir_vector;
                                    est.tope_mas.dir_vector =
direcciones[next_j].dir_vector;
                                    //Se crea el polígono
correspondiente
                                    //Se mira cuanto se puede
inflar el poligono.
                                    direction dmas, dmenos;
                                    int angulo_menos =
(int)System::Math::Max((int)(est.direccion.degrees -
((paso_actual+1) * PRECISION)), (int)est.tope_menos.degrees);
                                    dmenos =
ObtenerVectorAngulo(angulo_menos);
                                    int angulo_mas =
(int)System::Math::Min((int)(est.direccion.degrees +
((paso_actual+1) * PRECISION)), (int)est.tope_mas.degrees);
                                    dmas =
ObtenerVectorAngulo(angulo_mas);
                                    fin_inflado = false;
                                    //Obtenemos el primer punto
intermedio
                                    Fp2dLine linea_dmenos,
linea_dmas;
                                    linea_dmenos =
Fp2dLine(vector_puntos[i], dmenos.dir_vector);
                                    linea_dmas =
Fp2dLine(vector_puntos[next_i], dmas.dir_vector);
                                    CGAL::Object result;
                                    result =
CGAL::intersection(linea_dmenos,linea_dmas);
                                    const Fp2dPoint * intermedio1
= CGAL::object_cast<Fp2dPoint>(&result);
                                    //Obtenemos el segundo
                                    linea_dmenos =
Fp2dLine(vector_puntos[next_i], dmenos.dir_vector);
                                    linea_dmas =
Fp2dLine(vector_puntos[i], dmas.dir_vector);
                                    CGAL::Object result2;
                                    result2 =
CGAL::intersection(linea_dmenos,linea_dmas);
                                    const Fp2dPoint * intermedio2
= CGAL::object_cast<Fp2dPoint>(&result2);

      est.poligono.push_back(vector_puntos[i]);

      est.poligono.push_back(*intermedio1);

      est.poligono.push_back(vector_puntos[next_i]);

      est.poligono.push_back(*intermedio2);
                                    lest.push_back(est);
                                    break;
                                }
                            }
                        }
```

```
                }
                //Guardar estado
                ultimo_paso = new EstadoInflado(lest);
                ultimo_paso->setPaso(paso_actual);
                paso_actual++;
                lista_estados.push_back(*ultimo_paso);
        }else{
                //Para cada poligono inflar
                std::list<Estado>::iterator it_estado;
                std::list<Estado> nuevo_estado;
                //bool parar = true;
                for(it_estado = ultimo_paso->lista_elementos.begin();
it_estado != ultimo_paso->lista_elementos.end(); it_estado++)
                {
                        Estado est = *it_estado;
                        //Si en el primer paso no se ha creado poligono
es porque la dirección estaba en el conjunto
                        if(!est.pertenece_d)
                        {
                                direction dmas, dmenos;
                                int angulo_menos =
(int)System::Math::Max((int)(est.direccion.degrees -
((paso_actual+1) * PRECISION)), (int)est.tope_menos.degrees);
                                dmenos =
ObtenerVectorAngulo(angulo_menos);
                                int angulo_mas =
(int)System::Math::Min((int)(est.direccion.degrees +
((paso_actual+1) * PRECISION)), (int)est.tope_mas.degrees);
                                dmas = ObtenerVectorAngulo(angulo_mas);
                                if(angulo_menos != est.tope_menos.degrees
|| angulo_mas != est.tope_mas.degrees)
                                {
                                        fin_inflado = false;
                                }
                                //Obtenemos el primer punto intermedio
                                Fp2dLine linea_dmenos, linea_dmas;
                                Fp2dPoint punto_origen, punto_destino;
                                punto_origen = est.poligono[0];
                                punto_destino = est.poligono[2];
                                linea_dmenos = Fp2dLine(punto_origen,
dmenos.dir_vector);
                                linea_dmas = Fp2dLine(punto_destino,
dmas.dir_vector);
                                CGAL::Object result;
                                result =
CGAL::intersection(linea_dmenos,linea_dmas);
                                const Fp2dPoint * intermedio1 =
CGAL::object_cast<Fp2dPoint>(&result);
                                //Obtenemos el segundo
                                linea_dmenos = Fp2dLine(punto_destino,
dmenos.dir_vector);
                                linea_dmas = Fp2dLine(punto_origen,
dmas.dir_vector);
                                CGAL::Object result2;
                                result2 =
CGAL::intersection(linea_dmenos,linea_dmas);
                                const Fp2dPoint * intermedio2 =
CGAL::object_cast<Fp2dPoint>(&result2);
```

```
                        est.poligono.clear();
                        if(!fin_inflado){

      est.poligono.push_back(punto_origen);

      est.poligono.push_back(*intermedio1);

      est.poligono.push_back(punto_destino);

      est.poligono.push_back(*intermedio2);
                        }else{

      est.poligono.push_back(punto_origen);
                              Fp2dPoint p1 =
Fp2dPoint(RedondearAlEnteroMasProximo(intermedio1->x()),
RedondearAlEnteroMasProximo(intermedio1->y()));
                              Fp2dPoint p2 =
Fp2dPoint(RedondearAlEnteroMasProximo(intermedio2->x()),
RedondearAlEnteroMasProximo(intermedio2->y()));
                              est.poligono.push_back(p1);

      est.poligono.push_back(punto_destino);
                              est.poligono.push_back(p2);
                        }
                  }

                  nuevo_estado.push_back(est);
            }
            //Guardar estado
            ultimo_paso = new EstadoInflado(nuevo_estado);
            ultimo_paso->setPaso(paso_actual);
            paso_actual++;
            lista_estados.push_back(*ultimo_paso);

            //Comprobar si es estado final

            this->inflado_completo = fin_inflado;
            if(inflado_completo)
            {
                  std::list<Estado>::iterator it_estado;
                  std::list<ListOfPoints*>
segmentos_pertenecientes;
                  std::list<Estado> nuevo_estado;
                  bool parar = true;
                  for(it_estado = ultimo_paso-
>lista_elementos.begin(); it_estado != ultimo_paso-
>lista_elementos.end(); it_estado++)
                  {
                        Estado est = *it_estado;
                        if(!est.pertenece_d)
                        {
                              this->estado_final-
>AddPoligono(est.poligono);
                        }else{
                              ListOfPoints *lp = new
ListOfPoints();
                              lp->Add(est.poligono[0].x(),
est.poligono[0].y(), 3);
```

```
                                    lp->Add(est.poligono[1].x(),
est.poligono[1].y(), 3);

      segmentos_pertenecientes.push_back(lp);
                        }
                  }
                  //try{
                        this->estado_final-
>InterseccionDePoligonos();
                  /*}catch(System::Exception ^ex)
                  {
                        System::Console::WriteLine(ex->Message);
                  }*/
                  this->lista_segmentos = this->estado_final-
>SegmentosResultado();
                  std::list<ListOfPoints*>::iterator it_lop;
                  for(it_lop = segmentos_pertenecientes.begin();
it_lop != segmentos_pertenecientes.end(); it_lop++)
                  {
                        lista_segmentos.push_back((*it_lop));
                  }

            }
      }
}
```

DirectionsControlControl.h

```
#pragma once

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
using namespace System::Collections::Generic;


namespace DirectionsControl {

	/// <summary>
	/// Summary for DirectionsControlControl
	/// </summary>
	///
	/// WARNING: If you change the name of this class, you will
need to change the
	///          'Resource File Name' property for the managed
resource compiler tool
	///          associated with all .resx files this class
depends on.  Otherwise,
	///          the designers will not be able to interact
properly with localized
	///          resources associated with this form.
	public ref class DirectionsControlControl : public
System::Windows::Forms::UserControl
	{
	public:
		List<int> ^lista_angulos;
		DirectionsControlControl(void)
		{
			InitializeComponent();
			//
			//TODO: Add the constructor code here
			//
			lista_angulos = gcnew List<int>();
			rDireccion->Checked = true;
			tbDestino->Enabled = false;
			tbOrigen->Enabled = false;
		}

	protected:
		/// <summary>
		/// Clean up any resources being used.
		/// </summary>
		~DirectionsControlControl()
		{
			if (components)
			{
				delete components;
			}
		}
	private: System::Windows::Forms::Panel^  panel1;
```

```
        private: System::Windows::Forms::Label^  label1;
        private: System::Windows::Forms::TextBox^  tbOrigen;
        private: System::Windows::Forms::TextBox^  tbDestino;
        private: System::Windows::Forms::Label^  lbl2;
        private: System::Windows::Forms::Label^  label2;
        private: System::Windows::Forms::RadioButton^  rIntervalo;
        private: System::Windows::Forms::RadioButton^  rDireccion;


        private: System::Windows::Forms::Label^  label3;
        private: System::Windows::Forms::TextBox^  tbDireccion;
        private: System::Windows::Forms::Button^  btAdd;
        private: System::Windows::Forms::Button^  btReiniciar;
        private: System::Windows::Forms::Button^  btQuitar;



        protected:

        private:
              /// <summary>
              /// Required designer variable.
              /// </summary>
              System::ComponentModel::Container^ components;

#pragma region Windows Form Designer generated code
              /// <summary>
              /// Required method for Designer support - do not
modify
              /// the contents of this method with the code editor.
              /// </summary>
              void InitializeComponent(void)
              {
                    this->panel1 = (gcnew
System::Windows::Forms::Panel());
                    this->label1 = (gcnew
System::Windows::Forms::Label());
                    this->tbOrigen = (gcnew
System::Windows::Forms::TextBox());
                    this->tbDestino = (gcnew
System::Windows::Forms::TextBox());
                    this->lbl2 = (gcnew
System::Windows::Forms::Label());
                    this->label2 = (gcnew
System::Windows::Forms::Label());
                    this->rIntervalo = (gcnew
System::Windows::Forms::RadioButton());
                    this->rDireccion = (gcnew
System::Windows::Forms::RadioButton());
                    this->label3 = (gcnew
System::Windows::Forms::Label());
                    this->tbDireccion = (gcnew
System::Windows::Forms::TextBox());
                    this->btAdd = (gcnew
System::Windows::Forms::Button());
                    this->btReiniciar = (gcnew
System::Windows::Forms::Button());
```

```
                    this->btQuitar = (gcnew
System::Windows::Forms::Button());
                    this->SuspendLayout();
                    //
                    // panel1
                    //
                    this->panel1->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>(((System::Wind
ows::Forms::AnchorStyles::Bottom |
System::Windows::Forms::AnchorStyles::Left)
                                |
System::Windows::Forms::AnchorStyles::Right));
                    this->panel1->BorderStyle =
System::Windows::Forms::BorderStyle::Fixed3D;
                    this->panel1->Location =
System::Drawing::Point(3, 113);
                    this->panel1->Name = L"panel1";
                    this->panel1->Size = System::Drawing::Size(300,
300);
                    this->panel1->TabIndex = 0;
                    this->panel1->Paint += gcnew
System::Windows::Forms::PaintEventHandler(this,
&DirectionsControlControl::panel1_Paint);
                    this->panel1->MouseClick += gcnew
System::Windows::Forms::MouseEventHandler(this,
&DirectionsControlControl::panel1_MouseClick);
                    //
                    // label1
                    //
                    this->label1->AutoSize = true;
                    this->label1->Location =
System::Drawing::Point(3, 17);
                    this->label1->Name = L"label1";
                    this->label1->Size = System::Drawing::Size(51,
13);
                    this->label1->TabIndex = 1;
                    this->label1->Text = L"Intervalo:";
                    //
                    // tbOrigen
                    //
                    this->tbOrigen->Location =
System::Drawing::Point(84, 10);
                    this->tbOrigen->Name = L"tbOrigen";
                    this->tbOrigen->Size =
System::Drawing::Size(54, 20);
                    this->tbOrigen->TabIndex = 0;
                    //
                    // tbDestino
                    //
                    this->tbDestino->Location =
System::Drawing::Point(183, 10);
                    this->tbDestino->Name = L"tbDestino";
                    this->tbDestino->Size =
System::Drawing::Size(54, 20);
                    this->tbDestino->TabIndex = 1;
                    this->tbDestino->TextChanged += gcnew
System::EventHandler(this,
&DirectionsControlControl::tbDestino_TextChanged);
```

```
                    //
                    // lbl2
                    //
                    this->lbl2->AutoSize = true;
                    this->lbl2->Location =
System::Drawing::Point(59, 17);
                    this->lbl2->Name = L"lbl2";
                    this->lbl2->Size = System::Drawing::Size(19,
13);
                    this->lbl2->TabIndex = 4;
                    this->lbl2->Text = L"de";
                    this->lbl2->Click += gcnew
System::EventHandler(this,
&DirectionsControlControl::label2_Click);
                    //
                    // label2
                    //
                    this->label2->AutoSize = true;
                    this->label2->Location =
System::Drawing::Point(144, 17);
                    this->label2->Name = L"label2";
                    this->label2->Size = System::Drawing::Size(33,
13);
                    this->label2->TabIndex = 5;
                    this->label2->Text = L"hasta";
                    //
                    // rIntervalo
                    //
                    this->rIntervalo->AutoSize = true;
                    this->rIntervalo->Location =
System::Drawing::Point(268, 17);
                    this->rIntervalo->Name = L"rIntervalo";
                    this->rIntervalo->Size =
System::Drawing::Size(14, 13);
                    this->rIntervalo->TabIndex = 6;
                    this->rIntervalo->TabStop = true;
                    this->rIntervalo->UseVisualStyleBackColor =
true;
                    this->rIntervalo->CheckedChanged += gcnew
System::EventHandler(this,
&DirectionsControlControl::radioButton1_CheckedChanged);
                    //
                    // rDireccion
                    //
                    this->rDireccion->AutoSize = true;
                    this->rDireccion->Location =
System::Drawing::Point(268, 54);
                    this->rDireccion->Name = L"rDireccion";
                    this->rDireccion->Size =
System::Drawing::Size(14, 13);
                    this->rDireccion->TabIndex = 7;
                    this->rDireccion->TabStop = true;
                    this->rDireccion->UseVisualStyleBackColor =
true;
                    this->rDireccion->CheckedChanged += gcnew
System::EventHandler(this,
&DirectionsControlControl::rDireccion_CheckedChanged);
                    //
```

```
                    // label3
                    //
                    this->label3->AutoSize = true;
                    this->label3->Location =
System::Drawing::Point(3, 54);
                    this->label3->Name = L"label3";
                    this->label3->Size = System::Drawing::Size(55,
13);
                    this->label3->TabIndex = 8;
                    this->label3->Text = L"Dirección:";
                    //
                    // tbDireccion
                    //
                    this->tbDireccion->Location =
System::Drawing::Point(84, 47);
                    this->tbDireccion->Name = L"tbDireccion";
                    this->tbDireccion->Size =
System::Drawing::Size(153, 20);
                    this->tbDireccion->TabIndex = 2;
                    //
                    // btAdd
                    //
                    this->btAdd->Location =
System::Drawing::Point(6, 84);
                    this->btAdd->Name = L"btAdd";
                    this->btAdd->Size = System::Drawing::Size(75,
23);
                    this->btAdd->TabIndex = 3;
                    this->btAdd->Text = L"Añadir";
                    this->btAdd->UseVisualStyleBackColor = true;
                    this->btAdd->Click += gcnew
System::EventHandler(this,
&DirectionsControlControl::btAdd_Click);
                    //
                    // btReiniciar
                    //
                    this->btReiniciar->Location =
System::Drawing::Point(207, 84);
                    this->btReiniciar->Name = L"btReiniciar";
                    this->btReiniciar->Size =
System::Drawing::Size(75, 23);
                    this->btReiniciar->TabIndex = 5;
                    this->btReiniciar->Text = L"Reiniciar";
                    this->btReiniciar->UseVisualStyleBackColor =
true;
                    this->btReiniciar->Click += gcnew
System::EventHandler(this,
&DirectionsControlControl::button1_Click);
                    //
                    // btQuitar
                    //
                    this->btQuitar->Location =
System::Drawing::Point(108, 84);
                    this->btQuitar->Name = L"btQuitar";
                    this->btQuitar->Size =
System::Drawing::Size(75, 23);
                    this->btQuitar->TabIndex = 4;
                    this->btQuitar->Text = L"Quitar";
```

```
                    this->btQuitar->UseVisualStyleBackColor = true;
                    this->btQuitar->Click += gcnew
System::EventHandler(this,
&DirectionsControlControl::btQuitar_Click);
                    //
                    // DirectionsControlControl
                    //
                    this->AutoScaleDimensions =
System::Drawing::SizeF(6, 13);
                    this->AutoScaleMode =
System::Windows::Forms::AutoScaleMode::Font;
                    this->BorderStyle =
System::Windows::Forms::BorderStyle::Fixed3D;
                    this->Controls->Add(this->btQuitar);
                    this->Controls->Add(this->btReiniciar);
                    this->Controls->Add(this->btAdd);
                    this->Controls->Add(this->tbDireccion);
                    this->Controls->Add(this->label3);
                    this->Controls->Add(this->rDireccion);
                    this->Controls->Add(this->rIntervalo);
                    this->Controls->Add(this->label2);
                    this->Controls->Add(this->lbl2);
                    this->Controls->Add(this->tbDestino);
                    this->Controls->Add(this->tbOrigen);
                    this->Controls->Add(this->label1);
                    this->Controls->Add(this->panel1);
                    this->Name = L"DirectionsControlControl";
                    this->Size = System::Drawing::Size(306, 419);
                    this->ResumeLayout(false);
                    this->PerformLayout();

                }
#pragma endregion

      private: System::Void
DirectionsControlControl_Load(System::Object^  sender,
System::EventArgs^  e) {
                    }
      private: System::Void label1_Click(System::Object^  sender,
System::EventArgs^  e) {
                    }
      private: System::Void tbDestino_TextChanged(System::Object^
sender, System::EventArgs^  e) {
                    }
private: System::Void label2_Click(System::Object^  sender,
System::EventArgs^  e) {
             }

private: System::Void panel1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {
                    System::Drawing::Graphics ^g = e->Graphics;

                    Pen ^p = gcnew Pen(Color::Blue);
                    Pen ^red_p = gcnew Pen(Color::Red);
                    Pen ^green_p = gcnew Pen(Color::Green, 3);
                    PointF ^centro = gcnew PointF(panel1->Width/2,
panel1->Height/2);
                    int r = 250;
```

```
                    centro->X -= r/2;
                    centro->Y -= r/2;
                     //int x = (int)panel1->Width - ini;
                    //Axis and circunference are drawn
                    g->DrawEllipse(p,centro->X,centro-
>Y,(float)r,(float)r);
                    g->DrawLine(green_p,(float)0, (float)panel1-
>Height/2, (float)panel1->Width, (float)panel1->Height/2);
                    g->DrawLine(green_p,(float)panel1-
>Height/2,(float)0, (float)panel1->Height/2, (float)panel1-
>Height);
                    System::Drawing::Drawing2D::Matrix ^matriz =
gcnew System::Drawing::Drawing2D::Matrix(1,0,0,-1,panel1-
>Width/2,panel1->Height/2);
                    g->Transform = matriz;
                    for(int i = 0; i < lista_angulos->Count; i++){
                          float xx, yy;
                          double radianes;
                         //Pasamos de grados a radianes
                          radianes = (lista_angulos[i] *
2*Math::PI)/360;
                         //Obtenemos la 'x' y la 'y' de la recta
                         yy = (float)Math::Sin(radianes);
                         xx = (float)Math::Cos(radianes);
                         //Apaño debido a los decimales
                         if((float)Math::Abs(xx) < 0.0000000001)
xx = 0;
                         if((float)Math::Abs(yy) < 0.0000000001)
yy = 0;

                          g->DrawLine(red_p, (float)(-xx * r),
(float)(-yy * r), (float)(xx * r), (float)(yy * r));
                    }
              }
private: System::Void
radioButton1_CheckedChanged(System::Object^  sender,
System::EventArgs^  e) {
                    tbOrigen->Enabled = rIntervalo->Checked;
                    tbDestino->Enabled = rIntervalo->Checked;
                    if(rIntervalo->Checked){
                          tbDireccion->Text = "";
                    }

              }
private: System::Void rDireccion_CheckedChanged(System::Object^
sender, System::EventArgs^  e) {
                    tbDireccion->Enabled = rDireccion->Checked;
                    btQuitar->Enabled = rDireccion->Checked;
                    if(rDireccion->Checked){
                          tbOrigen->Text = "";
                          tbDestino->Text = "";
                    }
              }
private: System::Void btAdd_Click(System::Object^  sender,
System::EventArgs^  e) {
                    if(rDireccion->Checked){
                          if(tbDireccion->Text->Equals("")){
```

```
                                    MessageBox::Show("Debe introducir
un ángulo", "Error", MessageBoxButtons::OK,
                                          MessageBoxIcon::Error);
                              }else{
                                    int ang;
                                    try{
                                          //Se obtiene el ángulo
indicado del texto
                                          ang =
int::Parse(tbDireccion->Text);
                                          if(ang < 0 || ang > 180){
                                                MessageBox::Show("El
ángulo introducido debe estar entre 0 y 180 grados", "Error",
MessageBoxButtons::OK,
                                          MessageBoxIcon::Error);
                                                tbDireccion->Focus();
                                          tbDireccion->SelectAll();
                                                return;
                                          }
                                          //Se adapta al angulo
múltiplo de 5 más cercano
                                          int bandera = ang % 5;
                                          int nuevo_origen;
                                          if(bandera > 2) nuevo_origen
= ang + (5-bandera);
                                          else nuevo_origen = ang -
bandera;
                                          if(!lista_angulos-
>Contains(nuevo_origen))
                                                lista_angulos-
>Add(nuevo_origen);
                                          tbDireccion->Text = "";
                                    }catch(Exception ^ex){
                                          tbDireccion->Focus();
                                          tbDireccion->SelectAll();
                                         MessageBox::Show("El texto
introducido no es numérico", "Error", MessageBoxButtons::OK,
                                          MessageBoxIcon::Error);

                                    }
                              }
                        }else if(rIntervalo->Checked){
                              if(tbOrigen->Text->Equals("") &&
tbDestino->Text->Equals("")){
                                    MessageBox::Show("Debe introducir
un ángulo", "Error", MessageBoxButtons::OK,
                                          MessageBoxIcon::Error);
                              }else{
                                    int angO;
                                    int angD;
                                    try{
                                          //Se obtienen los ángulos
extremos de las cajas de texto
                                          angO = int::Parse(tbOrigen-
>Text);
                                          angD = int::Parse(tbDestino-
>Text);
                                          if(angO < 0 || angO > 180){
```

```
                                        MessageBox::Show("El
ángulo de origen introducido debe estar entre 0 y 180 grados",
"Error", MessageBoxButtons::OK,

MessageBoxIcon::Error);
                                        tbOrigen->Focus();
                                        tbOrigen->SelectAll();
                                    }
                                    if(angD < 0 || angD > 180){
                                        MessageBox::Show("El
ángulo de destino introducido debe estar entre 0 y 180 grados",
"Error", MessageBoxButtons::OK,

MessageBoxIcon::Error);
                                        tbDestino->Focus();
                                        tbDestino-
>SelectAll();
                                    }
                                    if(angO > angD){
                                        MessageBox::Show("El
ángulo de origen debe ser menor que el de destino", "Error",
MessageBoxButtons::OK,
                                        MessageBoxIcon::Error);
                                        tbOrigen->Focus();
                                        tbOrigen->SelectAll();
                                        tbDestino->Focus();
                                        tbDestino->SelectAll();
                                        return;
                                    }
                                    //Como los ángulos se
tomarán de 5 en 5 se ajustan los valores introducidos a esa
medida
                                    int bandera = angO % 5;
                                    int nuevo_origen;
                                    if(bandera > 2) nuevo_origen
= angO + (5-bandera);
                                    else nuevo_origen = angO -
bandera;
                                    bandera = angD % 5;
                                    int nuevo_destino;
                                    if(bandera > 2)
nuevo_destino = angD + (5-bandera);
                                    else nuevo_destino = angD -
bandera;
                                    //Añadimos el rango de
ángulos
                                    for(int i = nuevo_origen; i
< nuevo_destino; i+=5){
                                        if(!lista_angulos-
>Contains(i))
                                            lista_angulos-
>Add(i);
                                    }
                                    tbOrigen->Text = "";
                                    tbDestino->Text = "";
                                }catch(Exception ^ex){
```

```
                                    MessageBox::Show("El texto
introducido en alguno de las dos cajas de texto no es numérico",
"Error", MessageBoxButtons::OK,
                                     MessageBoxIcon::Error);
                                    tbOrigen->SelectAll();
                                         tbDestino->SelectAll();
                              }
                        }
                  }
                  panel1->Invalidate();
            }
private: System::Void button1_Click(System::Object^  sender,
System::EventArgs^  e) {
                  lista_angulos->Clear();
                  panel1->Invalidate();
            }
private: System::Void panel1_MouseClick(System::Object^  sender,
System::Windows::Forms::MouseEventArgs^  e) {
                  //Obtenemos el ángulo que se ha pulsado
                  if(e->Y > panel1->Height/2) return;
                  double rad = Math::Atan2(panel1->Width/2 - e-
>X, e->Y - panel1->Height/2);
                  double grados = (rad * 180.0) / Math::PI;
                  if(grados < 0)
                        grados = 360 + grados;
                  grados -= 270;

                  if(grados < 0)
                        grados = -grados;

                  int deg = (int)grados % 360;
                  //Aproximamos el ángulo al múltiplo de 5 más
cercano
                  int bandera = deg % 5;
                  int nuevo_ang;
                  if(bandera > 2) nuevo_ang = deg + (5-bandera);
                  else nuevo_ang = deg - bandera;

                  rDireccion->Checked = true;
                  String ^s = "" + nuevo_ang;
                  tbDireccion->Text = s;
            }
private: System::Void btQuitar_Click(System::Object^  sender,
System::EventArgs^  e) {
                  if(rDireccion->Checked){
                        if(tbDireccion->Text->Equals("")){
                              MessageBox::Show("Debe introducir
un ángulo", "Error", MessageBoxButtons::OK,
                                    MessageBoxIcon::Error);
                        }else{
                              int ang;
                              try{
                                    //Se obtiene el ángulo
indicado del texto
                                    ang =
int::Parse(tbDireccion->Text);
                                    if(ang < 0 || ang > 180){
```

```
                                                MessageBox::Show("El
ángulo introducido debe estar entre 0 y 180 grados", "Error",
MessageBoxButtons::OK,
                                          MessageBoxIcon::Error);
                                                tbDireccion->Focus();
                                          tbDireccion->SelectAll();
                                                return;
                                          }
                                          //Se adapta al angulo
múltiplo de 5 más cercano
                                          int bandera = ang % 5;
                                          int nuevo_origen;
                                          if(bandera > 2) nuevo_origen
= ang + (5-bandera);
                                          else nuevo_origen = ang -
bandera;
                                          if(lista_angulos-
>Contains(nuevo_origen))
                                                lista_angulos-
>Remove(nuevo_origen);
                                          tbDireccion->Text = "";
                                          panel1->Invalidate();
                                    }catch(Exception ^ex){
                                          tbDireccion->Focus();
                                          tbDireccion->SelectAll();
                                         MessageBox::Show("El texto
introducido no es numérico", "Error", MessageBoxButtons::OK,
                                          MessageBoxIcon::Error);

                                    }
                              }
                        }
                  }
};
}
```

VentanaPrincipal.h

```
#pragma once
#include "stdafx.h"
#include "ListOfPoints.h"
#include "ListOfDirections.h"
#include "GeneradorPuntos.h"
#include "FormularioInflado.h"

namespace TFCPrototipo {

      using namespace System;
      using namespace System::ComponentModel;
      using namespace System::Collections;
      using namespace System::Windows::Forms;
      using namespace System::Data;
      using namespace System::Drawing;
      using namespace System::Collections::Generic;
      using namespace System::IO;


      /**
```

```
	* Resumen de VentanaPrincipal (VentanaPrincipal).
	* Esta clase es la clase de la ventana principal del
algoritmo de las escaleras.
	*/
	public ref class VentanaPrincipal : public
System::Windows::Forms::Form
	{
	public:
		/**
		* Constructor de VentanaPrincipal (VentanaPrincipal).
		* En el se lleva a cabo la inicialización de todas
las variables necesarias.
		*/
		VentanaPrincipal(void)
		{
			InitializeComponent();
			numErroresTriang = 0;
			flag=false;
			ds = true;
			paso_a_paso = false;
			guardar_captura = false;
			modo_triangulacion = false;
			//Se inicializan las variables de salvaguarda
de contexto
			contextoCH = new bool[3];
			contextoCH[0] = false;
			contextoCH[1] = false;
			contextoCH[2] = false;
			contextoT = new bool[3];
			contextoT[0] = false;
			contextoT[1] = false;
			contextoT[2] = false;
			//Se inicializan las variables de resultado
			resultado_ortogonal = new ListOfPoints();
			resultado_seleccion = new ListOfPoints();
			resultado_todas = new ListOfPoints();
			resultado_t_ortogonal = new
std::list<ListOfPoints>();
			resultado_t_seleccion = new
std::list<ListOfPoints>();
			resultado_t_todas = new
std::list<ListOfPoints>();
			resultado_todas->setRepetidos(true);
			resultado_seleccion->setRepetidos(true);
			resultado_ortogonal->setRepetidos(true);
			//Se inicializan las variables de interfaz
			color_ortogonal = Color::Green;
			color_seleccion = Color::Blue;
			color_todas = Color::Red;
			//Se inicializan las variables que almacenan
las variables de entrada
			triangulos = new std::list<ListOfPoints*>();
			lop = new ListOfPoints();
			lop_triangulacion = new ListOfPoints();
			//Por defecto se añadirán los puntos que
			//usamos el tutor y yo para hacer ejemplos.
			lop->Add(100, 180, 3);
			lop->Add(110, 150, 3);
```

```
                    lop->Add(130, 220, 3);
                    lop->Add(140, 240, 3);
                    lop->Add(160, 210, 3);
                    lop->Add(170, 130, 3);
                    lop->Add(180, 100, 3);
                    lop->Add(190, 200, 3);
                    lop->Add(210, 140, 3);
                    lop->Add(250, 190, 3);
                    lop->Add(260, 170, 3);
                    lod = new ListOfDirections();
                    /*myPoint punto, punto2;
                    punto.p = Fp2dPoint(1,0);
                    int x = punto.p.x();
                    int y = punto.p.y();
                    punto2 = AuxiliarFunctions::RotateZ(-90,
punto);
                    int xx = punto2.p.x();
                    int yy = punto2.p.y();*/
              }

        public:


        protected:
              /**Lista de triángulos que conforman una
trianglación*/
              std::list<ListOfPoints*> *triangulos;
              /**Resultado del cierre convexo con direcciones
ortogonales*/
              mySegment *resultado_ortogonal;
              /**Resultado del cierre convexo con direcciones
seleccionadas por el usuario*/
              mySegment *resultado_seleccion;
              /**Resultado del cierre convexo con TODAS las
direcciones*/
              mySegment *resultado_todas;


              /**Lista de resultados para las triangulaciones con
direcciones ortogonales*/
              std::list<mySegment> *resultado_t_ortogonal;
              /**Lista de resultados para las triangulaciones con
direcciones seleccionadas por el usuario*/
              std::list<mySegment> *resultado_t_seleccion;
              /**Lista de resultados para las triangulaciones con
TODAS las direcciones*/
              std::list<mySegment> *resultado_t_todas;

              //Variable de prueba
              mySegment *auxiliar;
              /**Colores de la interfaz*/
              Color color_ortogonal;
              /**Colores de la interfaz*/
              Color color_seleccion;
              /**Colores de la interfaz*/
              Color color_todas;
              /**Flag que indica si hay qie borrar la pantalla*/
              bool flag;
```

```
            /**Indica si se está usando el modo paso a paso o
no*/
            bool paso_a_paso;
            /**Indica si hay que guardar una captura de la
pantalla*/
            bool guardar_captura;
            /**Indica si estamos en modo cierre convexo o en modo
triangulación*/
            bool modo_triangulacion;
            /**Variable que indica si los puntos han sido
cargados desde fichero o se han introducido a mano
(triangulaciones)*/
            bool modo_delaunay;
            /**Lista de direcciones*/
            ListOfDirections *lod;
            /**Variable que indica cuando hay que empezar a
dibujar resultados o no*/
            bool ds;
            /**Lista de puntos del panel*/
            ListOfPoints *lop;
            /**Lista de puntos del panel para el caso de los
triangulos*/
            ListOfPoints *lop_triangulacion;
            /**Variable que almacena los contextos*/
            bool *contextoCH;
            /**Variable que almacena los contextos*/
            bool *contextoT;

            /**Variable que almacena los errores en los
triángulos*/
            int numErroresTriang;

            /**
            * Destructor de la clase.
            * Limpiar los recursos que se estén utilizando.
            */
            ~VentanaPrincipal()
            {
                  //delete mp;
                  lop->~ListOfPoints();
                  lod->~ListOfDirections();
                  triangulos->~list();
                  if(resultado_ortogonal != NULL)
resultado_ortogonal->~ListOfPoints();
                  if(resultado_seleccion != NULL)
resultado_seleccion->~ListOfPoints();
                  if(resultado_todas != NULL) resultado_todas-
>~ListOfPoints();

                  if (components)
                  {
                        delete components;
                  }
            }

      private:
            /**
            * Método de dibujado del panel de la ventana.
```

```
            * Dibuja lo que corresponde en cada caso. Si se le
pide realiza una captura de pantalla la realiza
            */
            System::Void panelDibujo_Paint(System::Object^
sender, System::Windows::Forms::PaintEventArgs^  e) {
                        System::Drawing::Graphics ^g = e-
>Graphics;
                        System::Drawing::Drawing2D::Matrix
^matriz = gcnew System::Drawing::Drawing2D::Matrix(1,0,0,-
1,0,panelDibujo->Height);
                        g->Transform = matriz;
                        //if(jare == NULL)lop->DrawPoints(g);
                        //Bandera que indica si hay resultado
que dibujar
                        //if(ds){
                              //Bandera de borrado de la
pantalla
                              if(flag){
                                    SolidBrush ^sd = gcnew
SolidBrush(panelDibujo->BackColor);
                                    g-
>FillRectangle(sd,0,0,panelDibujo->Width, panelDibujo->Height);
                                    flag = false;
                              }
                              SolidBrush ^sd = gcnew
SolidBrush(Color::Black);
                              if(!modo_triangulacion){
                                    if(!ds)return;
                              if(!paso_a_paso){
                                    /*if(resultado_ortogonalV2
!= NULL &&
                                          resultado_ortogonalV2-
>size() > 0){

      std::list<mySegment*>::iterator it;
                                          for(it =
resultado_ortogonalV2->begin(); it != resultado_ortogonalV2-
>end(); it++){
                                                (*it)-
>DrawSegment(g, color_ortogonal);
                                          }
                                    }*/
                                    if(resultado_ortogonal !=
NULL && resultado_ortogonal->Count() > 0)resultado_ortogonal-
>DrawSegment(g, color_ortogonal);
                                    if(resultado_ortogonal !=
NULL)lop->DrawPoints(g);
                                    if(resultado_seleccion !=
NULL && resultado_seleccion->Count() > 0)resultado_seleccion-
>DrawSegment(g, color_seleccion);
                                    if(resultado_seleccion !=
NULL)lop->DrawPoints(g);
                                    if(resultado_todas != NULL
&& resultado_todas->Count() > 0)resultado_todas->DrawSegment(g,
color_todas);
                                    if(resultado_todas !=
NULL)lop->DrawPoints(g);
                              }else
```

```
                                {
                                        if(resultado_ortogonal !=
NULL&& resultado_ortogonal->Count() > 0)resultado_ortogonal-
>DrawStep(g, color_ortogonal);
                                        if(resultado_ortogonal !=
NULL)lop->DrawPoints(g);
                                        if(resultado_seleccion !=
NULL && resultado_seleccion->Count() > 0)resultado_seleccion-
>DrawStep(g, color_seleccion);
                                        if(resultado_seleccion !=
NULL)lop->DrawPoints(g);
                                        if(resultado_todas != NULL
&& resultado_todas->Count() > 0)resultado_todas->DrawStep(g,
color_todas);
                                        if(resultado_todas !=
NULL)lop->DrawPoints(g);
                                }
                                }else{
                                        /*Para las triangulaciones
si la hemos leido de fichero dibujamos los puntos del fichero
                                        pero si la estamos
introduciendo a mano se dibujan los puntos inroducidos por el
usuario*/
                                        if(lop_triangulacion-
>Count() > 0 && modo_delaunay) lop_triangulacion->DrawPoints(g);
                                        if(!modo_delaunay){

std::list<ListOfPoints*>::iterator it_triangulos;
                                                for(it_triangulos =
triangulos->begin();
                                                        it_triangulos !=
triangulos->end(); it_triangulos++){

(*it_triangulos)->DrawPoints(g);

                                                }
                                        }
                                        //Si hay solución para la
triangulación en el caso general se dibuja
                                        if(resultado_t_todas != NULL
&& resultado_t_todas->size() > 0){

std::list<ListOfPoints>::iterator it;
                                                for(it =
resultado_t_todas->begin(); it != resultado_t_todas->end();
it++){
                                                        it-
>DrawClosedSegment(g, color_todas);
                                                }
                                        }
                                        int i = -1;

                                        /*if(tbNT->Text != ""){
                                                i = int::Parse(tbNT-
>Text);
                                        }*/
                                        int n_t = 0;
```

```
                                        //Si hay solución para la
triangulación en el caso ortogonal se dibuja
                                        if(resultado_t_ortogonal !=
NULL && resultado_t_ortogonal->size() > 0){

      std::list<ListOfPoints>::iterator it;
                                              for(it =
resultado_t_ortogonal->begin(); it != resultado_t_ortogonal-
>end(); it++){
                                                    if(i != -1 && i
!= n_t){ n_t++; continue; }
                                                    it-
>DrawSegment(g, color_ortogonal);
                                                    n_t++;
                                              }
                                        }
                                        //Si hay solución para la
triangulación en el caso de todas direcciones se dibuja
                                        if(resultado_t_seleccion !=
NULL && resultado_t_seleccion->size() > 0){

      std::list<ListOfPoints>::iterator it;
                                             for(it =
resultado_t_seleccion->begin(); it != resultado_t_seleccion-
>end(); it++){
                                                    it-
>DrawSegment(g, color_seleccion);
                                              }
                                        }

                                  }
                            //}

                            if(guardar_captura)
                            {
                                  guardar_captura = false;
                                  Bitmap ^bmp = gcnew
Bitmap(panelDibujo->Width, panelDibujo->Height);
                                  Graphics ^g2 =
Graphics::FromImage(bmp);
                                  g2->CopyFromScreen(this-
>Location.X + panelDibujo->Location.X + 7, 25 + this->Location.Y
+ panelDibujo->Location.Y, 0, 0,
                                        *(new
System::Drawing::Size(panelDibujo->Width, panelDibujo->Height)),
System::Drawing::CopyPixelOperation::SourceCopy);

                                  SaveFileDialog ^sfd = gcnew
SaveFileDialog();
                                  sfd->Filter = "Archivo de mapa de
bits | *.bmp";
                                  if(sfd->ShowDialog(this) ==
System::Windows::Forms::DialogResult::OK)
                                  {
                                        bmp->Save(sfd->FileName);
                                  }
                            }
                      }
```

```
      private:
            /**
                  * Método que captura el evento de click sobre
el panel de dibujo
                  * cada vez que se hace click se añade un
punto.
                  */
            System::Void panelDibujo_MouseDown(System::Object^
sender, System::Windows::Forms::MouseEventArgs^  e) {
                        //if(modo_triangulacion) return;
                        //Matriz de transformación afín que
sirve para invertir el origen de coordenadas
                        float matrix[3][3] = {
                              {1, 0, 0},
                              {0, -1, 0},
                              {0, panelDibujo->Height, 1}
                        };
                        //Aplicamos la transformación sobre el
punto clickado
                        myPoint transformada =
AuxiliarFunctions::ChangeOrigin(matrix,(double)e->X,(double)e-
>Y);
                        //Añadimos el punto al conjunto y
refrescamos el panel
                        if(modo_triangulacion && modo_delaunay)
lop_triangulacion->Add(transformada.p.x(), transformada.p.y(),
3);
                        else if(!modo_triangulacion) lop-
>Add(transformada.p.x(), transformada.p.y(), 3);
                        //ds = true;
                        panelDibujo->Refresh();
                  }



      private:
            /**
                  * Método que captura el evento de click sobre
el botón de limpiar.
                  */
            System::Void button2_Click(System::Object^  sender,
System::EventArgs^  e) {
                        LimpiarPanel();
                  }
 private:

       /**
       * Método que sirve para limpiar el panel de dibujo
       */
       void LimpiarPanel(){
             if(modo_triangulacion){
                  lop_triangulacion->Clear();
                  triangulos->clear();
                  if(resultado_t_ortogonal != NULL)
      resultado_t_ortogonal->clear();
```

```
                    if(resultado_t_seleccion != NULL)
      resultado_t_seleccion->clear();
                    if(resultado_t_todas != NULL)
      resultado_t_todas->clear();
                }else{
                     lop->Clear();
                    if(resultado_ortogonal != NULL)
resultado_ortogonal->Clear();
                    if(resultado_seleccion != NULL)
resultado_seleccion->Clear();
                    if(resultado_todas != NULL) resultado_todas-
>Clear();
                }
                cbDireccionesControl->Checked = false;
                cbTodas->Checked = false;
                cbOrtogonal->Checked = false;
                flag = true;
                panelDibujo->Refresh();
         }

         /**
         * Método que genera el cierre convexo de los puntos del
panel con las opciones indicadas en la interfaz.
         */
                void GenerarCierre(){
                    //ds = false;
                    paso_a_paso = false;
                    if(lop != NULL && lop->Count() > 0){
                          ListOfDirections lis_d;
                          lis_d.Clear();
                          //lis_d.Add(0);
                          //lis_d.Add(90);
                          //lis_d.Add(30);
                          //lis_d.Add(70);
                          //lis_d.Add(135);
                          resultado_todas->ResetAnimation();
                          resultado_ortogonal->ResetAnimation();
                          resultado_seleccion->ResetAnimation();
                          if(!cbTodas->Checked && !cbOrtogonal-
>Checked && !cbDireccionesControl->Checked){
                                MessageBox::Show("No hay ningún
tipo de envolvente seleccionada", "Información",
MessageBoxButtons::OK, MessageBoxIcon::Asterisk);
                                return;
                          }
                          if(lop->Count() < 3){
                                MessageBox::Show("Debe introducir
al menos 3 puntos", "Información", MessageBoxButtons::OK,
MessageBoxIcon::Asterisk);
                                return;
                          }
                          if(cbTodas->Checked){
                                TCierre modo = TCierre::todas;
                                int estado_salida = 0;
                                auxiliar =
AuxiliarFunctions::CalcularEnvolvente(*lop,lis_d, modo,
&estado_salida);
                                resultado_todas->Clear();
```

```
                                    std::list<myPoint>::iterator
iterador;
                                    for(iterador = auxiliar-
>getIterator(); iterador != auxiliar->getEnd(); iterador++){
                                          resultado_todas-
>Add(iterador->p.x(), iterador->p.y(), iterador->radius);
                                    }

                                    auxiliar->~ListOfPoints();
                                    auxiliar = NULL;
                              }
                              if(cbOrtogonal->Checked){
                                    lis_d.Add(0);
                                    lis_d.Add(90);
                                    TCierre modo;
                                    modo = TCierre::ortogonal;

//AuxiliarFunctions::CalcularEnvolventeV2(*lop,lis_d, modo,
resultado_ortogonalV2);
                                    int estado_salida = 0;
                                    auxiliar =
AuxiliarFunctions::CalcularEnvolvente(*lop,lis_d, modo,
&estado_salida);
                                    resultado_ortogonal->Clear();
                                    if(estado_salida ==
AuxiliarFunctions::error)
                                    {
                                          MessageBox::Show("Error al
realizar la envolvente", "Error", MessageBoxButtons::OK,
MessageBoxIcon::Asterisk);
                                          return;
                                    }

                                    std::list<myPoint>::iterator
iterador;
                                    for(iterador = auxiliar-
>getIterator(); iterador != auxiliar->getEnd(); iterador++){
                                          resultado_ortogonal-
>Add(iterador->p.x(), iterador->p.y(), iterador->radius);
                                    }
                                    resultado_ortogonal->es_solucion =
auxiliar->es_solucion;
                                    if(!resultado_ortogonal-
>es_solucion){
                                          MessageBox::Show("Para este
conjunto no ha sido posible realizar la intersección",
"Información", MessageBoxButtons::OK, MessageBoxIcon::Asterisk);

                                    }
                                    auxiliar->~ListOfPoints();
                                    auxiliar = NULL;
                                    lis_d.Clear();
                                    //ds = true;
                              }
                              if(cbDireccionesControl->Checked){
                                    bool hay_direcciones = true;
```

```
                                if(directionsControlControl1-
>lista_angulos->Count <= 0){
                                    MessageBox::Show("No hay
ninguna dirección seleccionada", "Información",
MessageBoxButtons::OK, MessageBoxIcon::Asterisk);
                                    hay_direcciones = false;
                                }else
if(directionsControlControl1->lista_angulos->Count == 1){
                                    MessageBox::Show("Para una
única dirección la solución es el mismo conjunto de puntos con
direcciones ortogonales", "Información", MessageBoxButtons::OK,
MessageBoxIcon::Asterisk);
                                    hay_direcciones = false;
                                }
                                //lis_d.Add(30);
                                //lis_d.Add(70);
                                //lis_d.Add(135);
                                if(hay_direcciones){
                                     for(int i = 0; i <
directionsControlControl1->lista_angulos->Count; i ++){

lis_d.Add(directionsControlControl1->lista_angulos[i]);
                                     }
                                    TCierre modo;
                                    modo = medio;
                                    int estado_salida = 0;
                                    auxiliar =
AuxiliarFunctions::CalcularEnvolvente(*lop,lis_d, modo,
&estado_salida);
                                    resultado_seleccion->Clear();
                                    if(estado_salida ==
AuxiliarFunctions::error)
                                     {
                                          MessageBox::Show("Error
al realizar la envolvente", "Error", MessageBoxButtons::OK,
MessageBoxIcon::Asterisk);
                                          return;
                                     }

                                     std::list<myPoint>::iterator
iterador;
                                     for(iterador = auxiliar-
>getIterator(); iterador != auxiliar->getEnd(); iterador++){
                                           resultado_seleccion-
>Add(iterador->p.x(), iterador->p.y(), iterador->radius);
                                     }
                                     resultado_seleccion-
>es_solucion = auxiliar->es_solucion;
                                     if(!resultado_seleccion-
>es_solucion){
                                           MessageBox::Show("Para
este conjunto no ha sido posible realizar la intersección con
direcciones seleccionadas", "Información",
MessageBoxButtons::OK, MessageBoxIcon::Asterisk);

                                     }
                                     auxiliar->~ListOfPoints();
                                     auxiliar = NULL;
```

```
                              }
                        }
                        ds = true;
                        if(resultado_ortogonal->Count() == 0
                              && resultado_seleccion->Count() ==
0 && resultado_todas->Count() == 0) return;
                        if(MessageBox::Show("Pulse Sí para el
modo paso a paso", "Modo", MessageBoxButtons::YesNo,
MessageBoxIcon::Question) ==
System::Windows::Forms::DialogResult::Yes)
                        {
                              paso_a_paso = true;
                        }

                        panelDibujo->Refresh();
                  }else{
                        MessageBox::Show("No hay ningún punto en
el panel", "Información", MessageBoxButtons::OK,
MessageBoxIcon::Asterisk);
                  }
}



      /**
      * Método que genera el cierre convexo de un triangulo
cuando la aplicación está en modo triangulación.
      * @param lop_t es un puntero a la lista de puntos que
conforma el triangulo.
      */
            void GenerarCierre(ListOfPoints *lop_t){
                  //ds = false;
                  paso_a_paso = false;
                  if(lop_t != NULL && lop_t->Count() > 0){
                        ListOfDirections lis_d;
                        lis_d.Clear();
                        mySegment *triangulacion_ortogonal = new
mySegment();
                        mySegment *triangulacion_seleccion = new
mySegment();
                        mySegment triangulacion_todas;
                        triangulacion_ortogonal-
>setRepetidos(true);
                        triangulacion_seleccion-
>setRepetidos(true);
                        /*resultado_todas->ResetAnimation();
                        resultado_ortogonal->ResetAnimation();
                        resultado_seleccion->ResetAnimation();*/
                        if(!cbTodas->Checked && !cbOrtogonal-
>Checked && !cbDireccionesControl->Checked){
                              //MessageBox::Show("No hay ningún
tipo de envolvente seleccionada", "Información",
MessageBoxButtons::OK, MessageBoxIcon::Asterisk);
                              return;
                        }
                        if(cbTodas->Checked){
                              triangulacion_todas = *(lop_t);
```

```
                                    resultado_t_todas-
>push_back(triangulacion_todas);
                              }
                              if(cbOrtogonal->Checked){
                                    lis_d.Add(0);
                                    lis_d.Add(90);
                                    TCierre modo;
                                    modo = TCierre::ortogonal;
                                    int estado_salida = 0;
                                    auxiliar =
AuxiliarFunctions::CalcularEnvolvente(*lop_t,lis_d, modo,
&estado_salida);
                                    if(estado_salida ==
AuxiliarFunctions::error)
                                    {
                                          //MessageBox::Show("Error al
realizar la envolvente (Triángulo con puntos alineados)",
"Error", MessageBoxButtons::OK, MessageBoxIcon::Asterisk);
                                          numErroresTriang++;
                                          return;
                                    }

                                    std::list<myPoint>::iterator
iterador;
                                    for(iterador = auxiliar-
>getIterator(); iterador != auxiliar->getEnd(); iterador++){
                                           triangulacion_ortogonal-
>Add(iterador->p.x(), iterador->p.y(), iterador->radius);
                                    }
                                    auxiliar->~ListOfPoints();
                                    auxiliar = NULL;
                                    lis_d.Clear();
                                    resultado_t_ortogonal-
>push_back(*triangulacion_ortogonal);
                              }
                              if(cbDireccionesControl->Checked){
                                    bool hay_direcciones = true;
                                    //Hay que poner un flag para
controlar que esto sólo salga una vez
                                    if(directionsControlControl1-
>lista_angulos->Count <= 0){
                                          //MessageBox::Show("No hay
ninguna dirección seleccionada", "Información",
MessageBoxButtons::OK, MessageBoxIcon::Asterisk);
                                          hay_direcciones = false;
                                    }else
if(directionsControlControl1->lista_angulos->Count == 1){
                                          //MessageBox::Show("Para una
única dirección la solución es el mismo conjunto de puntos",
"Información", MessageBoxButtons::OK, MessageBoxIcon::Asterisk);
                                          hay_direcciones = false;
                                    }
                                    if(hay_direcciones){
                                           for(int i = 0; i <
directionsControlControl1->lista_angulos->Count; i ++){

lis_d.Add(directionsControlControl1->lista_angulos[i]);
                                           }
```

```
                                    TCierre modo;
                                    modo = medio;
                                    int estado_salida = 0;
                                    auxiliar =
AuxiliarFunctions::CalcularEnvolvente(*lop_t,lis_d, modo,
&estado_salida);
                                    if(estado_salida ==
AuxiliarFunctions::error)
                                     {

      //MessageBox::Show("Error al realizar la envolvente
(Triángulo con puntos alineados)", "Error",
MessageBoxButtons::OK, MessageBoxIcon::Asterisk);
                                          numErroresTriang++;
                                          return;
                                    }

                                     std::list<myPoint>::iterator
iterador;
                                     triangulacion_seleccion-
>setRepetidos(true);
                                     for(iterador = auxiliar-
>getIterator(); iterador != auxiliar->getEnd(); iterador++){

triangulacion_seleccion->Add(iterador->p.x(), iterador->p.y(),
iterador->radius);
                                     }
                                     auxiliar->~ListOfPoints();
                                     auxiliar = NULL;
                                     resultado_t_seleccion-
>push_back(*triangulacion_seleccion);
                                }
                           }
                           if(resultado_ortogonal->Count() == 0
                                && resultado_seleccion->Count() ==
0 && resultado_todas->Count() == 0) return;
                           /*if(MessageBox::Show("Pulse Sí para el
modo paso a paso", "Modo", MessageBoxButtons::YesNo,
MessageBoxIcon::Question) ==
System::Windows::Forms::DialogResult::Yes)
                           {
                                paso_a_paso = true;
                           }*/
                           ds = true;
                           panelDibujo->Refresh();
                     }
}




private:
      /**
            * Manejador del evento click del botón d egenerar el
cierre
            */
      System::Void button6_Click(System::Object^  sender,
System::EventArgs^  e) {
```

```
                    if(!modo_triangulacion)
                          //Generación del cierre para el caso de
sólo la envolvente
                         GenerarCierre();
                    else{
                          /*std::list<mySegment>::iterator
it_orto, it_selec;
                          for(it_orto = resultado_t_ortogonal-
>begin(); it_orto != resultado_t_ortogonal->end(); it_orto++){
                                it_orto->~ListOfPoints();
                          }
                          for(it_selec = resultado_t_seleccion-
>begin(); it_selec != resultado_t_seleccion->end(); it_selec++){
                                it_orto->~ListOfPoints();
                          }*/
                          if(!modo_delaunay && (triangulos == NULL
|| triangulos->size() <= 0))
                          {
                                MessageBox::Show("No hay ningún
punto en el panel", "Información", MessageBoxButtons::OK,
MessageBoxIcon::Asterisk);
                                return;
                          }
                          if(modo_delaunay && (lop_triangulacion
== NULL || lop_triangulacion->Count() <= 0))
                          {
                                MessageBox::Show("No hay ningún
punto en el panel", "Información", MessageBoxButtons::OK,
MessageBoxIcon::Asterisk);
                                return;
                          }
                          if(!cbTodas->Checked && !cbOrtogonal-
>Checked && !cbDireccionesControl->Checked){
                                MessageBox::Show("No hay ningún
tipo de envolvente seleccionada", "Información",
MessageBoxButtons::OK, MessageBoxIcon::Asterisk);
                                return;
                          }
                          if(cbDireccionesControl->Checked &&
directionsControlControl1->lista_angulos->Count <= 0){
                                MessageBox::Show("No hay ninguna
dirección seleccionada", "Información", MessageBoxButtons::OK,
MessageBoxIcon::Asterisk);
                                return;
                          }else if(cbDireccionesControl->Checked
&& directionsControlControl1->lista_angulos->Count == 1){
                                MessageBox::Show("Para una única
dirección la solución es el mismo conjunto de puntos",
"Información", MessageBoxButtons::OK, MessageBoxIcon::Asterisk);
                                return;
                          }
                          if(modo_delaunay){
                                TriangulacionDelaunay();
                          }
                          resultado_t_ortogonal->clear();
                          resultado_t_seleccion->clear();
                          resultado_t_todas->clear();
                          numErroresTriang = 0;
```

```
                        //POSIBLEMENTE HAYA QUE LIBERAR RECURSOS
                        std::list<ListOfPoints*>::iterator
it_triangulos;
                        for(it_triangulos = triangulos->begin();
it_triangulos != triangulos->end(); it_triangulos++){
                              GenerarCierre((*it_triangulos));
                        }

                        if(numErroresTriang > 0)
                        {
                              MessageBox::Show("Error al
realizar la envolvente en alguno de los triangulos\n(Existen
triángulos con puntos alineados)", "Error",
MessageBoxButtons::OK, MessageBoxIcon::Asterisk);
                        }

                        panelDibujo->Refresh();
                  }
            }


private:
      /**
            * Manejador del evento de click sobre el botón o el
menú salir.
            */
      System::Void salirToolStripMenuItem_Click(System::Object^
sender, System::EventArgs^  e) {
                  this->Close();
            }


private:
      /**
            * Manejador del evento de click sobre el  menú
generar cierre.
            */
      System::Void
generarCierreToolStripMenuItem_Click(System::Object^  sender,
System::EventArgs^  e) {
                  GenerarCierre();
            }
private: System::Void
generarCierreToolStripMenuItem_MouseDown(System::Object^
sender, System::Windows::Forms::MouseEventArgs^  e) {
                  //MessageBox::Show("");
            }


private:
      /**
            * Manejador del evento de click sobre el menú de
guardar una distribución de puntos.
            */
      System::Void
guardarDistribuciónToolStripMenuItem_Click(System::Object^
sender, System::EventArgs^  e) {
                  SaveFileDialog ^sfd = gcnew SaveFileDialog();
```

```
                    //Los puntos se guardan en ficheros de formato
ch
                    sfd->Filter = "Archivos de cierre convexo
(*.ch)|*.ch";
                    if(lop == NULL || lop->Count() <= 0) return;
                    if(sfd->ShowDialog(this) ==
System::Windows::Forms::DialogResult::OK){
                          FileStream ^fs = gcnew FileStream(sfd-
>FileName, FileMode::Create);
                          StreamWriter ^sw = gcnew
StreamWriter(fs);
                          std::list<myPoint>::iterator it_guardar;
                          for(it_guardar = lop-
>getIterator();it_guardar != lop->getEnd();it_guardar++){
                                String ^s;
                                //Se escriben las coordenadas y el
radio separados por #
                                s = it_guardar->p.x().ToString() +
"#" +  it_guardar->p.y().ToString() + "#" +
                                      it_guardar-
>radius.ToString();
                                sw->WriteLine(s);
                          }
                          sw->Close();
                          fs->Close();
                    }
              }


private:
      /**
              * Manejador del evento de click sobre el menú de
carga de una distribución de puntos.
              */
      System::Void
cargarDistribuciónToolStripMenuItem_Click(System::Object^
sender, System::EventArgs^  e) {
                    OpenFileDialog ^ofd = gcnew OpenFileDialog();
                    ofd->Filter = "Archivos de cierre convexo
(*.ch)|*.ch";
                    //Se abre el fichero especificado y se
empiezan a leer los puntos de uno en uno con cuidado de que no
haya errores
                    //de formato
                    if(ofd->ShowDialog(this) ==
System::Windows::Forms::DialogResult::OK)
                    {
                          LimpiarPanel();
                          FileStream ^fs;
                          StreamReader ^sr;
                          try{
                                fs = gcnew FileStream(ofd-
>FileName, FileMode::Open);
                                sr = gcnew StreamReader(fs);
                          }catch(Exception ^ex){
                                MessageBox::Show("Se produjo un
error en el fichero de triangulación", "Error",
```

```
                                                MessageBoxButtons::OK,
MessageBoxIcon::Error);
                                    return;
                              }
                              lop->Clear();

                              while(!sr->EndOfStream){
                                    String ^linea = sr->ReadLine();
                                    int i = linea->IndexOf('#');
                                    String ^str_x = linea-
>Substring(0, i);
                                    linea = linea->Substring(i+1);
                                    i = linea->IndexOf('#');
                                    String ^str_y = linea-
>Substring(0, i);
                                    linea = linea->Substring(i+1);
                                    float x, y, r;
                                    try
                                    {
                                          x = float::Parse(str_x);
                                          y = float::Parse(str_y);
                                          r = float::Parse(linea);
                                    }catch(Exception ^ex){
                                          MessageBox::Show("Error en
la conversión");
                                    }
                                    lop->Add(x, y, r);

                              }

                              fs->Close();
                              sr->Close();
                              panelDibujo->Refresh();
                        }
                  }


      private:
            /**
                  * Manejador del evento de click sobre el menú de
color del área de dibujo
                  */
            System::Void
colorDelÁreaDeDibujoToolStripMenuItem_Click(System::Object^
sender, System::EventArgs^  e) {
                        ColorDialog ^cd = gcnew ColorDialog();
                        cd->Color = panelDibujo->BackColor;
                        if(cd->ShowDialog(this) ==
System::Windows::Forms::DialogResult::OK)
                        {
                              panelDibujo->BackColor = cd->Color;
                              panelDibujo->Refresh();
                        }
                  }


      private:
            /**
```

```
            * Manejador del evento de click sobre el menú de
color de la envolvente.
            */
      System::Void
colorDeLaEnvolventeConvexaToolStripMenuItem_Click(System::Object
^  sender, System::EventArgs^  e) {
                  ColorDialog ^cd = gcnew ColorDialog();
                   cd->Color = panelDibujo->BackColor;
                   if(cd->ShowDialog(this) ==
System::Windows::Forms::DialogResult::OK)
                   {
                         color_todas = cd->Color;
                         panelDibujo->Refresh();
                   }
            }


private:
      /**
            * Manejador del evento de click sobre el menú de
color de la envolvente ortogonal.
            */
      System::Void
colorDeLaEnvolventeConDireccionesOrtogonalesToolStripMenuItem_Cl
ick(System::Object^  sender, System::EventArgs^  e) {
                   ColorDialog ^cd = gcnew ColorDialog();
                   cd->Color = panelDibujo->BackColor;
                   if(cd->ShowDialog(this) ==
System::Windows::Forms::DialogResult::OK)
                   {
                         color_ortogonal = cd->Color;
                         panelDibujo->Refresh();
                   }
            }


private:
      /**
            * Manejador del evento de click sobre el menú de
color de la envolvente con direcciones personalizadas.
            */
      System::Void
colorDeLaEnvolventeConDireccionesPersonalizadasToolStripMenuItem
_Click(System::Object^  sender, System::EventArgs^  e) {
                   ColorDialog ^cd = gcnew ColorDialog();
                   cd->Color = panelDibujo->BackColor;
                   if(cd->ShowDialog(this) ==
System::Windows::Forms::DialogResult::OK)
                   {
                         color_seleccion = cd->Color;
                         panelDibujo->Refresh();
                   }
            }


private:
       /**
```

```
		* Manejador del evento de cambio en el check de modo
ortogonal.
		*/
	System::Void cbOrtogonal_CheckedChanged(System::Object^
sender, System::EventArgs^  e) {

			if(!cbOrtogonal->Checked){
				if(!modo_triangulacion)
					resultado_ortogonal->Clear();
				panelDibujo->Refresh();
			}
		}


private:
	/**
		* Manejador del evento de cambio en el check de modo
personalizado
		*/
	System::Void
cbDireccionesControl_CheckedChanged(System::Object^  sender,
System::EventArgs^  e) {

			if(!cbDireccionesControl->Checked){
				if(!modo_triangulacion)
					resultado_seleccion->Clear();
				panelDibujo->Refresh();
			}
			//panelDibujo->Refresh();
		}


private:
	/**
		* Manejador del evento de cambio en el check de modo
todas direcciones.
		*/
	System::Void cbTodas_CheckedChanged(System::Object^
sender, System::EventArgs^  e) {

			if(!cbTodas->Checked){
				if(!modo_triangulacion)
					resultado_todas->Clear();
				panelDibujo->Refresh();
			}
			panelDibujo->Refresh();
		}


private:
	/**
		* Método NO USADO.
		*/
	System::Void normalToolStripMenuItem_Click(System::Object^
sender, System::EventArgs^  e) {
			lop->~ListOfPoints();
			lop = GeneradorPuntos::GenerarPuntos(3);
			panelDibujo->Refresh();
```

```
            }


private:
      /**
            * Manejador del evento de click en el botón de
avance de paso
            */
      System::Void btSiguientePaso_Click(System::Object^  sender,
System::EventArgs^  e) {
                  if(resultado_ortogonal != NULL &&
resultado_ortogonal->Count() > 0)
                        resultado_ortogonal->Next();
                  if(resultado_seleccion != NULL &&
resultado_seleccion->Count() > 0)
                        resultado_seleccion->Next();
                  if(resultado_todas != NULL && resultado_todas-
>Count() > 0)
                        resultado_todas->Next();

                  panelDibujo->Refresh();
            }


private:
      /**
            * Manejador del evento de click en el botón de
retroceso de paso
            */
      System::Void btAnteriorPaso_Click(System::Object^  sender,
System::EventArgs^  e) {
                  if(resultado_ortogonal != NULL &&
resultado_ortogonal->Count() > 0)
                        resultado_ortogonal->Previous();
                  if(resultado_seleccion != NULL &&
resultado_seleccion->Count() > 0)
                        resultado_seleccion->Previous();
                  if(resultado_todas != NULL && resultado_todas-
>Count() > 0)
                        resultado_todas->Previous();

                  panelDibujo->Refresh();
            }


private:
      /**
            * Manejador del evento de apertura del menu de
cierre convexo
            */
      System::Void
cierreConvexoToolStripMenuItem_DropDownOpened(System::Object^
sender, System::EventArgs^  e) {
                  modoPasoAPasoToolStripMenuItem->Checked =
paso_a_paso;
                  modoTriangulaciónToolStripMenuItem->Checked =
modo_triangulacion;
            }
```

```
private:
	/**
		* Manejador del evento de click en el menú de paso a
paso
		*/
	System::Void
modoPasoAPasoToolStripMenuItem_Click(System::Object^  sender,
System::EventArgs^  e) {
			paso_a_paso = !paso_a_paso;
			panelDibujo->Refresh();
		}


private:
	/**
		* Manejador del evento de click en el menú que
guarda la pantalla como una imagen
		*/
	System::Void
guardarComoImágenToolStripMenuItem_Click(System::Object^
sender, System::EventArgs^  e) {
			guardar_captura = true;
			panelDibujo->Refresh();
		}


private:
	/**
		* Manejador del evento de click en el menú que lee
una triangulación de fichero
		*/
	System::Void
leerDeFicheroToolStripMenuItem_Click(System::Object^  sender,
System::EventArgs^  e) {
			OpenFileDialog ^ofd = gcnew OpenFileDialog();
			//Los triángulos se guardan en texto plano
			ofd->Filter = "Archivos de texto
(*.txt)|*.txt";
			if(ofd->ShowDialog(this) ==
System::Windows::Forms::DialogResult::OK)
			{
				FileStream ^fs;
				StreamReader ^sr;
				try{
					fs = gcnew FileStream(ofd-
>FileName, FileMode::Open);
					sr = gcnew StreamReader(fs);
				}catch(Exception ^ex){
					MessageBox::Show("Se produjo un
error en el fichero de triangulación", "Error",
							MessageBoxButtons::OK,
MessageBoxIcon::Error);
					return;
				}
				triangulos->clear();
				while(!sr->EndOfStream){
```

```
                                    //Creamos una lista de puntos que
        almacenará los puntos
                                    //de un triángulo.
                                    ListOfPoints *un_triangulo = new
        ListOfPoints();
                                    int x, y;
                                    //Se lee una linea
                                    String ^linea = sr->ReadLine();
                                    int i = linea->IndexOf('#');
                                    //Se lee la x1
                                    String ^str_x = linea-
        >Substring(0, i);
                                    linea = linea->Substring(i+1);
                                    i = linea->IndexOf('#');
                                    //Se lee la y1
                                    String ^str_y = linea-
        >Substring(0, i);
                                    linea = linea->Substring(i+1);
                                    i = linea->IndexOf('#');
                                    try{
                                          x = int::Parse(str_x);
                                          y = int::Parse(str_y);
                                    }catch(Exception ^ex){
                                          MessageBox::Show("Se produjo
        un error en el fichero de triangulación", "Error",
                                                MessageBoxButtons::OK,
        MessageBoxIcon::Error);
                                          return;
                                    }
                                    un_triangulo->Add(x, y, 3);
                                    //Se lee la x2
                                    str_x = linea->Substring(0, i);
                                    linea = linea->Substring(i+1);
                                    i = linea->IndexOf('#');
                                    //Se lee la y2
                                    str_y = linea->Substring(0, i);
                                    linea = linea->Substring(i+1);
                                    i = linea->IndexOf('#');
                                    try{
                                          x = int::Parse(str_x);
                                          y = int::Parse(str_y);
                                    }catch(Exception ^ex){
                                          MessageBox::Show("Se produjo
        un error en el fichero de triangulación", "Error",
                                                MessageBoxButtons::OK,
        MessageBoxIcon::Error);
                                          return;
                                    }
                                    un_triangulo->Add(x, y, 3);
                                    //Se lee la x3
                                    str_x = linea->Substring(0, i);
                                    linea = linea->Substring(i+1);
                                    //Se lee la y3
                                    str_y = linea->Substring(0);
                                    try{
                                          x = int::Parse(str_x);
                                          y = int::Parse(str_y);
                                    }catch(Exception ^ex){
```

```
                                        MessageBox::Show("Se produjo
un error en el fichero de triangulación", "Error",
                                                MessageBoxButtons::OK,
MessageBoxIcon::Error);
                                        return;
                                    }
                                    /*if(un_triangulo->isAligned(x,
y))
                                    {
                                        MessageBox::Show("Se produjo
un error en el fichero de triangulación (No puden haber dos
puntos alineados en un triangulo)", "Error",
                                                MessageBoxButtons::OK,
MessageBoxIcon::Error);
                                        return;
                                    }*/
                                    un_triangulo->Add(x, y, 3);
                                    triangulos-
>push_back(un_triangulo);
                                }
                                fs->Close();
                                sr->Close();
                                modo_delaunay = false;
                                //CambioDeContexto();
                                modo_triangulacion = true;
                                flag = true;
                                //LimpiarPanel();
                                cbModoTriangulacion->Checked = true;
                                panelDibujo->Invalidate();
                            }
                        }


private:
        /**
                * Manejador del evento de cambio en el menú de modo
triangulación.
                */
        System::Void
modoTriangulaciónToolStripMenuItem_Click(System::Object^
sender, System::EventArgs^  e) {

                    modo_triangulacion = !modo_triangulacion;
                    CambioDeContexto();
                    modoTriangulaciónToolStripMenuItem->Checked =
modo_triangulacion;
                    cbModoTriangulacion->Checked =
modo_triangulacion;
                    panelDibujo->Invalidate();
                }


private:
        /**
                * Manejador del evento de cambio en el check de modo
triangulación.
                */
```

```
      System::Void checkBox1_CheckedChanged(System::Object^
sender, System::EventArgs^  e) {
                  modo_triangulacion = cbModoTriangulacion-
>Checked;
                  CambioDeContexto();
                  btAnteriorPaso->Enabled =
!cbModoTriangulacion->Checked;
                  btSiguientePaso->Enabled =
!cbModoTriangulacion->Checked;
                  cargarDistribuciónToolStripMenuItem->Enabled =
!cbModoTriangulacion->Checked;
                  guardarDistribuciónToolStripMenuItem->Enabled
= !cbModoTriangulacion->Checked;
                  modoPasoAPasoToolStripMenuItem->Enabled =
!cbModoTriangulacion->Checked;
                  //if(!modo_triangulacion && lop->Count() > 0)
ds = true;
                  ds = true;
                  panelDibujo->Invalidate();
            }
            /**
            * Función que realiza el cambio de contexto entre el
modo normal y el modo triangulación.
            */
            void CambioDeContexto(){
                  if(!modo_triangulacion){
                        contextoT[0] = cbOrtogonal->Checked;
                        contextoT[1] = cbDireccionesControl-
>Checked;
                        contextoT[2] = cbTodas->Checked;
                        cbOrtogonal->Checked = contextoCH[0];
                        cbDireccionesControl->Checked =
contextoCH[1];
                        cbTodas->Checked = contextoCH[2];
                  }else{
                        contextoCH[0] = cbOrtogonal->Checked;
                        contextoCH[1] = cbDireccionesControl-
>Checked;
                        contextoCH[2] = cbTodas->Checked;
                        cbOrtogonal->Checked = contextoT[0];
                        cbDireccionesControl->Checked =
contextoT[1];
                        cbTodas->Checked = contextoT[2];
                  }
            }


private:
      /**
            * Manejador del evento de cambio de tamaño de la
ventana.
            */
      System::Void panelDibujo_Resize(System::Object^  sender,
System::EventArgs^  e) {
                  flag = true;
                  panelDibujo->Invalidate();
            }
```

```
private:
	/**
		* Manejador del evento de apertura del menú
triangulación.
		*/
	System::Void
triangulaciónToolStripMenuItem_DropDownOpened(System::Object^
sender, System::EventArgs^  e) {

generarTriangulaciónDeDelaunayToolStripMenuItem->Checked =
modo_delaunay;
			guardarDelaunayEnFicheroToolStripMenuItem-
>Enabled = modo_delaunay;
		}


private:
	/**
		* Manejador del evento de click en el menú de
triangulación de Delaunay.
		*/
	System::Void
generarTriangulaciónDeDelaunayToolStripMenuItem_Click(System::Ob
ject^  sender, System::EventArgs^  e) {
			modo_delaunay = !modo_delaunay;
			lop_triangulacion->Clear();
			triangulos->clear();
			resultado_t_ortogonal->clear();
			resultado_t_seleccion->clear();
			resultado_t_todas->clear();
			panelDibujo->Refresh();
		}
private:
	/**
	* Este método calcula la triangulación de Delaunay para un
conjunto de puntos introducidos por el usuario.
	*/
	void TriangulacionDelaunay(){
		if(!modo_delaunay) return;
		//Miramos que al menos haya 3 puntos
		if(lop_triangulacion->Count() < 3){
			MessageBox::Show("Una triangulación debe
constar al menos de 3 puntos", "Información",
				MessageBoxButtons::OK,
MessageBoxIcon::Exclamation);
		}
		triangulos->clear();
		AuxiliarFunctions::Delaunay(*lop_triangulacion,
triangulos);

	}


private:
	/**
		* Manejador del evento de click en el menú de
guardar una triangulación de Delaunay.
```

```
            */
      System::Void
guardarDelaunayEnFicheroToolStripMenuItem_Click(System::Object^
sender, System::EventArgs^  e) {
                  SaveFileDialog ^sfd = gcnew SaveFileDialog();
                  //Las triangulaciones se guardan en texto
plano
                  sfd->Filter = "Archivos de texto para
triangulación (*.txt)|*.txt";
                  if(modo_delaunay){
                        TriangulacionDelaunay();
                  }else return;
                  if(triangulos == NULL || triangulos->size() <=
0) return;
                  if(sfd->ShowDialog(this) ==
System::Windows::Forms::DialogResult::OK){
                        FileStream ^fs = gcnew FileStream(sfd-
>FileName, FileMode::Create);
                        StreamWriter ^sw = gcnew
StreamWriter(fs);
                        std::list<ListOfPoints*>::iterator
it_guardar;
                        for(it_guardar = triangulos-
>begin();it_guardar != triangulos->end();it_guardar++)
                        {
                              std::list<myPoint>::iterator
it_puntos;
                              String ^s = "";
                              int i = 0;
                              for(it_puntos = (*it_guardar)-
>getIterator(); it_puntos != (*it_guardar)->getEnd();
it_puntos++){
                                    s += it_puntos-
>p.x().ToString() + "#" + it_puntos->p.y().ToString();
                                    if(i != 2) s += "#";
                                    i++;
                              }

                              sw->WriteLine(s);
                        }
                        sw->Close();
                        fs->Close();
                  }
            }


#pragma region Atributos del Formulario
      protected:

      private: System::Windows::Forms::Button^  btLimpiar;
      private: DirectionsControl::DirectionsControlControl^
directionsControlControl1;
      private: System::Windows::Forms::Button^  btCierre;
      private: System::Windows::Forms::MenuStrip^  menuStrip1;
      private: System::Windows::Forms::ToolStripMenuItem^
archivoToolStripMenuItem;
      private: System::Windows::Forms::ToolStripMenuItem^
salirToolStripMenuItem;
```

```
        private: System::Windows::Forms::ToolStripMenuItem^
puntosToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
guardarDistribuciónToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
cargarDistribuciónToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
generarToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
ayudaToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
guardarComoImágenToolStripMenuItem;
        private: System::Windows::Forms::ToolStripSeparator^
toolStripSeparator1;
        private: System::Windows::Forms::ToolStripMenuItem^
cierreConvexoToolStripMenuItem;

        private: System::Windows::Forms::ToolStripSeparator^
toolStripSeparator2;
        private: System::Windows::Forms::ToolStripMenuItem^
preferenciasToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
colorDelÁreaDeDibujoToolStripMenuItem;
        private: System::Windows::Forms::GroupBox^  groupBox1;
        private: System::Windows::Forms::CheckBox^  cbTodas;

        private: System::Windows::Forms::CheckBox^
cbDireccionesControl;

        private: System::Windows::Forms::CheckBox^  cbOrtogonal;
        private: System::Windows::Forms::ToolStripMenuItem^
colorDeLaEnvolventeConvexaToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
colorDeLaEnvolventeConDireccionesOrtogonalesToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
colorDeLaEnvolventeConDireccionesPersonalizadasToolStripMenuItem
;
        private: System::Windows::Forms::ToolStripMenuItem^
generarCierreToolStripMenuItem;
        private: System::Windows::Forms::Button^  btSiguientePaso;
        private: System::Windows::Forms::Button^  btAnteriorPaso;
        private: System::Windows::Forms::ToolStripMenuItem^
cosmosToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
uniformeEnCuadradoToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
uniformeEnDiscoToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
normalToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
lineaToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
modoPasoAPasoToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
triangulaciónToolStripMenuItem;
        private: System::Windows::Forms::ToolStripMenuItem^
leerDeFicheroToolStripMenuItem;
```

```
      private: System::Windows::Forms::ToolStripMenuItem^
generarTriangulaciónDeDelaunayToolStripMenuItem;
      private: System::Windows::Forms::ToolStripMenuItem^
modoTriangulaciónToolStripMenuItem;
      private: System::Windows::Forms::CheckBox^
cbModoTriangulacion;

private: System::Windows::Forms::ToolStripMenuItem^
guardarDelaunayEnFicheroToolStripMenuItem;

      private: System::Windows::Forms::ToolStripMenuItem^
limpiarPantallaToolStripMenuItem;
#pragma endregion
      protected:

private: System::Windows::Forms::Panel^  panelDibujo;
protected:

      protected:

      private:
            /**
            * Variable del diseñador requerida.
            */
            System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
            /**
            * Método necesario para admitir el Diseñador.
            * No se puede modificar el contenido del método con
el editor de código.
            */
            void InitializeComponent(void)
            {

      System::ComponentModel::ComponentResourceManager^
resources = (gcnew
System::ComponentModel::ComponentResourceManager(VentanaPrincipa
l::typeid));
                  this->panelDibujo = (gcnew
System::Windows::Forms::Panel());
                  this->btLimpiar = (gcnew
System::Windows::Forms::Button());
                  this->btCierre = (gcnew
System::Windows::Forms::Button());
                  this->menuStrip1 = (gcnew
System::Windows::Forms::MenuStrip());
                  this->archivoToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                  this->guardarComoImágenToolStripMenuItem =
(gcnew System::Windows::Forms::ToolStripMenuItem());
                  this->toolStripSeparator2 = (gcnew
System::Windows::Forms::ToolStripSeparator());
                  this->preferenciasToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                  this->colorDelÁreaDeDibujoToolStripMenuItem =
(gcnew System::Windows::Forms::ToolStripMenuItem());
```

```
                    this-
>colorDeLaEnvolventeConvexaToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this-
>colorDeLaEnvolventeConDireccionesOrtogonalesToolStripMenuItem =
(gcnew System::Windows::Forms::ToolStripMenuItem());
                    this-
>colorDeLaEnvolventeConDireccionesPersonalizadasToolStripMenuIte
m = (gcnew System::Windows::Forms::ToolStripMenuItem());
                    this->toolStripSeparator1 = (gcnew
System::Windows::Forms::ToolStripSeparator());
                    this->salirToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->puntosToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->guardarDistribuciónToolStripMenuItem =
(gcnew System::Windows::Forms::ToolStripMenuItem());
                    this->cargarDistribuciónToolStripMenuItem =
(gcnew System::Windows::Forms::ToolStripMenuItem());
                    this->generarToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->uniformeEnCuadradoToolStripMenuItem =
(gcnew System::Windows::Forms::ToolStripMenuItem());
                    this->uniformeEnDiscoToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->normalToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->cosmosToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->lineaToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->limpiarPantallaToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->cierreConvexoToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->generarCierreToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->modoPasoAPasoToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->modoTriangulaciónToolStripMenuItem =
(gcnew System::Windows::Forms::ToolStripMenuItem());
                    this->triangulaciónToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->leerDeFicheroToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this-
>generarTriangulaciónDeDelaunayToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->guardarDelaunayEnFicheroToolStripMenuItem
= (gcnew System::Windows::Forms::ToolStripMenuItem());
                    this->ayudaToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->groupBox1 = (gcnew
System::Windows::Forms::GroupBox());
                    this->cbTodas = (gcnew
System::Windows::Forms::CheckBox());
                    this->cbDireccionesControl = (gcnew
System::Windows::Forms::CheckBox());
```

```
                    this->cbOrtogonal = (gcnew
System::Windows::Forms::CheckBox());
                    this->btSiguientePaso = (gcnew
System::Windows::Forms::Button());
                    this->btAnteriorPaso = (gcnew
System::Windows::Forms::Button());
                    this->cbModoTriangulacion = (gcnew
System::Windows::Forms::CheckBox());
                    this->directionsControlControl1 = (gcnew
DirectionsControl::DirectionsControlControl());
                    this->menuStrip1->SuspendLayout();
                    this->groupBox1->SuspendLayout();
                    this->SuspendLayout();
                    //
                    // panelDibujo
                    //
                    this->panelDibujo->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((((System::Win
dows::Forms::AnchorStyles::Top |
System::Windows::Forms::AnchorStyles::Bottom)
                            |
System::Windows::Forms::AnchorStyles::Left)
                            |
System::Windows::Forms::AnchorStyles::Right));
                    this->panelDibujo->BackColor =
System::Drawing::Color::FromArgb(static_cast<System::Int32>(stat
ic_cast<System::Byte>(192)),
static_cast<System::Int32>(static_cast<System::Byte>(255)),

          static_cast<System::Int32>(static_cast<System::Byte>(192)))
;
                    this->panelDibujo->Location =
System::Drawing::Point(338, 156);
                    this->panelDibujo->Name = L"panelDibujo";
                    this->panelDibujo->Size =
System::Drawing::Size(443, 318);
                    this->panelDibujo->TabIndex = 0;
                    this->panelDibujo->Paint += gcnew
System::Windows::Forms::PaintEventHandler(this,
&VentanaPrincipal::panelDibujo_Paint);
                    this->panelDibujo->MouseDown += gcnew
System::Windows::Forms::MouseEventHandler(this,
&VentanaPrincipal::panelDibujo_MouseDown);
                    this->panelDibujo->Resize += gcnew
System::EventHandler(this,
&VentanaPrincipal::panelDibujo_Resize);
                    //
                    // btLimpiar
                    //
                    this->btLimpiar->Location =
System::Drawing::Point(587, 55);
                    this->btLimpiar->Name = L"btLimpiar";
                    this->btLimpiar->Size =
System::Drawing::Size(75, 37);
                    this->btLimpiar->TabIndex = 2;
                    this->btLimpiar->Text = L"Limpiar";
                    this->btLimpiar->UseVisualStyleBackColor =
true;
```

```
                    this->btLimpiar->Click += gcnew
System::EventHandler(this, &VentanaPrincipal::button2_Click);
                    //
                    // btCierre
                    //
                    this->btCierre->Location =
System::Drawing::Point(691, 55);
                    this->btCierre->Name = L"btCierre";
                    this->btCierre->Size =
System::Drawing::Size(75, 37);
                    this->btCierre->TabIndex = 8;
                    this->btCierre->Text = L"Calcular cierre";
                    this->btCierre->UseVisualStyleBackColor = true;
                    this->btCierre->Click += gcnew
System::EventHandler(this, &VentanaPrincipal::button6_Click);
                    //
                    // menuStrip1
                    //
                    this->menuStrip1->Items->AddRange(gcnew
cli::array< System::Windows::Forms::ToolStripItem^  >(5) {this-
>archivoToolStripMenuItem,
                        this->puntosToolStripMenuItem, this-
>cierreConvexoToolStripMenuItem, this-
>triangulaciónToolStripMenuItem, this->ayudaToolStripMenuItem});
                    this->menuStrip1->Location =
System::Drawing::Point(0, 0);
                    this->menuStrip1->Name = L"menuStrip1";
                    this->menuStrip1->Size =
System::Drawing::Size(793, 24);
                    this->menuStrip1->TabIndex = 9;
                    this->menuStrip1->Text = L"menuStrip1";
                    //
                    // archivoToolStripMenuItem
                    //
                    this->archivoToolStripMenuItem->DropDownItems-
>AddRange(gcnew cli::array<
System::Windows::Forms::ToolStripItem^  >(5) {this-
>guardarComoImágenToolStripMenuItem,
                        this->toolStripSeparator2, this-
>preferenciasToolStripMenuItem, this->toolStripSeparator1, this-
>salirToolStripMenuItem});
                    this->archivoToolStripMenuItem->Name =
L"archivoToolStripMenuItem";
                    this->archivoToolStripMenuItem->Size =
System::Drawing::Size(60, 20);
                    this->archivoToolStripMenuItem->Text =
L"&Archivo";
                    //
                    // guardarComoImágenToolStripMenuItem
                    //
                    this->guardarComoImágenToolStripMenuItem->Name
= L"guardarComoImágenToolStripMenuItem";
                    this->guardarComoImágenToolStripMenuItem->Size
= System::Drawing::Size(193, 22);
                    this->guardarComoImágenToolStripMenuItem->Text
= L"Guardar como imagen";
```

```
                    this->guardarComoImágenToolStripMenuItem->Click
+= gcnew System::EventHandler(this,
&VentanaPrincipal::guardarComoImágenToolStripMenuItem_Click);
                    //
                    // toolStripSeparator2
                    //
                    this->toolStripSeparator2->Name =
L"toolStripSeparator2";
                    this->toolStripSeparator2->Size =
System::Drawing::Size(190, 6);
                    //
                    // preferenciasToolStripMenuItem
                    //
                    this->preferenciasToolStripMenuItem-
>DropDownItems->AddRange(gcnew cli::array<
System::Windows::Forms::ToolStripItem^  >(4) {this-
>colorDelÁreaDeDibujoToolStripMenuItem,
                        this-
>colorDeLaEnvolventeConvexaToolStripMenuItem, this-
>colorDeLaEnvolventeConDireccionesOrtogonalesToolStripMenuItem,
this-
>colorDeLaEnvolventeConDireccionesPersonalizadasToolStripMenuIte
m});
                    this->preferenciasToolStripMenuItem->Name =
L"preferenciasToolStripMenuItem";
                    this->preferenciasToolStripMenuItem->Size =
System::Drawing::Size(193, 22);
                    this->preferenciasToolStripMenuItem->Text =
L"Preferencias";
                    //
                    // colorDelÁreaDeDibujoToolStripMenuItem
                    //
                    this->colorDelÁreaDeDibujoToolStripMenuItem-
>Name = L"colorDelÁreaDeDibujoToolStripMenuItem";
                    this->colorDelÁreaDeDibujoToolStripMenuItem-
>Size = System::Drawing::Size(358, 22);
                    this->colorDelÁreaDeDibujoToolStripMenuItem-
>Text = L"Color del área de dibujo";
                    this->colorDelÁreaDeDibujoToolStripMenuItem-
>Click += gcnew System::EventHandler(this,
&VentanaPrincipal::colorDelÁreaDeDibujoToolStripMenuItem_Click);
                    //
                    // colorDeLaEnvolventeConvexaToolStripMenuItem
                    //
                    this-
>colorDeLaEnvolventeConvexaToolStripMenuItem->Name =
L"colorDeLaEnvolventeConvexaToolStripMenuItem";
                    this-
>colorDeLaEnvolventeConvexaToolStripMenuItem->Size =
System::Drawing::Size(358, 22);
                    this-
>colorDeLaEnvolventeConvexaToolStripMenuItem->Text = L"Color de
la envolvente convexa";
                    this-
>colorDeLaEnvolventeConvexaToolStripMenuItem->Click += gcnew
System::EventHandler(this,
&VentanaPrincipal::colorDeLaEnvolventeConvexaToolStripMenuItem_C
lick);
```

```
                    //
                    //
colorDeLaEnvolventeConDireccionesOrtogonalesToolStripMenuItem
                    //
                    this-
>colorDeLaEnvolventeConDireccionesOrtogonalesToolStripMenuItem-
>Name =
L"colorDeLaEnvolventeConDireccionesOrtogonalesToolStripMenuItem"
;
                    this-
>colorDeLaEnvolventeConDireccionesOrtogonalesToolStripMenuItem-
>Size = System::Drawing::Size(358, 22);
                    this-
>colorDeLaEnvolventeConDireccionesOrtogonalesToolStripMenuItem-
>Text = L"Color de la envolvente con direcciones ortogonales";
                    this-
>colorDeLaEnvolventeConDireccionesOrtogonalesToolStripMenuItem-
>Click += gcnew System::EventHandler(this,
&VentanaPrincipal::colorDeLaEnvolventeConDireccionesOrtogonalesT
oolStripMenuItem_Click);
                    //
                    //
colorDeLaEnvolventeConDireccionesPersonalizadasToolStripMenuItem
                    //
                    this-
>colorDeLaEnvolventeConDireccionesPersonalizadasToolStripMenuIte
m->Name =
L"colorDeLaEnvolventeConDireccionesPersonalizadasToolStripMenuIt
em";
                    this-
>colorDeLaEnvolventeConDireccionesPersonalizadasToolStripMenuIte
m->Size = System::Drawing::Size(358, 22);
                    this-
>colorDeLaEnvolventeConDireccionesPersonalizadasToolStripMenuIte
m->Text = L"Color de la envolvente con direcciones
personalizadas";
                    this-
>colorDeLaEnvolventeConDireccionesPersonalizadasToolStripMenuIte
m->Click += gcnew System::EventHandler(this,
&VentanaPrincipal::colorDeLaEnvolventeConDireccionesPersonalizad
asToolStripMenuItem_Click);
                    //
                    // toolStripSeparator1
                    //
                    this->toolStripSeparator1->Name =
L"toolStripSeparator1";
                    this->toolStripSeparator1->Size =
System::Drawing::Size(190, 6);
                    //
                    // salirToolStripMenuItem
                    //
                    this->salirToolStripMenuItem->Name =
L"salirToolStripMenuItem";
                    this->salirToolStripMenuItem->Size =
System::Drawing::Size(193, 22);
                    this->salirToolStripMenuItem->Text = L"Salir";
```

```
                    this->salirToolStripMenuItem->Click += gcnew
System::EventHandler(this,
&VentanaPrincipal::salirToolStripMenuItem_Click);
                    //
                    // puntosToolStripMenuItem
                    //
                    this->puntosToolStripMenuItem->DropDownItems-
>AddRange(gcnew cli::array<
System::Windows::Forms::ToolStripItem^  >(4) {this-
>guardarDistribuciónToolStripMenuItem,
                            this-
>cargarDistribuciónToolStripMenuItem, this-
>generarToolStripMenuItem, this-
>limpiarPantallaToolStripMenuItem});
                    this->puntosToolStripMenuItem->Name =
L"puntosToolStripMenuItem";
                    this->puntosToolStripMenuItem->Size =
System::Drawing::Size(56, 20);
                    this->puntosToolStripMenuItem->Text =
L"&Puntos";
                    //
                    // guardarDistribuciónToolStripMenuItem
                    //
                    this->guardarDistribuciónToolStripMenuItem-
>Name = L"guardarDistribuciónToolStripMenuItem";
                    this->guardarDistribuciónToolStripMenuItem-
>Size = System::Drawing::Size(182, 22);
                    this->guardarDistribuciónToolStripMenuItem-
>Text = L"Guardar distribución";
                    this->guardarDistribuciónToolStripMenuItem-
>Click += gcnew System::EventHandler(this,
&VentanaPrincipal::guardarDistribuciónToolStripMenuItem_Click);
                    //
                    // cargarDistribuciónToolStripMenuItem
                    //
                    this->cargarDistribuciónToolStripMenuItem->Name
= L"cargarDistribuciónToolStripMenuItem";
                    this->cargarDistribuciónToolStripMenuItem->Size
= System::Drawing::Size(182, 22);
                    this->cargarDistribuciónToolStripMenuItem->Text
= L"Cargar distribución";
                    this->cargarDistribuciónToolStripMenuItem-
>Click += gcnew System::EventHandler(this,
&VentanaPrincipal::cargarDistribuciónToolStripMenuItem_Click);
                    //
                    // generarToolStripMenuItem
                    //
                    this->generarToolStripMenuItem->DropDownItems-
>AddRange(gcnew cli::array<
System::Windows::Forms::ToolStripItem^  >(5) {this-
>uniformeEnCuadradoToolStripMenuItem,
                            this->uniformeEnDiscoToolStripMenuItem,
this->normalToolStripMenuItem, this->cosmosToolStripMenuItem,
this->lineaToolStripMenuItem});
                    this->generarToolStripMenuItem->Name =
L"generarToolStripMenuItem";
                    this->generarToolStripMenuItem->Size =
System::Drawing::Size(182, 22);
```

```
                    this->generarToolStripMenuItem->Text =
L"Generar...";
                    this->generarToolStripMenuItem->Visible =
false;
                    //
                    // uniformeEnCuadradoToolStripMenuItem
                    //
                    this->uniformeEnCuadradoToolStripMenuItem->Name
= L"uniformeEnCuadradoToolStripMenuItem";
                    this->uniformeEnCuadradoToolStripMenuItem->Size
= System::Drawing::Size(193, 22);
                    this->uniformeEnCuadradoToolStripMenuItem->Text
= L"Uniforme en cuadrado";
                    //
                    // uniformeEnDiscoToolStripMenuItem
                    //
                    this->uniformeEnDiscoToolStripMenuItem->Name =
L"uniformeEnDiscoToolStripMenuItem";
                    this->uniformeEnDiscoToolStripMenuItem->Size =
System::Drawing::Size(193, 22);
                    this->uniformeEnDiscoToolStripMenuItem->Text =
L"Uniforme en disco";
                    //
                    // normalToolStripMenuItem
                    //
                    this->normalToolStripMenuItem->Name =
L"normalToolStripMenuItem";
                    this->normalToolStripMenuItem->Size =
System::Drawing::Size(193, 22);
                    this->normalToolStripMenuItem->Text =
L"Normal";
                    this->normalToolStripMenuItem->Click += gcnew
System::EventHandler(this,
&VentanaPrincipal::normalToolStripMenuItem_Click);
                    //
                    // cosmosToolStripMenuItem
                    //
                    this->cosmosToolStripMenuItem->Name =
L"cosmosToolStripMenuItem";
                    this->cosmosToolStripMenuItem->Size =
System::Drawing::Size(193, 22);
                    this->cosmosToolStripMenuItem->Text =
L"Constelación";
                    //
                    // lineaToolStripMenuItem
                    //
                    this->lineaToolStripMenuItem->Name =
L"lineaToolStripMenuItem";
                    this->lineaToolStripMenuItem->Size =
System::Drawing::Size(193, 22);
                    this->lineaToolStripMenuItem->Text = L"Linea";
                    //
                    // limpiarPantallaToolStripMenuItem
                    //
                    this->limpiarPantallaToolStripMenuItem->Name =
L"limpiarPantallaToolStripMenuItem";
                    this->limpiarPantallaToolStripMenuItem->Size =
System::Drawing::Size(182, 22);
```

```
                    this->limpiarPantallaToolStripMenuItem->Text =
L"Limpiar pantalla";
                    this->limpiarPantallaToolStripMenuItem->Click
+= gcnew System::EventHandler(this,
&VentanaPrincipal::button2_Click);
                    //
                    // cierreConvexoToolStripMenuItem
                    //
                    this->cierreConvexoToolStripMenuItem-
>DropDownItems->AddRange(gcnew cli::array<
System::Windows::Forms::ToolStripItem^  >(3) {this-
>generarCierreToolStripMenuItem,
                        this->modoPasoAPasoToolStripMenuItem,
this->modoTriangulaciónToolStripMenuItem});
                    this->cierreConvexoToolStripMenuItem->Name =
L"cierreConvexoToolStripMenuItem";
                    this->cierreConvexoToolStripMenuItem->Size =
System::Drawing::Size(97, 20);
                    this->cierreConvexoToolStripMenuItem->Text =
L"&Cierre convexo";
                    this->cierreConvexoToolStripMenuItem-
>DropDownOpened += gcnew System::EventHandler(this,
&VentanaPrincipal::cierreConvexoToolStripMenuItem_DropDownOpened
);
                    //
                    // generarCierreToolStripMenuItem
                    //
                    this->generarCierreToolStripMenuItem->Name =
L"generarCierreToolStripMenuItem";
                    this->generarCierreToolStripMenuItem->Size =
System::Drawing::Size(179, 22);
                    this->generarCierreToolStripMenuItem->Text =
L"Generar cierre";
                    this->generarCierreToolStripMenuItem->MouseDown
+= gcnew System::Windows::Forms::MouseEventHandler(this,
&VentanaPrincipal::generarCierreToolStripMenuItem_MouseDown);
                    this->generarCierreToolStripMenuItem->Click +=
gcnew System::EventHandler(this,
&VentanaPrincipal::generarCierreToolStripMenuItem_Click);
                    //
                    // modoPasoAPasoToolStripMenuItem
                    //
                    this->modoPasoAPasoToolStripMenuItem->Name =
L"modoPasoAPasoToolStripMenuItem";
                    this->modoPasoAPasoToolStripMenuItem->Size =
System::Drawing::Size(179, 22);
                    this->modoPasoAPasoToolStripMenuItem->Text =
L"Modo paso a paso";
                    this->modoPasoAPasoToolStripMenuItem->Click +=
gcnew System::EventHandler(this,
&VentanaPrincipal::modoPasoAPasoToolStripMenuItem_Click);
                    //
                    // modoTriangulaciónToolStripMenuItem
                    //
                    this->modoTriangulaciónToolStripMenuItem->Name
= L"modoTriangulaciónToolStripMenuItem";
                    this->modoTriangulaciónToolStripMenuItem->Size
= System::Drawing::Size(179, 22);
```

```
                    this->modoTriangulaciónToolStripMenuItem->Text
= L"Modo triangulación";
                    this->modoTriangulaciónToolStripMenuItem->Click
+= gcnew System::EventHandler(this,
&VentanaPrincipal::modoTriangulaciónToolStripMenuItem_Click);
                    //
                    // triangulaciónToolStripMenuItem
                    //
                    this->triangulaciónToolStripMenuItem-
>DropDownItems->AddRange(gcnew cli::array<
System::Windows::Forms::ToolStripItem^  >(3) {this-
>leerDeFicheroToolStripMenuItem,
                        this-
>generarTriangulaciónDeDelaunayToolStripMenuItem, this-
>guardarDelaunayEnFicheroToolStripMenuItem});
                    this->triangulaciónToolStripMenuItem->Name =
L"triangulaciónToolStripMenuItem";
                    this->triangulaciónToolStripMenuItem->Size =
System::Drawing::Size(92, 20);
                    this->triangulaciónToolStripMenuItem->Text =
L"&Triangulación";
                    this->triangulaciónToolStripMenuItem-
>DropDownOpened += gcnew System::EventHandler(this,
&VentanaPrincipal::triangulaciónToolStripMenuItem_DropDownOpened
);
                    //
                    // leerDeFicheroToolStripMenuItem
                    //
                    this->leerDeFicheroToolStripMenuItem->Name =
L"leerDeFicheroToolStripMenuItem";
                    this->leerDeFicheroToolStripMenuItem->Size =
System::Drawing::Size(256, 22);
                    this->leerDeFicheroToolStripMenuItem->Text =
L"Leer de fichero";
                    this->leerDeFicheroToolStripMenuItem->Click +=
gcnew System::EventHandler(this,
&VentanaPrincipal::leerDeFicheroToolStripMenuItem_Click);
                    //
                    //
generarTriangulaciónDeDelaunayToolStripMenuItem
                    //
                    this-
>generarTriangulaciónDeDelaunayToolStripMenuItem->Name =
L"generarTriangulaciónDeDelaunayToolStripMenuItem";
                    this-
>generarTriangulaciónDeDelaunayToolStripMenuItem->Size =
System::Drawing::Size(256, 22);
                    this-
>generarTriangulaciónDeDelaunayToolStripMenuItem->Text =
L"Generar triangulación de Delaunay";
                    this-
>generarTriangulaciónDeDelaunayToolStripMenuItem->Click += gcnew
System::EventHandler(this,
&VentanaPrincipal::generarTriangulaciónDeDelaunayToolStripMenuIt
em_Click);
                    //
                    // guardarDelaunayEnFicheroToolStripMenuItem
                    //
```

```
                    this-
>guardarDelaunayEnFicheroToolStripMenuItem->Name =
L"guardarDelaunayEnFicheroToolStripMenuItem";
                    this-
>guardarDelaunayEnFicheroToolStripMenuItem->Size =
System::Drawing::Size(256, 22);
                    this-
>guardarDelaunayEnFicheroToolStripMenuItem->Text = L"Guardar
Delaunay en fichero";
                    this-
>guardarDelaunayEnFicheroToolStripMenuItem->Click += gcnew
System::EventHandler(this,
&VentanaPrincipal::guardarDelaunayEnFicheroToolStripMenuItem_Cli
ck);
                    //
                    // ayudaToolStripMenuItem
                    //
                    this->ayudaToolStripMenuItem->Name =
L"ayudaToolStripMenuItem";
                    this->ayudaToolStripMenuItem->Size =
System::Drawing::Size(53, 20);
                    this->ayudaToolStripMenuItem->Text = L"A&yuda";
                    //
                    // groupBox1
                    //
                    this->groupBox1->Controls->Add(this->cbTodas);
                    this->groupBox1->Controls->Add(this-
>cbDireccionesControl);
                    this->groupBox1->Controls->Add(this-
>cbOrtogonal);
                    this->groupBox1->Location =
System::Drawing::Point(338, 55);
                    this->groupBox1->Name = L"groupBox1";
                    this->groupBox1->Size =
System::Drawing::Size(161, 87);
                    this->groupBox1->TabIndex = 10;
                    this->groupBox1->TabStop = false;
                    this->groupBox1->Text = L"Tipo de envolvente";
                    //
                    // cbTodas
                    //
                    this->cbTodas->AutoSize = true;
                    this->cbTodas->Location =
System::Drawing::Point(7, 66);
                    this->cbTodas->Name = L"cbTodas";
                    this->cbTodas->Size =
System::Drawing::Size(129, 17);
                    this->cbTodas->TabIndex = 2;
                    this->cbTodas->Text = L"Todas las direcciones";
                    this->cbTodas->UseVisualStyleBackColor = true;
                    this->cbTodas->CheckedChanged += gcnew
System::EventHandler(this,
&VentanaPrincipal::cbTodas_CheckedChanged);
                    //
                    // cbDireccionesControl
                    //
                    this->cbDireccionesControl->AutoSize = true;
```

```
                    this->cbDireccionesControl->Location =
System::Drawing::Point(7, 43);
                    this->cbDireccionesControl->Name =
L"cbDireccionesControl";
                    this->cbDireccionesControl->Size =
System::Drawing::Size(153, 17);
                    this->cbDireccionesControl->TabIndex = 1;
                    this->cbDireccionesControl->Text =
L"Direcciones seleccionadas";
                    this->cbDireccionesControl-
>UseVisualStyleBackColor = true;
                    this->cbDireccionesControl->CheckedChanged +=
gcnew System::EventHandler(this,
&VentanaPrincipal::cbDireccionesControl_CheckedChanged);
                    //
                    // cbOrtogonal
                    //
                    this->cbOrtogonal->AutoSize = true;
                    this->cbOrtogonal->Location =
System::Drawing::Point(7, 20);
                    this->cbOrtogonal->Name = L"cbOrtogonal";
                    this->cbOrtogonal->Size =
System::Drawing::Size(72, 17);
                    this->cbOrtogonal->TabIndex = 0;
                    this->cbOrtogonal->Text = L"Ortogonal";
                    this->cbOrtogonal->UseVisualStyleBackColor =
true;
                    this->cbOrtogonal->CheckedChanged += gcnew
System::EventHandler(this,
&VentanaPrincipal::cbOrtogonal_CheckedChanged);
                    //
                    // btSiguientePaso
                    //
                    this->btSiguientePaso->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((System::Windo
ws::Forms::AnchorStyles::Bottom |
System::Windows::Forms::AnchorStyles::Right));
                    this->btSiguientePaso->Location =
System::Drawing::Point(706, 495);
                    this->btSiguientePaso->Name =
L"btSiguientePaso";
                    this->btSiguientePaso->Size =
System::Drawing::Size(75, 35);
                    this->btSiguientePaso->TabIndex = 11;
                    this->btSiguientePaso->Text = L"Paso
Siguiente";
                    this->btSiguientePaso->UseVisualStyleBackColor
= true;
                    this->btSiguientePaso->Click += gcnew
System::EventHandler(this,
&VentanaPrincipal::btSiguientePaso_Click);
                    //
                    // btAnteriorPaso
                    //
                    this->btAnteriorPaso->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((System::Windo
ws::Forms::AnchorStyles::Bottom |
System::Windows::Forms::AnchorStyles::Right));
```

```
                    this->btAnteriorPaso->Location =
System::Drawing::Point(338, 495);
                    this->btAnteriorPaso->Name = L"btAnteriorPaso";
                    this->btAnteriorPaso->Size =
System::Drawing::Size(75, 35);
                    this->btAnteriorPaso->TabIndex = 12;
                    this->btAnteriorPaso->Text = L"Paso Anterior";
                    this->btAnteriorPaso->UseVisualStyleBackColor =
true;
                    this->btAnteriorPaso->Click += gcnew
System::EventHandler(this,
&VentanaPrincipal::btAnteriorPaso_Click);
                    //
                    // cbModoTriangulacion
                    //
                    this->cbModoTriangulacion->AutoSize = true;
                    this->cbModoTriangulacion->Location =
System::Drawing::Point(650, 121);
                    this->cbModoTriangulacion->Name =
L"cbModoTriangulacion";
                    this->cbModoTriangulacion->Size =
System::Drawing::Size(116, 17);
                    this->cbModoTriangulacion->TabIndex = 14;
                    this->cbModoTriangulacion->Text = L"Modo
triangulación";
                    this->cbModoTriangulacion-
>UseVisualStyleBackColor = true;
                    this->cbModoTriangulacion->CheckedChanged +=
gcnew System::EventHandler(this,
&VentanaPrincipal::checkBox1_CheckedChanged);
                    //
                    // directionsControlControl1
                    //
                    this->directionsControlControl1->BorderStyle =
System::Windows::Forms::BorderStyle::Fixed3D;
                    this->directionsControlControl1->Location =
System::Drawing::Point(13, 55);
                    this->directionsControlControl1->Name =
L"directionsControlControl1";
                    this->directionsControlControl1->Size =
System::Drawing::Size(306, 419);
                    this->directionsControlControl1->TabIndex = 4;
                    //
                    // VentanaPrincipal
                    //
                    this->AutoScaleDimensions =
System::Drawing::SizeF(6, 13);
                    this->AutoScaleMode =
System::Windows::Forms::AutoScaleMode::Font;
                    this->ClientSize = System::Drawing::Size(793,
542);
                    this->Controls->Add(this->cbModoTriangulacion);
                    this->Controls->Add(this->btAnteriorPaso);
                    this->Controls->Add(this->btSiguientePaso);
                    this->Controls->Add(this->groupBox1);
                    this->Controls->Add(this->btCierre);
                    this->Controls->Add(this-
>directionsControlControl1);
```

```
                    this->Controls->Add(this->btLimpiar);
                    this->Controls->Add(this->panelDibujo);
                    this->Controls->Add(this->menuStrip1);
                    this->Icon =
(cli::safe_cast<System::Drawing::Icon^  >(resources-
>GetObject(L"$this.Icon")));
                    this->MainMenuStrip = this->menuStrip1;
                    this->Name = L"VentanaPrincipal";
                    this->StartPosition =
System::Windows::Forms::FormStartPosition::CenterParent;
                    this->Text = L"Cierre convexo y triangulaciones
con direcciones restringidas - Método de las esc"
                          L"aleras";
                    this->menuStrip1->ResumeLayout(false);
                    this->menuStrip1->PerformLayout();
                    this->groupBox1->ResumeLayout(false);
                    this->groupBox1->PerformLayout();
                    this->ResumeLayout(false);
                    this->PerformLayout();

                }
#pragma endregion


};
}
```

FormularioInflado.h

```
#pragma once
#include "ElementoInflado.h"
#include "ListOfPoints.h"
#include "Inflador.h"
using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
using namespace System::IO;


namespace TFCPrototipo {

	/// <summary>
	/// Resumen de FormularioInflado
	///
	/// Esta clase se corresponde con el formulario en el que
se muestra la envolvente convexa por el método de inflado.
	/// Ofrece menús y botones que permiten calcular la
envolvente, iniciar la animción, cargar y guardar puntos...
	///
	/// </summary>
	public ref class FormularioInflado : public
System::Windows::Forms::Form
	{
	public:
		///<summary>
		/// Constructor de la clase
		///</summary>
		FormularioInflado(void)
		{
			InitializeComponent();
			//Se inicializan las variables que usará la
clase...
			//...la lista de puntos de entrada
			puntos = new ListOfPoints();
			//...los colores de dibujado
			color_CH = Color::Red;
			color_CH_dr = Color::Green;
			//...las listas de triangulos formados, de
infladores y de direcciones de entrada
			triangulos = new std::list<ListOfPoints*>();
			lista_infladores = new std::list<Inflador*>();
			direcciones = new ListOfDirections();
			//... y finalmente las variables relacionadas
con la animación de inflado
			paso_actual = 0;
			paso_maximo = 0;
			tiempo_animacion = 500;
			animacion_acabada = false;
			tmAnimacion->Interval = tiempo_animacion;
		}
```

```
        protected:
            /// <summary>
            /// Destructor de la clase
            /// Limpia los recursos que se estén utilizando.
            /// </summary>
            ~FormularioInflado()
            {
                if (components)
                {
                    delete components;
                }

                puntos->~ListOfPoints();
            }

        protected:
        //Miembros del formulario
        private:
            //Lista de elementos de la clase inflador que se
usará cada vez que se desee calcular un resultado.
            //Hay un inflador por triángulo
            std::list<Inflador*> *lista_infladores;
            //Flag de borrado de la pantalla
            bool flag;
            //Conjunto de direcciones de entrada del problema
            ListOfDirections *direcciones;
            //Conjunto de puntos de entrada del problema
            ListOfPoints *puntos;
            //Color de la envolvente convexa en sentido habitual
            Color color_CH;
            //Color de la envolvente convexa con direcciones
restringidas
            Color color_CH_dr;
            //Miembro que almacenará los triángulos (ya que solo
trabajaremos con triángulos)
            std::list<ListOfPoints*> *triangulos;

            //Miembro que determina el paso actual de la
animación
            int paso_actual;
            //Miembro que determina el paso máximo de la
animación
            int paso_maximo;
            //Miembro que determina el tiempo entre paso y paso
en la animación
            int tiempo_animacion;
            //Miembro que indica el estado de la animación
            bool animacion_acabada;


//Miembros definidos por el diseñador de VS
//Estos miembros representan elementos de la interfaz tales como
menús, botones...

#pragma region Elementos de la interfaz (botones, menús,
paneles...)
```

```
private: DirectionsControl::DirectionsControlControl^
ControlDirecciones;
private: System::Windows::Forms::MenuStrip^  menuStrip1;
private: System::Windows::Forms::ToolStripMenuItem^
archivoToolStripMenuItem;
private: System::Windows::Forms::ToolStripMenuItem^
ayudaToolStripMenuItem;
private: System::Windows::Forms::ToolStripMenuItem^
salirToolStripMenuItem;
private: System::Windows::Forms::Panel^  panelDibujo;
private: System::Windows::Forms::ToolStripMenuItem^
puntosToolStripMenuItem;
private: System::Windows::Forms::ToolStripMenuItem^
guardarDistribuciónToolStripMenuItem;
private: System::Windows::Forms::ToolStripMenuItem^
cargarDistribuciónToolStripMenuItem;
private: System::Windows::Forms::ToolStripMenuItem^
preferenciasToolStripMenuItem;
private: System::Windows::Forms::ToolStripMenuItem^
colorDelÁreaDeDibujoToolStripMenuItem;
private: System::Windows::Forms::Button^  btIniciar;
private: System::Windows::Forms::Button^  btLimpiar;
private: System::Windows::Forms::Button^  btPlay;
private: System::Windows::Forms::Button^  btPause;
private: System::Windows::Forms::Button^  btNext;
private: System::Windows::Forms::Button^  btPrevious;
private: System::Windows::Forms::Timer^  tmAnimacion;
private: System::ComponentModel::IContainer^  components;

protected:
protected:

#pragma endregion

      private:
            /// <summary>
            /// Variable del diseñador requerida.
            /// </summary>


#pragma region Windows Form Designer generated code
            /// <summary>
            /// Método necesario para admitir el Diseñador. No se
puede modificar
            /// el contenido del método con el editor de código.
            /// Da valores iniciales a todos los elementos de la
interfaz colocándolos en su sitio y estableciendo el valor de
sus propiedades.
            /// </summary>
            void InitializeComponent(void)
            {
                  this->components = (gcnew
System::ComponentModel::Container());

      System::ComponentModel::ComponentResourceManager^
resources = (gcnew
System::ComponentModel::ComponentResourceManager(FormularioInfla
do::typeid));
```

```
                    this->ControlDirecciones = (gcnew
DirectionsControl::DirectionsControlControl());
                    this->menuStrip1 = (gcnew
System::Windows::Forms::MenuStrip());
                    this->archivoToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->preferenciasToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->colorDelÁreaDeDibujoToolStripMenuItem =
(gcnew System::Windows::Forms::ToolStripMenuItem());
                    this->salirToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->puntosToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->guardarDistribuciónToolStripMenuItem =
(gcnew System::Windows::Forms::ToolStripMenuItem());
                    this->cargarDistribuciónToolStripMenuItem =
(gcnew System::Windows::Forms::ToolStripMenuItem());
                    this->ayudaToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                    this->panelDibujo = (gcnew
System::Windows::Forms::Panel());
                    this->btIniciar = (gcnew
System::Windows::Forms::Button());
                    this->btLimpiar = (gcnew
System::Windows::Forms::Button());
                    this->btPlay = (gcnew
System::Windows::Forms::Button());
                    this->btPause = (gcnew
System::Windows::Forms::Button());
                    this->btNext = (gcnew
System::Windows::Forms::Button());
                    this->btPrevious = (gcnew
System::Windows::Forms::Button());
                    this->tmAnimacion = (gcnew
System::Windows::Forms::Timer(this->components));
                    this->menuStrip1->SuspendLayout();
                    this->SuspendLayout();
                    //
                    // ControlDirecciones
                    //
                    this->ControlDirecciones->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>(((System::Wind
ows::Forms::AnchorStyles::Top |
System::Windows::Forms::AnchorStyles::Bottom)
                            |
System::Windows::Forms::AnchorStyles::Left));
                    this->ControlDirecciones->BorderStyle =
System::Windows::Forms::BorderStyle::Fixed3D;
                    this->ControlDirecciones->Location =
System::Drawing::Point(12, 94);
                    this->ControlDirecciones->Name =
L"ControlDirecciones";
                    this->ControlDirecciones->Size =
System::Drawing::Size(306, 419);
                    this->ControlDirecciones->TabIndex = 0;
                    //
                    // menuStrip1
```

```
                    //
                    this->menuStrip1->Items->AddRange(gcnew
cli::array< System::Windows::Forms::ToolStripItem^  >(3) {this-
>archivoToolStripMenuItem,
                        this->puntosToolStripMenuItem, this-
>ayudaToolStripMenuItem});
                    this->menuStrip1->Location =
System::Drawing::Point(0, 0);
                    this->menuStrip1->Name = L"menuStrip1";
                    this->menuStrip1->Size =
System::Drawing::Size(941, 24);
                    this->menuStrip1->TabIndex = 1;
                    this->menuStrip1->Text = L"menuStrip1";
                    //
                    // archivoToolStripMenuItem
                    //
                    this->archivoToolStripMenuItem->DropDownItems-
>AddRange(gcnew cli::array<
System::Windows::Forms::ToolStripItem^  >(2) {this-
>preferenciasToolStripMenuItem,
                        this->salirToolStripMenuItem});
                    this->archivoToolStripMenuItem->Name =
L"archivoToolStripMenuItem";
                    this->archivoToolStripMenuItem->Size =
System::Drawing::Size(60, 20);
                    this->archivoToolStripMenuItem->Text =
L"Archivo";
                    //
                    // preferenciasToolStripMenuItem
                    //
                    this->preferenciasToolStripMenuItem-
>DropDownItems->AddRange(gcnew cli::array<
System::Windows::Forms::ToolStripItem^  >(1) {this-
>colorDelÁreaDeDibujoToolStripMenuItem});
                    this->preferenciasToolStripMenuItem->Name =
L"preferenciasToolStripMenuItem";
                    this->preferenciasToolStripMenuItem->Size =
System::Drawing::Size(138, 22);
                    this->preferenciasToolStripMenuItem->Text =
L"Preferencias";
                    //
                    // colorDelÁreaDeDibujoToolStripMenuItem
                    //
                    this->colorDelÁreaDeDibujoToolStripMenuItem-
>Name = L"colorDelÁreaDeDibujoToolStripMenuItem";
                    this->colorDelÁreaDeDibujoToolStripMenuItem-
>Size = System::Drawing::Size(200, 22);
                    this->colorDelÁreaDeDibujoToolStripMenuItem-
>Text = L"Color del área de dibujo";
                    this->colorDelÁreaDeDibujoToolStripMenuItem-
>Click += gcnew System::EventHandler(this,
&FormularioInflado::colorDelÁreaDeDibujoToolStripMenuItem_Click)
;
                    //
                    // salirToolStripMenuItem
                    //
                    this->salirToolStripMenuItem->Name =
L"salirToolStripMenuItem";
```

```
                    this->salirToolStripMenuItem->Size =
System::Drawing::Size(138, 22);
                    this->salirToolStripMenuItem->Text = L"Salir";
                    this->salirToolStripMenuItem->Click += gcnew
System::EventHandler(this,
&FormularioInflado::salirToolStripMenuItem_Click);
                    //
                    // puntosToolStripMenuItem
                    //
                    this->puntosToolStripMenuItem->DropDownItems-
>AddRange(gcnew cli::array<
System::Windows::Forms::ToolStripItem^  >(2) {this-
>guardarDistribuciónToolStripMenuItem,
                            this-
>cargarDistribuciónToolStripMenuItem});
                    this->puntosToolStripMenuItem->Name =
L"puntosToolStripMenuItem";
                    this->puntosToolStripMenuItem->Size =
System::Drawing::Size(56, 20);
                    this->puntosToolStripMenuItem->Text =
L"Puntos";
                    //
                    // guardarDistribuciónToolStripMenuItem
                    //
                    this->guardarDistribuciónToolStripMenuItem-
>Name = L"guardarDistribuciónToolStripMenuItem";
                    this->guardarDistribuciónToolStripMenuItem-
>Size = System::Drawing::Size(182, 22);
                    this->guardarDistribuciónToolStripMenuItem-
>Text = L"Guardar distribución";
                    this->guardarDistribuciónToolStripMenuItem-
>Click += gcnew System::EventHandler(this,
&FormularioInflado::guardarDistribuciónToolStripMenuItem_Click);
                    //
                    // cargarDistribuciónToolStripMenuItem
                    //
                    this->cargarDistribuciónToolStripMenuItem->Name
= L"cargarDistribuciónToolStripMenuItem";
                    this->cargarDistribuciónToolStripMenuItem->Size
= System::Drawing::Size(182, 22);
                    this->cargarDistribuciónToolStripMenuItem->Text
= L"Cargar distribución";
                    this->cargarDistribuciónToolStripMenuItem-
>Click += gcnew System::EventHandler(this,
&FormularioInflado::cargarDistribuciónToolStripMenuItem_Click);
                    //
                    // ayudaToolStripMenuItem
                    //
                    this->ayudaToolStripMenuItem->Name =
L"ayudaToolStripMenuItem";
                    this->ayudaToolStripMenuItem->Size =
System::Drawing::Size(53, 20);
                    this->ayudaToolStripMenuItem->Text = L"Ayuda";
                    //
                    // panelDibujo
                    //
                    this->panelDibujo->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((((System::Win
```

```
dows::Forms::AnchorStyles::Top |
System::Windows::Forms::AnchorStyles::Bottom)
                        |
System::Windows::Forms::AnchorStyles::Left)
                        |
System::Windows::Forms::AnchorStyles::Right));
                    this->panelDibujo->BackColor =
System::Drawing::Color::LemonChiffon;
                    this->panelDibujo->Location =
System::Drawing::Point(333, 94);
                    this->panelDibujo->Name = L"panelDibujo";
                    this->panelDibujo->Size =
System::Drawing::Size(596, 419);
                    this->panelDibujo->TabIndex = 2;
                    this->panelDibujo->Paint += gcnew
System::Windows::Forms::PaintEventHandler(this,
&FormularioInflado::panelDibujo_Paint);
                    this->panelDibujo->MouseDown += gcnew
System::Windows::Forms::MouseEventHandler(this,
&FormularioInflado::panelDibujo_MouseDown);
                    //
                    // btIniciar
                    //
                    this->btIniciar->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((System::Windo
ws::Forms::AnchorStyles::Top |
System::Windows::Forms::AnchorStyles::Right));
                    this->btIniciar->Location =
System::Drawing::Point(827, 44);
                    this->btIniciar->Name = L"btIniciar";
                    this->btIniciar->Size =
System::Drawing::Size(101, 44);
                    this->btIniciar->TabIndex = 3;
                    this->btIniciar->Text = L"Calcular";
                    this->btIniciar->UseVisualStyleBackColor =
true;
                    this->btIniciar->Click += gcnew
System::EventHandler(this, &FormularioInflado::btIniciar_Click);
                    //
                    // btLimpiar
                    //
                    this->btLimpiar->Location =
System::Drawing::Point(333, 65);
                    this->btLimpiar->Name = L"btLimpiar";
                    this->btLimpiar->Size =
System::Drawing::Size(150, 23);
                    this->btLimpiar->TabIndex = 4;
                    this->btLimpiar->Text = L"Limpiar Pantalla";
                    this->btLimpiar->UseVisualStyleBackColor =
true;
                    this->btLimpiar->Click += gcnew
System::EventHandler(this, &FormularioInflado::btLimpiar_Click);
                    //
                    // btPlay
                    //
                    this->btPlay->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((System::Windo
```

```
ws::Forms::AnchorStyles::Top |
System::Windows::Forms::AnchorStyles::Right));
                    this->btPlay->Enabled = false;
                    this->btPlay->Image =
(cli::safe_cast<System::Drawing::Image^  >(resources-
>GetObject(L"btPlay.Image")));
                    this->btPlay->Location =
System::Drawing::Point(644, 44);
                    this->btPlay->Name = L"btPlay";
                    this->btPlay->Size = System::Drawing::Size(53,
44);
                    this->btPlay->TabIndex = 5;
                    this->btPlay->UseVisualStyleBackColor = true;
                    this->btPlay->Click += gcnew
System::EventHandler(this, &FormularioInflado::btPlay_Click);
                    //
                    // btPause
                    //
                    this->btPause->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((System::Windo
ws::Forms::AnchorStyles::Top |
System::Windows::Forms::AnchorStyles::Right));
                    this->btPause->Enabled = false;
                    this->btPause->Image =
(cli::safe_cast<System::Drawing::Image^  >(resources-
>GetObject(L"btPause.Image")));
                    this->btPause->Location =
System::Drawing::Point(592, 44);
                    this->btPause->Name = L"btPause";
                    this->btPause->Size = System::Drawing::Size(53,
44);
                    this->btPause->TabIndex = 6;
                    this->btPause->UseVisualStyleBackColor = true;
                    this->btPause->Click += gcnew
System::EventHandler(this, &FormularioInflado::btPause_Click);
                    //
                    // btNext
                    //
                    this->btNext->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((System::Windo
ws::Forms::AnchorStyles::Top |
System::Windows::Forms::AnchorStyles::Right));
                    this->btNext->Enabled = false;
                    this->btNext->Image =
(cli::safe_cast<System::Drawing::Image^  >(resources-
>GetObject(L"btNext.Image")));
                    this->btNext->Location =
System::Drawing::Point(696, 44);
                    this->btNext->Name = L"btNext";
                    this->btNext->Size = System::Drawing::Size(53,
44);
                    this->btNext->TabIndex = 7;
                    this->btNext->UseVisualStyleBackColor = true;
                    this->btNext->Click += gcnew
System::EventHandler(this, &FormularioInflado::btNext_Click);
                    //
                    // btPrevious
                    //
```

```
                    this->btPrevious->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((System::Windo
ws::Forms::AnchorStyles::Top |
System::Windows::Forms::AnchorStyles::Right));
                    this->btPrevious->Enabled = false;
                    this->btPrevious->Image =
(cli::safe_cast<System::Drawing::Image^  >(resources-
>GetObject(L"btPrevious.Image")));
                    this->btPrevious->Location =
System::Drawing::Point(540, 44);
                    this->btPrevious->Name = L"btPrevious";
                    this->btPrevious->Size =
System::Drawing::Size(53, 44);
                    this->btPrevious->TabIndex = 8;
                    this->btPrevious->UseVisualStyleBackColor =
true;
                    this->btPrevious->Click += gcnew
System::EventHandler(this,
&FormularioInflado::btPrevious_Click);
                    //
                    // tmAnimacion
                    //
                    this->tmAnimacion->Tick += gcnew
System::EventHandler(this,
&FormularioInflado::tmAnimacion_Tick);
                    //
                    // FormularioInflado
                    //
                    this->AutoScaleDimensions =
System::Drawing::SizeF(6, 13);
                    this->AutoScaleMode =
System::Windows::Forms::AutoScaleMode::Font;
                    this->ClientSize = System::Drawing::Size(941,
525);
                    this->Controls->Add(this->btPrevious);
                    this->Controls->Add(this->btNext);
                    this->Controls->Add(this->btPause);
                    this->Controls->Add(this->btPlay);
                    this->Controls->Add(this->btLimpiar);
                    this->Controls->Add(this->btIniciar);
                    this->Controls->Add(this->panelDibujo);
                    this->Controls->Add(this->ControlDirecciones);
                    this->Controls->Add(this->menuStrip1);
                    this->Icon =
(cli::safe_cast<System::Drawing::Icon^  >(resources-
>GetObject(L"$this.Icon")));
                    this->MainMenuStrip = this->menuStrip1;
                    this->Name = L"FormularioInflado";
                    this->StartPosition =
System::Windows::Forms::FormStartPosition::CenterParent;
                    this->Text = L"Cierre convexo y triangulaciones
con direcciones restringidas - Método de las esc"
                          L"aleras";
                    this->menuStrip1->ResumeLayout(false);
                    this->menuStrip1->PerformLayout();
                    this->ResumeLayout(false);
                    this->PerformLayout();
```

```
            }
#pragma endregion
      private:
            System::Void btPrueba_Click(System::Object^  sender,
System::EventArgs^  e) {
                  }
      private:
            ///<summary>
            /// Manejador del evento de click del botón salir.
            /// Provoca la salida del formulario
            ///</summary>
            System::Void
salirToolStripMenuItem_Click(System::Object^  sender,
System::EventArgs^  e) {
                        this->Close();
                  }
private:
      ///<summary>
      /// Maneja el evento de dibujado del panel
      /// Lo que hace es dibujar los puntos que haya, los
resultados...
      /// También lleva a cabo el proceso de limpiar la pantalla
y de hacer capturas.
      ///</summary>
      System::Void panelDibujo_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {
                  System::Drawing::Graphics ^g = e->Graphics;
                  System::Drawing::Drawing2D::Matrix ^matriz =
gcnew System::Drawing::Drawing2D::Matrix(1,0,0,-1,0,panelDibujo-
>Height);
                  g->Transform = matriz;
                        //Bandera de borrado de la pantalla
                  if(flag){
                        SolidBrush ^sd = gcnew
SolidBrush(panelDibujo->BackColor);
                        g->FillRectangle(sd,0,0,panelDibujo-
>Width, panelDibujo->Height);
                        flag = false;
                  }

                  if(puntos->Count() > 0)
                  {
                        puntos->DrawPoints(g);
                        if(lista_infladores != NULL &&
lista_infladores->size() > 0)
                        {
                              std::list<Inflador*>::iterator
it_inf;
                              for(it_inf = lista_infladores-
>begin(); it_inf != lista_infladores->end(); it_inf++)
                              {
                                    //Tomamos el mínimo entre el
paso que establece la animación y la cantidad de pasos de un
inflador en particular
                                    //con ello se permite que se
inflen los polígonos más grandes
                                    int paso_dibujar =
Math::Min(paso_actual, (*it_inf)->paso_actual-1);
```

```
                                        (*it_inf)-
>PintaPaso(paso_dibujar, g, color_CH_dr, color_CH);
                                    }
                               }
                          }
                     }
private:
      ///<summary>
      /// Manejador del evento de click sobre el panel.
      /// Toma los puntos y los guarda en el programa, para ello
tiene que hacer un cambio de coordenadas.
      ///</summary>
      System::Void panelDibujo_MouseDown(System::Object^  sender,
System::Windows::Forms::MouseEventArgs^  e) {
                    //Matriz de transformación afín que sirve para
invertir el origen de coordenadas
                    float matrix[3][3] = {
                          {1, 0, 0},
                          {0, -1, 0},
                          {0, panelDibujo->Height, 1}
                    };
                    //Aplicamos la transformación sobre el punto
clickado
                    myPoint transformada =
AuxiliarFunctions::ChangeOrigin(matrix,(double)e->X,(double)e-
>Y);
                    //Añadimos el punto al conjunto y refrescamos
el panel
                    puntos->Add(transformada.p.x(),
transformada.p.y(), 3);

                    tmAnimacion->Enabled = false;
                    btNext->Enabled = false;
                    btPrevious->Enabled = false;
                    btPlay->Enabled = false;
                    btPause->Enabled = false;

                    panelDibujo->Refresh();
               }

private:
      ///<summary>
      /// Función que lleva a cabo la limpieza de la pantalla
      ///</summary>
      void LimpiarPanel()
      {
            puntos->Clear();
            triangulos->clear();
            direcciones->Clear();
            lista_infladores->clear();
            flag = true;
            panelDibujo->Refresh();
      }

private:
      ///<summary>
```

```
      /// Manejador del evento de click sobre el menú de cargar
distribución
      /// Provoca la carga de un conjunto de puntos.
      ///</summary>
      System::Void
cargarDistribuciónToolStripMenuItem_Click(System::Object^
sender, System::EventArgs^  e) {
                  OpenFileDialog ^ofd = gcnew OpenFileDialog();
                  //Carga ficheros de extension .ch
                  //Controlando errores de formato interno del
archivo
                  ofd->Filter = "Archivos de cierre convexo
(*.ch)|*.ch";
                  if(ofd->ShowDialog(this) ==
System::Windows::Forms::DialogResult::OK)
                  {
                        LimpiarPanel();
                        FileStream ^fs;
                        StreamReader ^sr;
                        try{
                              fs = gcnew FileStream(ofd-
>FileName, FileMode::Open);
                              sr = gcnew StreamReader(fs);
                        }catch(Exception ^ex){
                              MessageBox::Show("Se produjo un
error en el fichero de triangulación", "Error",
                                          MessageBoxButtons::OK,
MessageBoxIcon::Error);
                              return;
                        }
                        puntos->Clear();

                        while(!sr->EndOfStream){
                              //Mientras queden lineas en el
archivo se van leyendo las coordenadas y el radio separados por
#
                              String ^linea = sr->ReadLine();
                              int i = linea->IndexOf('#');
                              String ^str_x = linea-
>Substring(0, i);
                              linea = linea->Substring(i+1);
                              i = linea->IndexOf('#');
                              String ^str_y = linea-
>Substring(0, i);
                              linea = linea->Substring(i+1);
                              float x, y, r;
                              try
                              {
                                    x = float::Parse(str_x);
                                    y = float::Parse(str_y);
                                    r = float::Parse(linea);
                              }catch(Exception ^ex){
                                    MessageBox::Show("Error en
la conversión");
                              }
                              puntos->Add(x, y, r);

                        }
```

```
                        fs->Close();
                        sr->Close();
                        panelDibujo->Refresh();
                    }
                }

private:
      ///<summary>
      /// Menajador del evento de click sobre el menú de guardar
una distribución
      /// Provoca el guardado de la misma con el formato usado
por toda la aplicación.
      ///</summary>
      System::Void
guardarDistribuciónToolStripMenuItem_Click(System::Object^
sender, System::EventArgs^  e) {
                    SaveFileDialog ^sfd = gcnew SaveFileDialog();
                    sfd->Filter = "Archivos de cierre convexo
(*.ch)|*.ch";
                    if(puntos == NULL || puntos->Count() <= 0)
return;
                    if(sfd->ShowDialog(this) ==
System::Windows::Forms::DialogResult::OK){
                        FileStream ^fs = gcnew FileStream(sfd-
>FileName, FileMode::Create);
                        StreamWriter ^sw = gcnew
StreamWriter(fs);
                        std::list<myPoint>::iterator it_guardar;
                        for(it_guardar = puntos-
>getIterator();it_guardar != puntos->getEnd();it_guardar++){
                            String ^s;
                            s = it_guardar->p.x().ToString() +
"#" +  it_guardar->p.y().ToString() + "#" +
                                it_guardar-
>radius.ToString();
                            sw->WriteLine(s);
                        }
                        sw->Close();
                        fs->Close();
                    }
                }
private:
      ///<summary>
      /// Manejador del evento click en el menú de cambiar el
color de área de dibujo
      /// Abre un diálogo y lleva a cabo el cambio
      ///</summary>
      System::Void
colorDelÁreaDeDibujoToolStripMenuItem_Click(System::Object^
sender, System::EventArgs^  e) {
                    ColorDialog ^cd = gcnew ColorDialog();
                    cd->Color = panelDibujo->BackColor;
                    if(cd->ShowDialog(this) ==
System::Windows::Forms::DialogResult::OK)
                    {
                        panelDibujo->BackColor = cd->Color;
                        panelDibujo->Refresh();
```

```
                        }
                }
private:
      ///<summary>
      /// Manejador del evento click sobre el botón que inicia el
cálculo de la CH
      ///</summary>
      System::Void btIniciar_Click(System::Object^  sender,
System::EventArgs^  e) {
            //Se controla que los datos de entrada sean correctos
                  if(puntos->Count() < 2)
                  {
                        MessageBox::Show("Debe haber al menos 3
puntos en el panel", "Error", MessageBoxButtons::OK,
MessageBoxIcon::Error);
                        return;
                  }
                  if(ControlDirecciones->lista_angulos->Count <=
0){
                        MessageBox::Show("No hay ninguna
dirección seleccionada", "Información", MessageBoxButtons::OK,
MessageBoxIcon::Asterisk);
                        return;
                        //hay_direcciones = false;
                  }else if(ControlDirecciones->lista_angulos-
>Count == 1){
                        MessageBox::Show("Para una única
dirección la solución es el mismo conjunto de puntos con
direcciones ortogonales", "Información", MessageBoxButtons::OK,
MessageBoxIcon::Asterisk);
                        return;
                        //hay_direcciones = false;
                  }

                  direcciones->Clear();

                  for(int i = 0; i < ControlDirecciones-
>lista_angulos->Count; i ++)
                  {
                  direcciones->Add(ControlDirecciones-
>lista_angulos[i]);
                  }

                  direcciones->Sort();

                  //Una vez comprobada la validez de la entrada se
triangula el conjunto de puntos dado
                  triangulos->clear();
                  lista_infladores->clear();
                  AuxiliarFunctions::Delaunay(*puntos, triangulos);
                  //Para cada triángulo se aplica el inflado
                  std::list<ListOfPoints*>::iterator it_triangulos;
                  paso_maximo = int::MinValue;
                  for(it_triangulos = triangulos->begin();
it_triangulos != triangulos->end(); it_triangulos++)
                  {
                        Inflador *nuevo_inflador = new
Inflador(*direcciones, (*(*it_triangulos)));
```

```
                    while(!nuevo_inflador->inflado_completo)
                    {
                          nuevo_inflador->Inflar();
                    }
                    paso_maximo = Math::Max(paso_maximo,
nuevo_inflador->paso_actual - 1);
                    lista_infladores->push_back(nuevo_inflador);
              }

              paso_actual = -1;

              btPlay->Enabled = true;
              btPause->Enabled = true;
              btPrevious->Enabled = true;
              btNext->Enabled = true;
              panelDibujo->Refresh();
        }

private:
        ///<summary>
        /// Manejador del evento de click sobre el botón limpiar
pantalla
        ///</summary>
        System::Void btLimpiar_Click(System::Object^  sender,
System::EventArgs^  e) {
                    LimpiarPanel();
              }

private:
        ///<summary>
        /// Manejador del evento de click sobre el botón de
retroceso de la animación
        ///</summary>
        System::Void btPrevious_Click(System::Object^  sender,
System::EventArgs^  e) {
                   paso_actual--;
                   if(paso_actual < -1) paso_actual = -1;
                   panelDibujo->Refresh();
              }
private:
        ///<summary>
        /// Manejador del evento de click sobre el botón de pausa
de la animación
        ///</summary>
        System::Void btPause_Click(System::Object^  sender,
System::EventArgs^  e) {
                    tmAnimacion->Enabled = false;
                    btNext->Enabled = true;
                    btPrevious->Enabled = true;
                    btPlay->Enabled = true;
                    btPause->Enabled = false;
              }

private:
        ///<summary>
        /// Manejador del evento de click sobre el botón de inicio
de la animación
        ///</summary>
```

```
	System::Void btPlay_Click(System::Object^  sender,
System::EventArgs^  e) {
			if(animacion_acabada)
			{
				paso_actual = -1;
				animacion_acabada = false;
			}
			tmAnimacion->Enabled = true;
			btNext->Enabled = false;
			btPrevious->Enabled = false;
			btPlay->Enabled = false;
			btPause->Enabled = true;
		}

private:
	///<summary>
	/// Manejador del evento de click sobre el botón de avance
de la animación
	///</summary>
	System::Void btNext_Click(System::Object^  sender,
System::EventArgs^  e) {
			paso_actual++;
			if(paso_actual > paso_maximo) paso_actual =
paso_maximo;
			panelDibujo->Refresh();
		}

private:
	///<summary>
	/// Manejador del evento de tick del timer de la animación
	///</summary>
	System::Void tmAnimacion_Tick(System::Object^  sender,
System::EventArgs^  e) {
			paso_actual++;
			if(paso_actual > paso_maximo){
				tmAnimacion->Enabled = false;
				animacion_acabada = true;
				paso_actual = paso_maximo;
				btNext->Enabled = true;
				btPrevious->Enabled = true;
				btPlay->Enabled = true;
				btPause->Enabled = false;
			}
			panelDibujo->Refresh();
			tmAnimacion->Interval = tiempo_animacion;

		}
};
}
```

FormularioEntrada.h

```
#pragma once


#include "VentanaPrincipal.h"
#include "FormularioInflado.h"

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;


namespace TFCPrototipo {

	/**
	 * Ventana de entrada.
	 * Esta clase es una interfaz con el usuario que le permite
elegir un modo de operación u otro.
	 *
	 */
	public ref class FormularioEntrada : public
System::Windows::Forms::Form
	{
	public:
		/**
		 * Constructor de la clase.
		 */
		FormularioEntrada(void)
		{
			InitializeComponent();
		}

	protected:
		/**
		 * Destructor de la clase.
		 * Limpiar los recursos que se estén utilizando.
		 */
		~FormularioEntrada()
		{
			if (components)
			{
				delete components;
			}
		}
	/**
	 * En esta sección se presentan los miembros de la clase
que representan los elementos de la interfaz.
	 * Estos miembros son botones y menús.
	 */
	private: System::Windows::Forms::Button^  btInflado;
	protected:
	private: System::Windows::Forms::Button^  btSalir;
	private: System::Windows::Forms::MenuStrip^  menuStrip1;
```

```
      private: System::Windows::Forms::ToolStripMenuItem^
archivoToolStripMenuItem;
      private: System::Windows::Forms::ToolStripMenuItem^
salirToolStripMenuItem;
      private: System::Windows::Forms::ToolStripMenuItem^
ayudaToolStripMenuItem;
      private: System::Windows::Forms::Button^  btEscalera;

      private:
            /**
             * Variable del diseñador requerida.
             */
            System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
            /**
             * Método necesario para admitir el Diseñador.
             * No se puede modificar el contenido del método con
el editor de código.
             */
            void InitializeComponent(void)
            {

      System::ComponentModel::ComponentResourceManager^
resources = (gcnew
System::ComponentModel::ComponentResourceManager(FormularioEntra
da::typeid));
                  this->btInflado = (gcnew
System::Windows::Forms::Button());
                  this->btSalir = (gcnew
System::Windows::Forms::Button());
                  this->menuStrip1 = (gcnew
System::Windows::Forms::MenuStrip());
                  this->archivoToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                  this->salirToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                  this->ayudaToolStripMenuItem = (gcnew
System::Windows::Forms::ToolStripMenuItem());
                  this->btEscalera = (gcnew
System::Windows::Forms::Button());
                  this->menuStrip1->SuspendLayout();
                  this->SuspendLayout();
                  //
                  // btInflado
                  //
                  resources->ApplyResources(this->btInflado,
L"btInflado");
                  this->btInflado->Name = L"btInflado";
                  this->btInflado->UseVisualStyleBackColor =
true;
                  this->btInflado->Click += gcnew
System::EventHandler(this, &FormularioEntrada::btInflado_Click);
                  //
                  // btSalir
                  //
                  resources->ApplyResources(this->btSalir,
L"btSalir");
```

```
                    this->btSalir->Name = L"btSalir";
                    this->btSalir->UseVisualStyleBackColor = true;
                    this->btSalir->Click += gcnew
System::EventHandler(this, &FormularioEntrada::btSalir_Click);
                    //
                    // menuStrip1
                    //
                    this->menuStrip1->Items->AddRange(gcnew
cli::array< System::Windows::Forms::ToolStripItem^  >(2) {this-
>archivoToolStripMenuItem,
                          this->ayudaToolStripMenuItem});
                    resources->ApplyResources(this->menuStrip1,
L"menuStrip1");
                    this->menuStrip1->Name = L"menuStrip1";
                    //
                    // archivoToolStripMenuItem
                    //
                    this->archivoToolStripMenuItem->DropDownItems-
>AddRange(gcnew cli::array<
System::Windows::Forms::ToolStripItem^  >(1) {this-
>salirToolStripMenuItem});
                    this->archivoToolStripMenuItem->Name =
L"archivoToolStripMenuItem";
                    resources->ApplyResources(this-
>archivoToolStripMenuItem, L"archivoToolStripMenuItem");
                    //
                    // salirToolStripMenuItem
                    //
                    this->salirToolStripMenuItem->Name =
L"salirToolStripMenuItem";
                    resources->ApplyResources(this-
>salirToolStripMenuItem, L"salirToolStripMenuItem");
                    this->salirToolStripMenuItem->Click += gcnew
System::EventHandler(this, &FormularioEntrada::btSalir_Click);
                    //
                    // ayudaToolStripMenuItem
                    //
                    this->ayudaToolStripMenuItem->Name =
L"ayudaToolStripMenuItem";
                    resources->ApplyResources(this-
>ayudaToolStripMenuItem, L"ayudaToolStripMenuItem");
                    //
                    // btEscalera
                    //
                    resources->ApplyResources(this->btEscalera,
L"btEscalera");
                    this->btEscalera->Name = L"btEscalera";
                    this->btEscalera->UseVisualStyleBackColor =
true;
                    this->btEscalera->Click += gcnew
System::EventHandler(this,
&FormularioEntrada::btEscalera_Click);
                    //
                    // FormularioEntrada
                    //
                    resources->ApplyResources(this, L"$this");
                    this->AutoScaleMode =
System::Windows::Forms::AutoScaleMode::Font;
```

```
                this->Controls->Add(this->btEscalera);
                this->Controls->Add(this->btSalir);
                this->Controls->Add(this->btInflado);
                this->Controls->Add(this->menuStrip1);
                this->FormBorderStyle =
System::Windows::Forms::FormBorderStyle::FixedSingle;
                this->MainMenuStrip = this->menuStrip1;
                this->MaximizeBox = false;
                this->MinimizeBox = false;
                this->Name = L"FormularioEntrada";
                this->menuStrip1->ResumeLayout(false);
                this->menuStrip1->PerformLayout();
                this->ResumeLayout(false);
                this->PerformLayout();

            }
#pragma endregion


      private:
            /**
             * Este método sirve de manejador del evento de click
sobre el botón salir.
             */
            System::Void btSalir_Click(System::Object^  sender,
System::EventArgs^  e) {
                        this->Close();
                }


private:
      /**
       * Este método sirve de manejador del evento click sobre el
botón del modo escalera.
       */
      System::Void btEscalera_Click(System::Object^  sender,
System::EventArgs^  e) {
                  VentanaPrincipal ^vp = gcnew
VentanaPrincipal();
                  vp->ShowDialog();
            }


private:
      /**
       * Este método sirve de manejador del evento click sobre el
botón del modo inflado.
       */
      System::Void btInflado_Click(System::Object^  sender,
System::EventArgs^  e) {
                  FormularioInflado ^fi = gcnew
FormularioInflado();
                  fi->ShowDialog();
            }
};
}
```

5.7. Manual de usuario

En este manual se presentarán de manera directa y sencilla las indicaciones necesarias para operar el software de la mejor manera posible. Asimismo se indicará el motivo de los distintos mensajes de error que puede mostrar la aplicación.

El propósito de este producto software es la realización de envolventes convexas y triangulaciones con direcciones restringidas mediante dos posibles métodos: el método de la intersección de semiplanos escalera y el método del inflado de paralelepípedos.

5.7.1.Menú de inicio

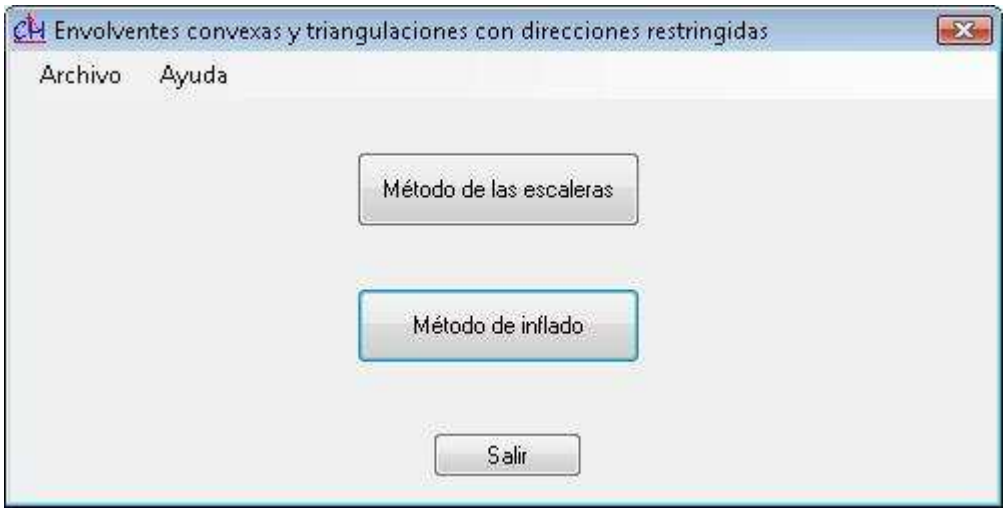


Imagen 5.4. Menú de inicio.

Nada más iniciar la aplicación aparece la pantalla mostrada en la Imagen 5.4. Pulsando sobre el botón salir se cerrará la aplicación. Asimimo, pulsando sobre el menú *Archivo* se desplegará un menú que unicamente contiene la opción *Salir*. Pulsando con el ratón sobre dicha opción también se cierra la aplicación.

Pulsando sobre el menú de ayuda se abrirá este manual.

Pulsando sobre el botón *Método de las escaleras* se abrirá una ventana para llevar a cabo envolventes convexas y triangulaciones con direcciones restringidas mediante el método de la intersección de semiplanos escalera, también llamado método de las escaleras.

Pulsando sobre el botón *Método del inflado* se abrirá una ventana para llevar a cabo triangulaciones con direcciones restringidas mediante el método del inflado de paralelepípedos.

5.7.2.Ventana *Método de las escaleras*

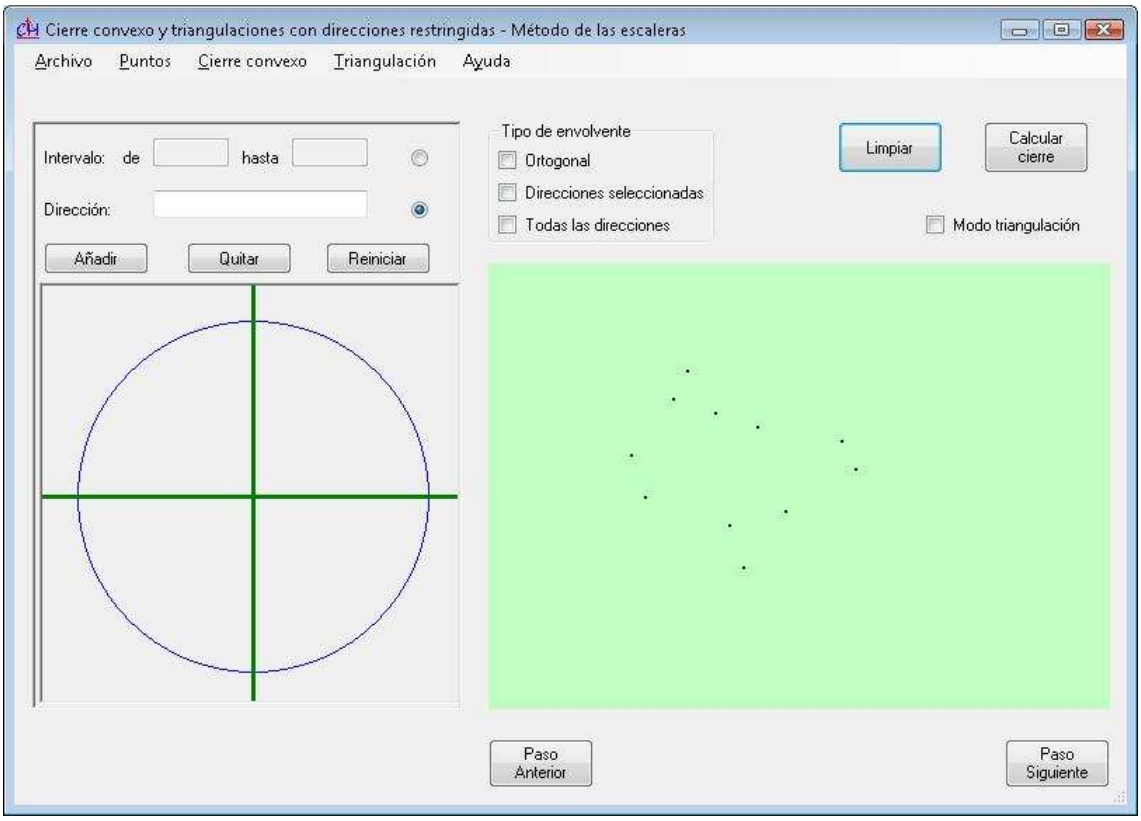


Imagen 5.5. Ventana *Método de las escaleras.*

Esta ventana ofrece la posibilidad de llevar a cabo tanto envolventes convexas como triangulaciones con direcciones restringidas.

Para limpiar el área de dibujo pulse sobre el botón *Limpiar* o acuda a la opción *Limpiar pantalla* del menú *Puntos.*

Mediante la opción *Guardar como imagen* del menú *Archivo* puede guardar el área de dibujo como una imagen para su posterior visualización.

Pulsando sobre el menú de *Ayuda* se abrirá este manual.

Para pesonalizar la interfaz hay que acudir a la opción *Preferencias* del menú *Archivo.* Dicha opción contiene sub-opciones para configurar diferentes elementos de la interfaz. Haciendo click sobre cada uno se podrán establecer los valores que el usuario desee.

Para volver al menú de inicio pulse sobre la “X” recuadrada en rojo de la esquina superior derecha o use la opción *Salir* del menú *Archivo.*

5.7.2.1. Selección de direcciones

A continuación se explicará el sistema de selección de direcciones. El usuario puede elegir entre tres tipos de modos de dirección en el panel *Tipo de envolvente*.

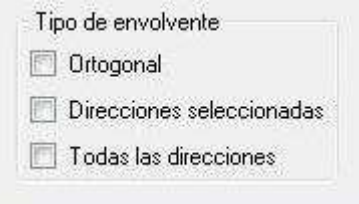


Imagen 5.6. Panel *Tipo de envolvente*.

El modo *Ortogonal* indica que se usarán las direcciones 0º y 90º, el modo *Todas las direcciones* indica que se usarán todas las direcciones, es decir, se hará la envolvente convexa en el caso usual, y *Direcciones seleccionadas* indica que se hará la envolvente convexa con las direcciones que el usuario seleccione mediante el control selector de direcciones.

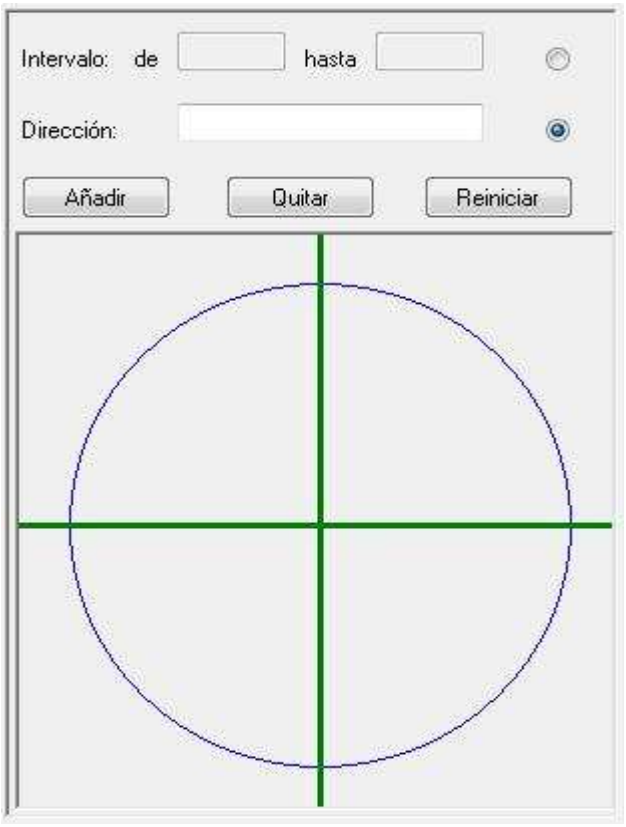


Imagen 5.7. Control selector de direcciones.

El usuario podrá elegir una dirección de dos posibles maneras: introduciendo su valor numérico en grados en el cuadro de texto *Dirección*, para lo cual éste debe estar activado, o bien pulsando sobre la mitad superior del círculo. Al pulsar sobre el círculo aparecerá el valor numérico de la dirección en el cuadro de texto *Dirección*. En cualquiera de los dos casos para añadir la dirección al conjunto habrá que pulsar sobre el botón *Añadir*.

De manera análoga se pueden eliminar direcciones del conjunto pulsando sobre el botón *Quitar*.

Para añadir un rango de direcciones habrá que activar la opción *Intervalo* (usando el botón circular a la derecha de dicha opción). Una vez activada se podrá introducir una dirección de inicio del rango y una de fin, ambas expresadas en grados. La dirección de inicio debe ser mayor que la de fin. Las direcciones introducidas irán en intervalos de cinco en cinco grados.

Pulsando sobre el botón *Reiniciar* se vacía el conjunto de direcciones.

5.7.2.2. Realización de una envolvente convexa

A continuación se explicará como realizar una envolvente convexa paso a paso.

Para llevar a cabo una envolvente lo primero que se debe hacer es **introducir puntos**. Para ello se puede usar el panel de dibujo (en verde en la Imagen 5.5) o se puede usar la opción *Cargar distribución* del menú *Puntos*. Los ficheros de puntos tienen una extensión *ch* y se crean usando la opción *Guardar distribución* de menú *Puntos*, que guarda la distribución que exista en el panel de dibujo en ese momento.

En cualquiera de los dos casos el **número mínimo de puntos es tres**. En caso contrarío se informará al usuario, antes de empezar el proceso, mediante un diálogo informativo.

Después habrá que **seleccionar las direcciones** del mismo modo que se explicó en la sección anterior.

Una vez hecho esto **pulse sobre el botón *Calcular cierre*** y obtendrá el resultado. Una vez obtenido el resultado el programa pregunta si desea visualizar paso a paso el mismo. Si pulsa sobre *Sí* tendrá que ir pulsando sobré el botón *Paso Siguiente* para avanzar o sobre el botón *Paso Anterior* para retroceder. Si pulsa *No* verá el resultado final. En cualquiera de los dos casos se puede activar/desactivar el modo paso a paso desde el menú *Cierre convexo* pulsando sobre la opción *Modo paso a paso.*

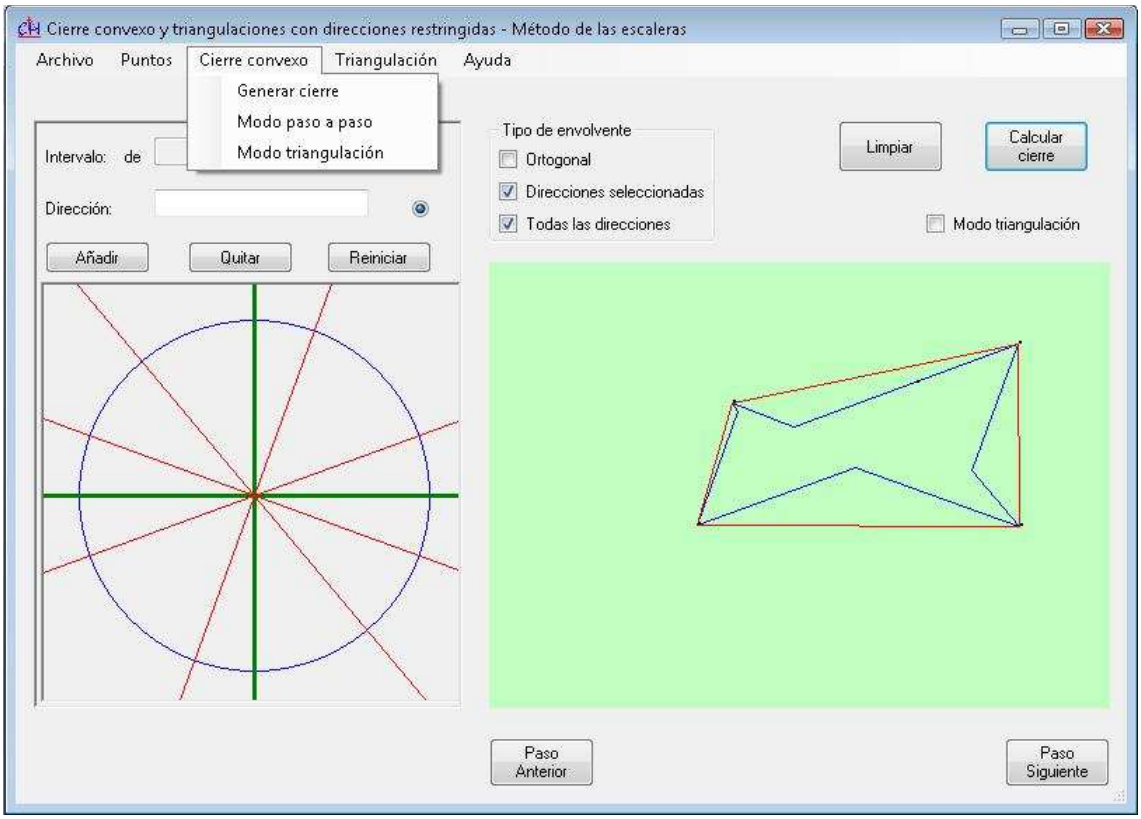


Imagen 5.8. Resultado del cálculo de envolvente convexa con todas las direcciones y con direcciones seleccionadas.

5.7.2.3. Realización de una triangulación

Para operar con triangulaciones debe **poner la aplicación en modo triangulación**. Para ello use la caja de selección etiquetada como *Modo triangulación* o use la opción *Modo triangulación* del menú *Cierre convexo.*

Para llevar a cabo una triangulación con direcciones restringidas hay que **introducir los triángulos** mediante fichero o mediante el panel de dibujo.

Los ficheros de entrada tendrán extensión txt para que puedan ser editados sin ningún problema. Cada línea del fichero se corresponde con un triángulo y tendrá la siguiente estructura: $x_1\#y_1\#x_2\#y_2\#x_3\#y_3$. Donde x_i es la *coordenada x* de cada punto e y_i es la *coordenada y* de cada punto.

Si no se quiere introducir una triangulación mediante fichero habrá que activar la opción *Generar triangulación de Delaunay* del menú *Triangulación.* Con ello se

pueden introducir puntos pinchando con el ratón sobre el panel de dibujo. Posteriormente se llevará a cabo la triangulación de Delaunay.

Tras introducir los puntos de la triangulación mediante cualquiera de los dos procedimientos descritos habrá que **seleccionar las direcciones** como se describe en el apartado 5.7.2.1.

Una vez hecho todo lo anteriormente descrito habrá que **pulsar sobre el botón *Calcular cierre***. Si no se han introducido puntos o no hay los suficientes (es necesario un mínimo de tres) se notificará mediante un diálogo informativo. Si, por el contrario, la entrada es correcta se mostrará el resultado.

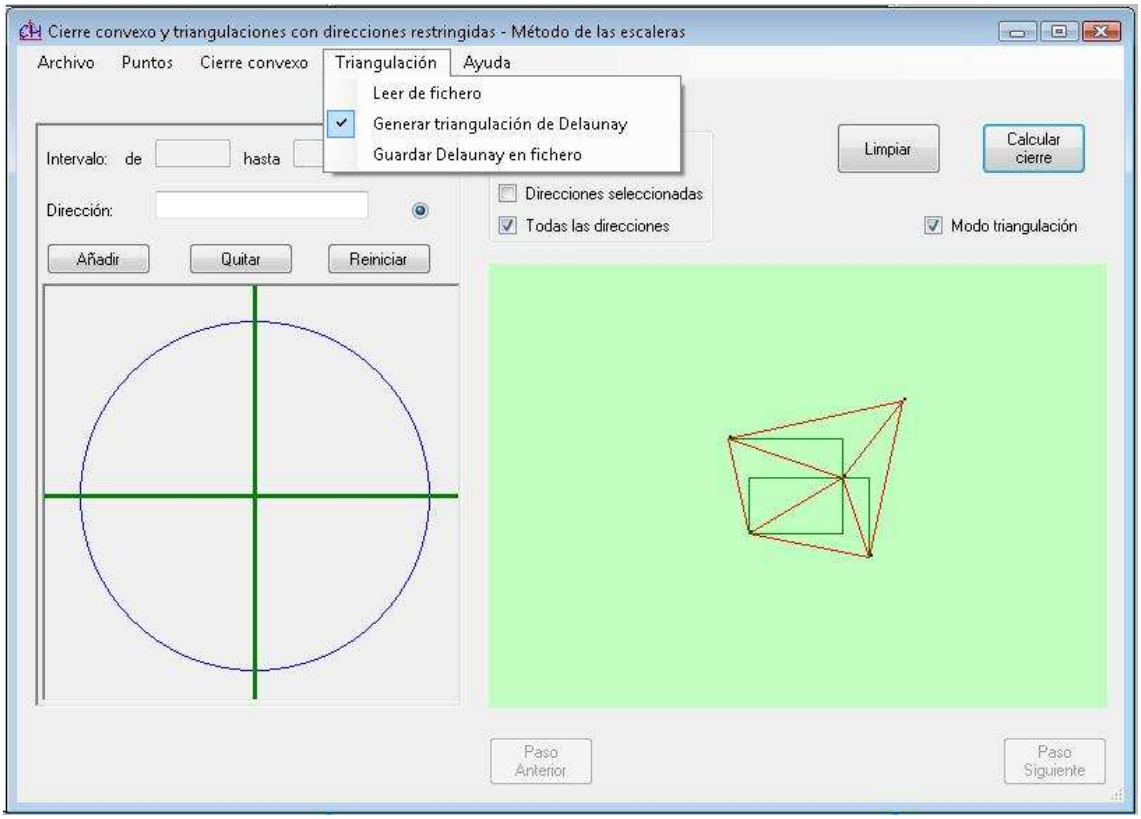


Imagen 5.9. Resultado de calcular una triangulación con todas las direcciones y con direcciones 0º y 90º. En este caso se ha realizado la triangulación de Delaunay de los puntos.

5.7.3.Mensajes de error

A continuación se presenta una lista de los mensajes de error conocidos y la causa de los mismos:

- *Error al realizar la envolvente* o *Error al realizar la envolvente en alguno de los triangulos. (Existen triángulos con puntos alineados)*: este mensaje de error se produce cuando al construir una escalera entre dos puntos maximales, uno de ellos no es único en la dirección correspondiente. También puede ser debido (en muy pocos casos) a algún error de precisión en los cálculos matemáticos usados para la construcción de la escalera. Para resolverlo vuelva a elegir la distribución de puntos.

- *Para este conjunto no ha sido posible realizar la intersección*: este error se produce cuando dos semiplanos escalera se intersecan en más de dos puntos. Si se produce este error se ignora el cálculo de la intersección de dichos semiplanos escalera y se muestra el resultado.

5.7.4.Ventana *Método de Inflado*

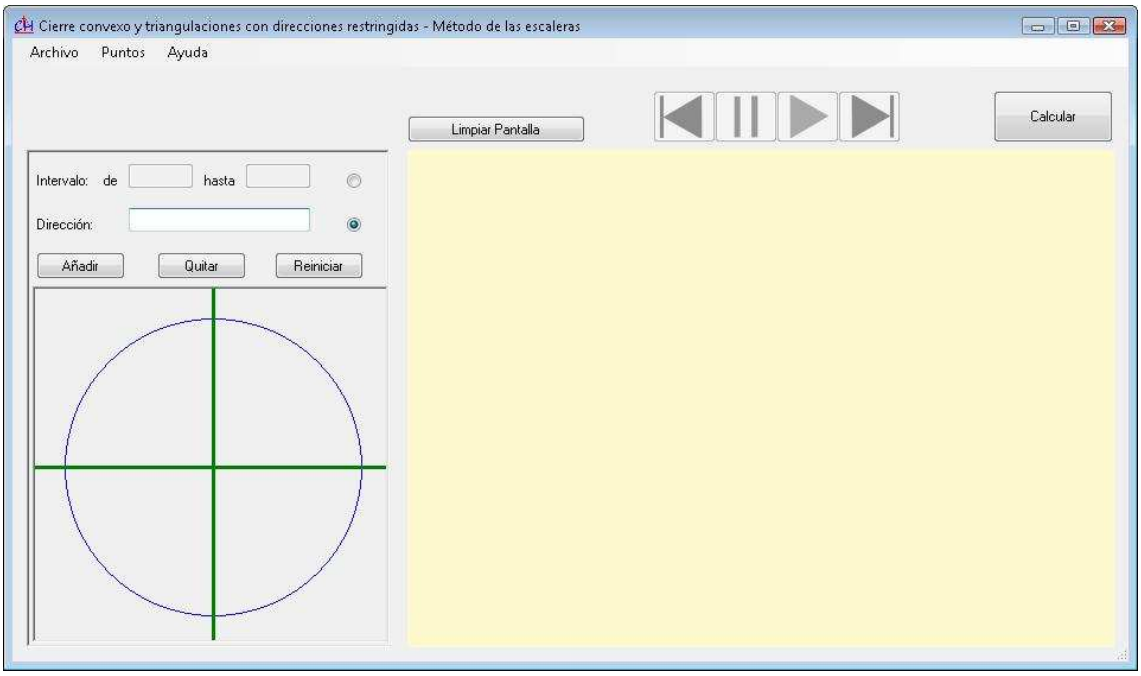


Imagen 5.10. Ventana *Método de Inflado*.

Esta ventana ofrece la posibilidad de introducir una distribución de puntos para que de dicha distribución se calcule la triangulación de Delaunay con direcciones restringidas. El proceso será mostrado mediante una animación.

El menú *Archivo* ofrece dos opciones: *Preferencias* y *Salir*. La primera permite cambiar el color del área de dibujo y la segunda salir de esta ventana, acción que también desencadenará un click sobre la X encuadrada en rojo de la esquina superior derecha.

El menú *Puntos* ofrece la posibilidad de cargar y guardar distribuciones de puntos mediante ficheros de extensión *ch*. Dichos ficheros se pueden generar guardando distribuciones introducidas con el ratón en las ventanas correspondientes a ambos métodos.

El botón *Limpiar Pantalla* elimina del área de dibujo todos los puntos introducidos limpiando también los resultados de calcular una O-triangulación.

El menú de *Ayuda* le dirigirá a este manual.

Para realizar una triangulación debe seguir los siguientes pasos:

- Introducir las direcciones pertinentes mediante el control selector de direcciones (ver sección 5.7.2.1).
- Introducir una distribución de puntos usando el ratón o cargándola de un fichero. Debe haber al menos tres puntos en el panel, en caso contrario se mostrará un mensaje informativo.
- Pulsar sobre el botón *Calcular*.
- Una vez calculada se activarán cuatro botones con los símbolos de *Play*, *Pausa*, *Avance* y *Retroceso*. A la vez aparecerá la triangulación de Delaunay con todas las direcciones de la distribución de puntos (delimitada con lineas de color rojo).
- Para iniciar la animación hay que pulsar el botón *Play*, si se desea pararla habrá que pulsar sobre el botón *Pausa*. Con la animación pausada pude avanzar o retroceder pasos manualmente mediante los botones de *Avance* y *Retroceso.*

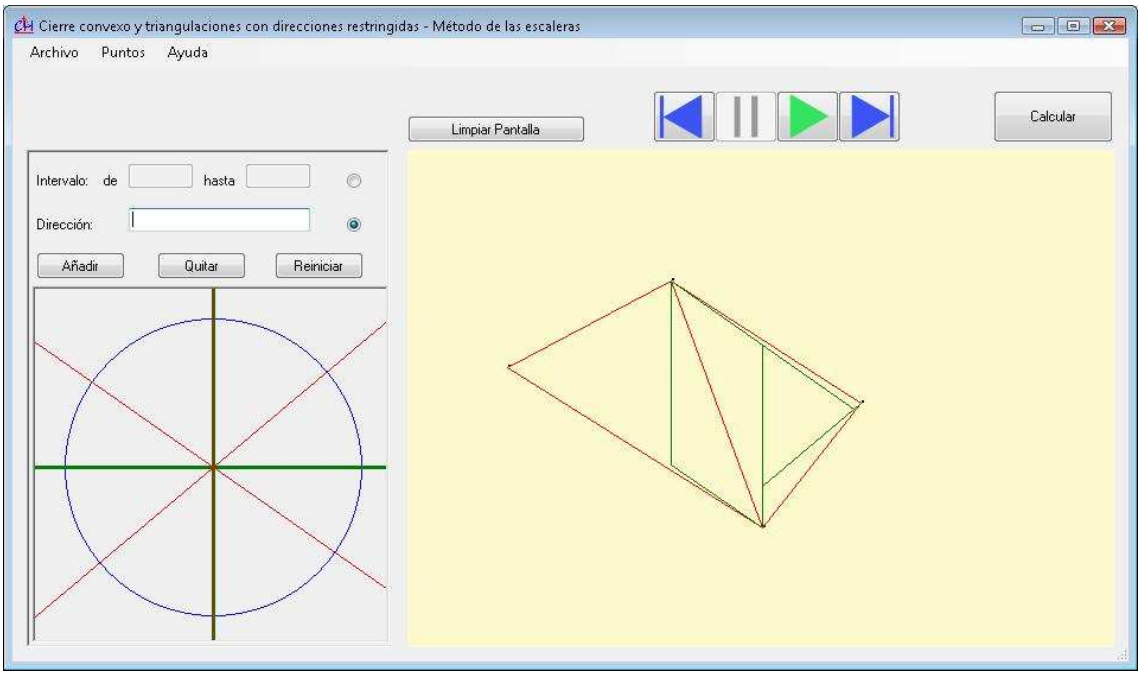


Imagen 5.11. Resultado de calcular una {40, 90, 145}-triangulación. La animación está parada en el paso final.

5.8. Manual de explotación

5.8.1. Instalación/Desinstalación de la aplicación

Será necesario instalar la aplicación en los ordenadores donde se desee usar. Los requisitos mínimos de dichos ordenadores son:

- Máquina con un procesador de 1,5 GHz como mínimo.
- 1 GB de memoria.
- Teclado y ratón.

Requisito indispensable para la instalación de la aplicación es que el ordenador de destino tenga un sistema operativo Windows instalado (se recomienda Windows XP o Windows Vista).

5.8.1.1. Instalación de la aplicación

Inserte el CD de la aplicación. Copie la carpeta ‘Aplicación’ en la ubicación que desee. Para ejecutarla simplemente haga doble click sobre el archivo .exe presente en dicha carpeta. Si lo desea puede crear un acceso directo a la aplicación de la siguiente

forma: pulse con el botón derecho sobre el archivo ejecutable, seleccione la opción “Crear acceso directo” y finalmente llévese el acceso directo donde desee.

5.8.1.2. Desinstalación de la aplicación

Para eliminar la aplicación del sistema simplemente debe borrar la carpeta ‘Aplicación’ que copió durante la instalación. Si en el proceso de instalación creó algún acceso directo éste debe ser eliminado también.

5.8.2.Actuaciones en caso de fallo

En caso de fallo lo que se debe hacer es comprobar si el error producido se encuentra contemplado en el manual de usuario.

Si no es así puede intentar solucionar el problema reiniciando el software.

En cualquiera de los dos casos póngase en contacto con el desarrollador del producto informando de la incidencia.

5.8.3.Copias de seguridad

En esta sección del manual se explicará cómo realizar copias de seguridad de los archivos de envolvente convexa y triangulación generados por el producto software.

Para realizar copias de seguridad de los mencionados ficheros hay que seguir los pasos siguientes:

1. Introducir un medio de almacenamiento con espacio libre (CD/DVD/Memoria Flash).
2. Localizar los archivos .ch y .txt que se desean salvaguardar.
3. Copiar los ficheros al medio seleccionado

Una vez hecho esto el usuario podrá llevar sus archivos a otro ordenador o bien recuperar sus archivos en caso de error en su máquina.

5.9. Conclusión de la sección

En esta quinta sección se ha llevado a cabo el análisis y diseño de una aplicación que permitirá la visualización de envolventes convexas y triangulaciones con direcciones restringidas.

La aplicación implementa los algoritmos descritos en la sección cuatro y ofrece soluciones programáticas a problemas inherentes a los mismos.

Por ello se puede concluir que en esta sección se ha satisfecho el quinto objetivo.

6. Resultados y conclusiones

6.1. Conclusiones y resultados finales

Al final de cada apartado se han ido presentando las conclusiones relativas a cada uno de ellos, no obstante en el presente apartado se van a recapitular dichas conclusiones y a mostrar los resultados del proyecto a nivel global.

En el apartado tercero se ha llevado a cabo un estudio de las definiciones de envolvente convexa y de triangulación. Además se han presentado propiedades, datos históricos y algoritmos para el cálculo tanto de envolventes convexas como de triangulaciones. Esto ha servido para comprender las dos estructuras de Geometría Computacional tratadas en el presente proyecto.

Una vez introducidos los conceptos de envolvente convexa y triangulación para el caso general se pasó a estudiar las diferentes definiciones de envolvente convexa con direcciones restringidas (envolvente O-convexa) que han sido presentadas en la literatura. Este estudio sirvió para discernir las diferentes propiedades de cada una y elegir qué definición se ajustaba mejor a las necesidades del presente proyecto. Finalmente la definición elegida fue:

Dado un conjunto de puntos, su cierre O-convexo es la intersección de todos los semiplanos escalera cerrados que lo contienen.

Con una definición elegida se pudo estudiar las particularidades de las triangulaciones formadas por O-triángulos y se pudieron presentar dos algoritmos.

Hasta este punto se puede decir que el resultado alcanzado ha sido acercar a personas que no están día a día en contacto con el mundo de las matemáticas (entre los que se incluye el autor) al peculiar concepto de las direcciones restringidas. Además se espera volver a despertar el interés de la comunidad en este tema, que se había dejado un poco aparcado.

Sin embargo, uno de los objetivos de este proyecto era superar la barrera de lo teórico y por ello se ha dado el paso siguiente: se llevó a cabo la complicada tarea de implementar los algoritmos estudiados para permitir la visualización de envolventes convexas y triangulaciones con direcciones restringidas.

El resultado de este trabajo de desarrollo es un programa que permite calcular envolventes convexas y triangulaciones con direcciones restringidas. De esta manera se hace más intuitiva la idea de direcciones restringidas y se dispone de una herramienta para su cálculo. Este software ofrece numerosas posibilidades como son: visualización paso a paso, una interfaz intuitiva mediante la cual introducir los datos de entrada, posibilidad de guardar y cargar distribuciones de puntos, posibilidad de configurar la interfaz y posibilidad de llevar a cabo una triangulación mediante el novedoso método

del inflado, que supone la forma más visual para comprender el concepto de direcciones restringidas.

Otro de los resultados tangibles del proyecto es la presente memoria en la que se incluye tanto la revisión bibliográfica llevada a cabo como la documentación técnica de la aplicación.

Se puede decir que la parte más difícil del proyecto fue la elaboración del software, pero sobre todo la construcción del módulo correspondiente al método de la intersección de semiplanos escalera. En dicho módulo se encontraron problemas que van desde la pérdida de precisión en operaciones de punto flotante, pasando por la dificultad a la hora de elegir estructuras de datos y llegando hasta problemas relativos a la complejidad inherente a los propios algoritmos.

Como conclusión global se puede decir que el proyecto ha logrado cumplir todos los objetivos inicialmente propuestos y ha conseguido la satisfacción de todas las personas implicadas en el mismo.

6.2. Futuros trabajos

Existen varias posibilidades en lo que se refiere a la realización de futuros proyectos de fin de carrera basados en éste. Una de las tareas que se podría llevar a cabo es estudiar los algoritmos y la implementación de los mismos para conocer su grado de optimización. Una vez realizado dicho estudio se podrían reformular los algoritmos o buscar nuevas estructuras y métodos de implementación que hagan más eficiente el funcionamiento del software.

Otro posible trabajo puede ser la reconstrucción del módulo del programa correspondiente al método de la intersección de semiplanos escalera para que soportara intersecciones múltiples entre dos semiplanos escalera. Actualmente sólo se soportan dos intersecciones y para más de dos se muestra un mensaje informativo, pero sería un buen trabajo replantear las estructuras de datos empleadas en la representación de los semiplanos escalera para permitir realizar más de dos intersecciones entre dos semiplanos escalera.

Además se podría utilizar este trabajo como base y herramienta para la realización de un proyecto en el que se mostraran las aplicaciones en robótica, por ejemplo, de las envolventes convexas con direcciones restringidas.

6.3. Metodología

Para la realización del estudio teórico del proyecto se ha seguido el proceso siguiente: revisión bibliográfica sobre triangulaciones y envolventes convexas, revisión bibliográfica sobre envolventes convexas con direcciones restringidas. No se ha seguido

ninguna metodología en particular de revisión bibliográfica, no obstante en la memoria se usará el método Harvard para citar las fuentes.

En lo referente al desarrollo software se ha aplicado una metodología Métrica v3, adaptada a un desarrollo unipersonal. Para adaptarla se han eliminado fases consideradas innecesarias. El ciclo de vida aplicado es, por lo tanto, en cascada.

6.4. Recursos

Para la realización de este proyecto ha sido necesario acceso a toda la documentación disponible acerca de envolventes convexas y triangulaciones, tanto para el caso sin restricciones como para el caso restringido.

Además ha sido necesario el siguiente equipamiento informático:

- Hardware: Se ha necesitado un ordenador personal en el cual se han instalado las herramientas necesarias tanto para realizar el estudio como para desarrollar el software. Además el ordenador ha necesitado acceso a Internet para poder obtener la información necesaria sobre los temas a tratar. Los ordenadores en los que se ejecute la aplicación fruto del trabajo deberán tener una potencia media-alta debido a los cálculos matemáticos que se deben realizar.

- Software: El software necesario se detalla a continuación.

  - Lenguaje de programación C++. Se ha elegido este lenguaje debido a que, para la elaboración del software, es necesario utilizar una librería matemática que sólo funciona con C++. Además este lenguaje es el más usado para desarrollar aplicaciones relativas a la Geometría Computacional por su velocidad.

  - Entorno de desarrollo Visual Studio. Se ha elegido Visual Studio como entorno de desarrollo por considerarse lo más cómodo para programar en el lenguaje seleccionado. Además ofrece un diseñador de interfaces de usuario que permitirá ahorrar mucho tiempo a la hora de crear las interfaces.

  - .NET Framework (en el ordenador que ejecute la aplicación). Se ha recurrido a .NET debido a los componentes que ofrece para las interfaces. Además como

C++ es uno de los lenguajes .NET ofrece una perfecta integración con el resto de herramientas a utilizar.

- Librería matemática CGAL. Para realizar operaciones geométricas complejas ha sido necesario recurrir a la librería matemática CGAL, escrita en C++.
- Sistema Operativo Windows (XP o Vista)
- Paquete de ofimática para elaborar informes y presentaciones.

7. Bibliografía

Abellanas, M. (2006), "Envolvente convexa, triangulación de Delaunay y diagrama de Voronoi: tres estructuras geométricas en una, con muchas aplicaciones", *Un Paseo por la Geometría*, capítulo 10, 157-172. http://divulgamat.ehu.es/weborriak/TestuakOnLine/06-07/PG-06-07-Abellanas.pdf

Berg, M. de., Kreveld, M. van., Overmars, M., Schwarzkopf O. (2000), *Computational Geometry, Algorithms and Applications*, Springer.

Bern, M. (2004), "Triangulations and mesh generation". En Goodman, J. E., O'Rourke, J., eds. , *Handbook of Discrete and Computational Geometry*, capítulo 25, 563-582, CRC Press. http://books.google.es/books?id=X1gBshCclnsC&dq=handbook+discrete+computational+geometry&pg=PP1&ots=CR_wFIU8jh&sig=q_CPvN_adkNdQmiwlGcPYI4lU0A&hl=es&sa=X&oi=book_result&resnum=4&ct=result#PPP1

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Clifford, S. (2001), *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill. capítulo 33.3, 947–957.

Durán, J. (?), "Triangulaciones de polígonos", *Matemática Aplicada I*, Universidad de Sevilla.

Felzenszwalb, P. F., Huttenlocher,D. P., (2005), "Pictorial structures for object recognition", *International Journal of Computer Vision*, volumen 61.

Fink, E., Wood, D. (2003), "Planar strong visibility". *International Journal of Computational Geometry and Applications*, 13(2), 173-187.

Fink, E., Wood, D. (2004), *Restricted-orientation convexity*, Springer.

Foscari, P. (2007), "The Realtime Raytracing Realm", *ACM Transactions on Graphics*.

Martynchik, V., Metelski, N., Wood, D. (1996), "O-convexity: Computing hulls, approximations, and orientation sets". *Proceedings of the Eighth Canadian Conference on Computational Geometry*, 2-7.

Martinsky, O. (2007), *Algorithmic and mathematical principles of automatic number plate recognition systems*, Brno Univesity of Technology.

Meeran, S., Shafie, A. (1991), *Optimum path planning using convex hull and local search heuristic algorithms*, Editorial Elsevier Science, Oxford.

Montuo, D. Y., Fournier, A. (1982), "Finding the x – y convex hull of a set of x – y polygons", *CSRG Technical Report*, 148.

Nicholl, T. M., et al. (1983), “Constructing the X – Y convex hull of a set of X – Y polygons”, *BIT*, 23, 456-471.

Nilsson, B. J., Ottman, T., Schuierer, S., Icking, C. (1992), *Restricted orientation Computational Geometry*. Lecture Notes in Computer Science, capítulo 594, 148-185.

O’Rourke, J. (1994), *Computational Geometry in C*, Cambridge University Press.

Ortega, L. (2006), “Tema 3: La envolvente convexa”, *Geometría Computacional*, Universidad de Jaen.

Ottman, T. (1984), “On the Definition and Computation of Rectilinear Convex Hulls”. *Information Sciences*, 33, 157-171.

Ottman, T., Soisalon-Soininen, E., Wood, D. (1983), “Rectilinear convex hull partitioning of sets of rectilinear polygons”, *Computer Science Department Technical Report*.

Preparata, F. P., Hong, S. J. (1977), “Convex Hulls of Finite Sets of Points in Two and Three Dimensions”, *Commun ACM*, vol. 20, no. 2, 87-93.

Preparata F. P., Shamos M. I. (1985), *Computational Geometry*, capítulo 4, Springer-Verlag, New York.

Rawlins, G. J. E., Wood, D. (1987), “Optimal Computation of Finitely Oriented Convex Hulls”. *Information and Computation*, 72, 150-166.

Rawlins, G. J. E., Wood, D. (1991), “Restricted-oriented convex sets”. *Information sciences*, 54, 263-281.

Seidel, R. (2004), “Convex hull computations”. En Goodman, J. E., O'Rourke, J., eds., *Handbook of Discrete and Computational Geometry*, capítulo 22, 495-512, CRC Press. http://books.google.es/books?id=X1gBshCclnsC&dq=handbook+discrete+computational+geometry&pg=PP1&ots=CR_wFIU8jh&sig=q_CPvN_adkNdQmiwlGcPYI4lU0A&hl=es&sa=X&oi=book_result&resnum=4&ct=result#PPP1

Shamos, M. (1975), "Problems in Computational Geometry", *Computational Geometry*, Carnegie Mellon University.

Shamos, M. (1978), *Computational Geometry*, Tesis Doctoral, Departamento de Ciencias de la Computación de la Universidad de Yale.

Shreiner, D., Woo M., Neider, J. (2007), *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, Paperback.

Toussaint, G. T. (1983), "A Counterexample to an Algorithm for Computing Monotone Hulls of Simple Polygons", *Pattern Recognition Letters*¸ vol. 1, capítulo 4, página 219.

Vogel, K. (1997), *A Surveying Problem Travels from China to Paris*, Yvonne Dold-Samplonius Editorial.