UNIVERSIDAD POLITÉCNICA DE MADRID

FACULTAD DE INFORMÁTICA

TRABAJO FIN DE CARRERA

POLÍGONO DE VISIBILIDAD DESDE UN LADO

AUTOR: Alberto Isidro González Díaz-Carralero
TUTOR: Gregorio Hernández Peñalver

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE FIGURAS

AGRADECIMIENTOS

En primer lugar quiero mostrar mi agradecimiento al tutor, Gregorio, por su paciencia y por darme la oportunidad de realizar este trabajo fin de carrera.

Gracias también a todos los que me animaron a terminarlo, familiares y amigos.

Por último agradecer a todos los internautas que de forma desinteresada crean guías de ayuda y herramientas, o resuelven problemas en los foros, y que tan útiles son para resolver las dudas cotidianas que siempre surgen en el mundo de la informática. Mención especial a Francisco José Cortijo Bon y Fernando Berzal Galiano por su "Curso de C++ Builder", a Marshall Cline por su "guía para mezclar C y C++" y a los desarrolladores de la herramienta de autoría de ayuda Helpmaker.

1. INTRODUCCIÓN Y OBJETIVOS

En este capítulo empezamos presentando la parte de la ciencia en la que se enmarca el trabajo realizado, la *geometría computacional*. A continuación introducimos el campo de la *visibilidad*. Al final, describimos los objetivos del trabajo realizado.

La geometría es la disciplina matemática que estudia el espacio y las formas (figuras y cuerpos) que en él se pueden imaginar. La *geometría computacional* surge con el desarrollo de los ordenadores que por su capacidad de almacenamiento y potencia de cálculo permiten centrarse en el diseño de los algoritmos geométricos al simplificarse el tratamiento de grandes volúmenes de datos y alto número de operaciones. Por ejemplo, se pueden afrontar métodos recursivos e iterativos, que eran inviables por el tiempo que había que invertir en ellos.

Las características buscadas en los algoritmos diseñados en geometría computacional son por un lado robustez, ya que se tratan problemas continuos en un dominio discreto lo que puede conllevar la degeneración de resultados, como por ejemplo al manejar operaciones en coma flotante. Por otro eficiencia ya que es necesario que los resultados se obtengan en un tiempo razonable. Esto se traduce en lograr unos algoritmos de complejidad óptima. Por último la eficacia que consiste en que el método resuelva el problema para todos los casos, si bien hay ocasiones en que se hace necesario acotar los problemas para conseguir mejorar los resultados en cuanto a eficiencia y robustez. Siguiendo la notación de Knuth en cuanto a complejidad de algoritmos, indicamos con $O(f(n))$ que la complejidad es como mucho del orden de f a partir de un cierto n (cota superior), con $\Omega(f(n))$ indicamos la cota inferior y con $\theta(f(n))$ que es óptima.

Las áreas de aplicación de la geometría computacional son diseño y fabricación asistidos por ordenador (Computer Aided Design / Computer Aided Manufacturing), computación gráfica, integración a escala muy grande (Very Large Scale Integration), robótica, gestión y estadística entre otras.

Dentro de la geometría computacional uno de los campos más importantes por las aplicaciones que tiene es el de *visibilidad*. Dos puntos interiores de un polígono son visibles si el segmento que los une cae enteramente dentro del polígono. La noción de visibilidad ha dado lugar a un gran número de cuestiones algorítmicas. A continuación presentamos las más relacionadas con el trabajo realizado.

POLÍGONO DE VISIBILIDAD DESDE UN LADO

El problema más fundamental de visibilidad consiste en: dado un punto *x* de un polígono *P*, determinar *V(x)*, la parte de *P* visible desde *x*. *V(x)* es llamado el *polígono de visibilidad desde un punto* en x. Se han conseguido algoritmos que resuelven este problema en tiempo no lineal (Davis y Benedikt 1979) y lineal (El-Gindy y Avis 1980; Lee 1983). El algoritmo de El-Gindy y Avis requiere tres pilas y es bastante complicado. El de Lee requiere solamente una pila. Posteriormente, Joe y Simpson (1985) simplificaron la organización del algoritmo de Lee. Permitir agujeros en el polígono conduce a una complejidad $\Omega(n \log n)$. En tres dimensiones, el cálculo de *V(x)* es el problema altamente estudiado de *eliminación de la línea escondida* que resulta tener una complejidad $\theta(n^2)$ (McKenna 1987).

Posteriormente, Avis y Toussaint (1981) extendieron la noción de visibilidad a un lado. Sea *P* un polígono y *e=(a,b)* un lado de *P*.

1. *P* es *completamente visible* desde *e* si cada punto de *P* es visible desde cada punto de *e*.
2. *P* es *fuertemente visible* desde *e* si hay al menos un punto de *e* que puede ver todo *P*.
3. *P* es *débilmente visible* desde *e* si cada punto de *P* es visible desde algún punto de *e*.

Es fácil ver que la región completamente visible desde *e* es la intersección de *V(a)* y *V(b)*. Estos polígonos de visibilidad desde un punto pueden ser construidos en tiempo *O(n)* y su intersección construida en tiempo *O(n)* fácilmente. No hay una única región fuertemente visible desde *e*; más bien hay muchas regiones visibles fuertemente desde un lado. De los tres problemas planteados, solamente el último representa un problema algorítmico interesante.

Sea *V(a,b)* la región poligonal de un polígono *P* débilmente visible desde un lado *e=(a,b)*. Podemos imaginar *V(a,b)* (*polígono de visibilidad débil*) como la parte de *P* iluminada por un tubo fluorescente colocado a lo largo del lado *e*. Se han presentado tres algoritmos para construir *V(a,b)* en tiempo *O(n log n)* de forma independiente y casi simultánea por El-Gindy (1985), Lee y Lin (1986) y Chazelle y Guibas (1985). Guibas et al. (1986) mostraron que este problema puede resolverse en tiempo *O(n)* si disponemos de una triangulación de *P*. Como podemos triangular *P* utilizando el algoritmo de Chazelle (1991) en tiempo *O(n)* el algoritmo de Guibas et al. (1986) se ejecuta en tiempo *O(n)*.

El algoritmo central que se trata en este trabajo es el presentado por Lee y Lin (1986) para calcular *V(a,b)*. En el trabajo desarrollamos una implementación de dicho algoritmo

de complejidad *O(n log n)*. Para ello nos apoyamos en una estructura de datos llamada (Aho, Hopcroft y Ullman 1974) *cola concatenable*, para la cual desarrollamos una implementación particular que hemos llamado *árbol-pila binario*. Desarrollamos una segunda implementación para la *cola concatenable* con una *lista* para poder mostrar los estados intermedios del algoritmo y poder analizarlo paso a paso. Esta segunda implementación al realizar búsquedas secuenciales en lugar de binarias eleva la complejidad del algoritmo total a $O(n^2)$. Presentamos un algoritmo para calcular el *polígono de visibilidad desde un vértice*, basado en el algoritmo de Lee y Lin para un lado. Este algoritmo tiene complejidad lineal. Implementamos igualmente este algoritmo basándonos en la implementación anterior para un lado. Por último hemos desarrollado un algoritmo para pasar un polígono a *forma estándar*.

2. ALGORITMOS

En este capítulo recogemos el algoritmo central del trabajo, así como el resto de algoritmos desarrollados alrededor del mismo. Empezamos introduciendo algunos términos básicos.

2.1. Definiciones

Definición 1. Un polígono simple *P* es una figura plana y cerrada cuyo contorno *bd(P)* está formado por segmentos y que se especifica como una secuencia de vértices $v_0, v_1,..., v_n=v_0$, donde $\overline{v_i, v_{i+1}}$, *i=0, 1,..., n-1*, es un lado de *bd(P)* y no hay dos lados no consecutivos cortándose. Un polígono simple es *orientado* si los lados de *P* están orientados de forma que el interior de *P*, indicado *int(P)* cae a la derecha según los recorremos. Un *segmento* de *bd(P)* es un segmento indicado como *e(u,v)*, donde *u* y *v* son los puntos finales del segmento sobre *bd(P)*. (Un lado es un segmento tal que los puntos finales están en los vértices.) En adelante, el símbolo *P* indica el conjunto unión de *int(P)* y *bd(P)*.

Definición 2. Para dos puntos cualesquiera *u* y *v* sobre *bd(P)*, el trozo de *bd(P)* recorrido cuando viajamos de *u* a *v* con *int(P)* a la izquierda lo llamamos *cadena izquierda*, indicado *LCH(u,v)*; el trozo de *bd(P)* recorrido cuando viajamos de *u* a *v* con *int(P)* a la derecha lo llamamos *cadena derecha*, indicado *RCH(u,v)*. Una cadena *LCH(u,v)* o *RCH(u,v)*, junto el

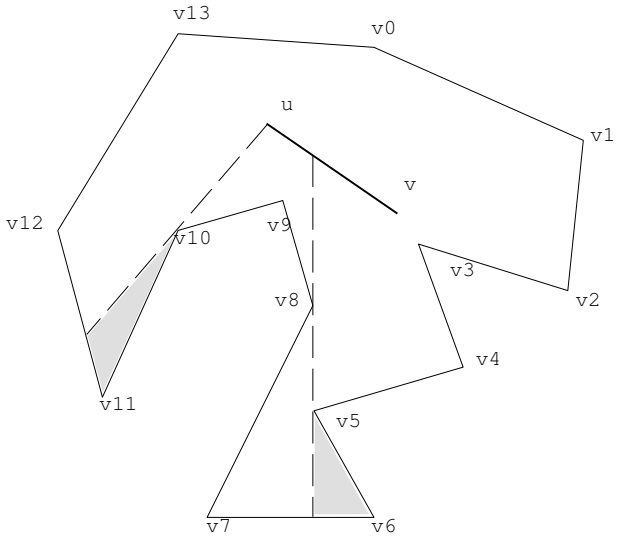

Figura 1. Polígono de visibilidad desde $\overline{u, v}$, parte no sombreada.

segmento $\overline{u,v}$, se dice que forma una región llamada *lóbulo* si la cadena no corta un punto interior de $\overline{u,v}$. El lóbulo se dice que es un *lóbulo izquierdo* o *derecho*, indicado, respectivamente, *LLB(u,v)* o *RLB(u,v)*, dependiendo del tipo de su cadena asociada.

Definición 3. Un punto *q* de *P* se dice que es *(débilmente) visible* desde un segmento $\overline{u,v}$ de *P* y viceversa, si existe un punto *r* sobre $\overline{u,v}$ tal que el segmento que conecta *q* y *r* se encuentra completamente en *P*. La región *VIS(u,v)*, conjunto de todos los puntos visibles desde $\overline{u,v}$ se llama *polígono de visibilidad* desde $\overline{u,v}$, es decir, *VIS(u,v)*={*r*|*r* en *P* y visible desde $\overline{u,v}$ }, y es un subconjunto de *P*; y $\overline{u,v}$ es conocida como un *ancla* de *P* (Figura 1).

Sin pérdida de generalidad consideraremos polígonos orientados siendo el ancla un lado del contorno en posición horizontal y el resto de los vértices encontrándose debajo del ancla. Esto es, sea *P* un polígono orientado, indicado $v_0, v_1, \ldots, v_n=v_0$, donde $\overline{v_0,v_1}$ es el ancla y ninguno de los vértices, v_i, $i=2, 3, \ldots, n-1$, está por encima de la línea que contiene $\overline{v_0,v_1}$. Se dice que el polígono *P* está en *forma estándar* (Figura 2). En el caso de que el ancla $\overline{u,v}$ es un lado pero los vértices de *P* puedan encontrarse en lados distintos de la línea que contiene $\overline{u,v}$, el problema se puede reducir a calcular polígonos de visibilidad con los puntos *u* y *v* siendo el ancla y un caso donde el polígono está en forma estándar dibujando un línea a través de $\overline{u,v}$ y cortando *P* en trozos que quedan a ambos lados de la línea. En la Figura 3 podemos calcular los polígonos de visibilidad desde los puntos *u* y *v* en P_1 y P_2, respectivamente, en tiempo lineal; ignorar el área sombreada y tenemos el polígono P_3 en forma estándar. El caso en el que el ancla es un segmento interior a *P* puede reducirse a dos ejemplos del problema donde el ancla es (una parte de) un lado del

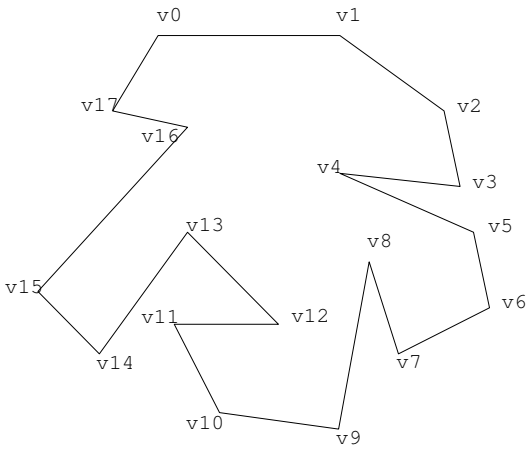

Figura 2. Un polígono en forma estándar.

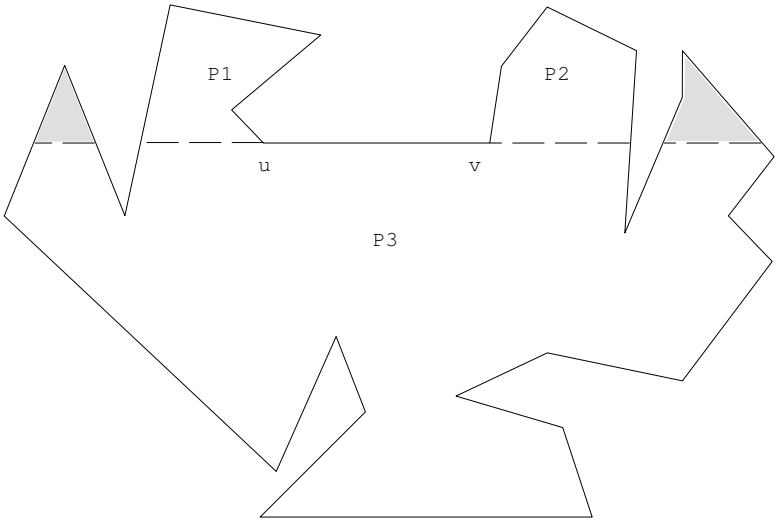

Figura 3. Convirtiendo *P* en P_1, P_2 y P_3.

polígono. En adelante, se asume que el polígono *P* está en forma estándar con $\overline{v_0, v_1}$ como el ancla.

Definición 4. Un punto *q* sobre $RCH(v_1,v_0)$ de $bd(P)$ es visible desde un punto *r* del ancla $\overline{v_0, v_1}$ con respecto a $RCH(v_1,q)$ si el segmento $\overline{r, q}$ no corta el interior de $RCH(v_1,q)$; la *interceptación derecha* R_q de *q* es el punto más a la derecha *r* sobre $\overline{v_0, v_1}$ tal que *r* es visible desde *q* con respecto a $RCH(v_1,q)$ y la *interceptación izquierda* L_q de *q* es el punto más a la izquierda *u* sobre $\overline{v_0, v_1}$ tal que *u* es visible desde *q* con respecto a $LCH(v_0,q)$, donde la visibilidad de un punto con respecto a $LCH(v_0,q)$ se define similarmente (Figura 4). La interceptación derecha o izquierda de *q* puede no existir si no hay ningún punto

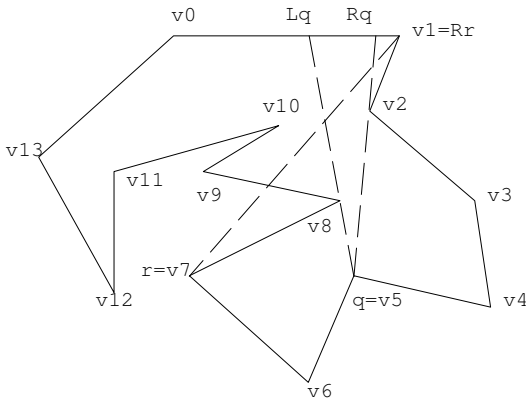

Figura 4. Interceptación derecha e izquierda del vértice q. $\overline{r, R_r}$ no corta $RCH(v_1,r)$ pero puede cortar $LCH(v_0,r)$.

sobre $\overline{v_0, v_1}$ tal que sea visible desde q con respecto a $RCH(v_1,q)$ o $LCH(v_0,q)$, respectivamente. El segmento $e(L_q,R_q)$, un trozo de $\overline{v_0, v_1}$, se conoce como *rango visible* de q, indicado $VR(q)$.

Definición 5. Dados tres puntos $p_i=(x_i,y_i)$, $i=1, 2, 3$, sea $s=x_3(y_1-y_2)+y_3(x_2-x_1)+y_2x_1-y_1x_2$. Decimos que $p_1p_2p_3$ es un giro *derecho* si s es negativo; $p_1p_2p_3$ es un giro *izquierdo* si s es positivo. En otras palabras, si p_3 se encuentra a la derecha del vector $\overrightarrow{p_1, p_2}$ entonces $p_1p_2p_3$ es un giro derecho; si p_3 se encuentra a la izquierda del vector $\overrightarrow{p_1, p_2}$ es un giro izquierdo.

Definición 6. Dado P en forma estándar con $\overline{v_0, v_1}$ como el ancla y un punto q sobre $RCH(v_1,q)$, la secuencia de vértices $v_1,...,q$ es llamada el *camino convexo derecho* desde v_1 a q, indicado $RCVX(v_1,q)$, tanto si el camino tiene dos vértices y q es visible desde v_1 con respecto a $RCH(v_1,q)$ como si el camino tiene tres o más vértices y cada tres vértices consecutivos empezado desde v_1 del camino forman un giro *izquierdo*. Similarmente definimos el *camino convexo izquierdo* desde v_0 a cualquier punto q sobre $LCH(v_0,q)$, indicado $LCVX(v_0,q)$, excepto que cada tres vértices consecutivos empezado desde v_0 del camino forman un giro *derecho*. Para cualesquiera dos vértices consecutivos sobre un camino convexo, el segmento que determinan es llamado un *pseudo-lado* si no es un lado de P (Figura 5).

Definición 7. Sea $\overline{u, v}$ un lado de P sobre $RCH(v_1,v_0)$. Sea $h(u,v)$ el semiplano que determina $\overline{u, v}$ e $int(P)$; es decir, el interior de P según recorremos $\overline{u, v}$ está contenido en $h(u,v)$. La parte del ancla que cae en $h(u,v)$ es llamada el *rango visible* de $\overline{u, v}$, indicado $VR(u,v)$. Si $VR(u,v)$ está vacío, el lado, excluyendo el vértice final v, es llamado un lado *trasero*; de otra forma, es un lado *frontal*.

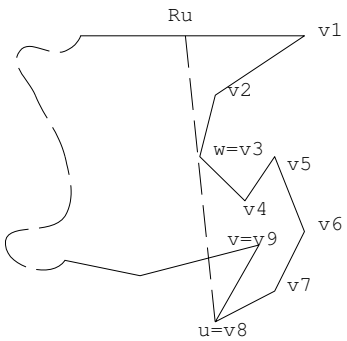

Figura 5. $\overline{v_3, v_8}$ es un pseudo-lado y (v_1,v_2,v_3,v_8) es un camino convexo derecho desde v_1 a v_8.

2.2. Polígono de Visibilidad desde un Lado

El algoritmo para calcular el *polígono de visibilidad desde un lado VIS(v_0,v_1)* de un polígono *P* en forma estándar consta de dos pasos. Primero aplicamos el algoritmo RIGHTSCAN, calculando el camino convexo derecho *RCVX(v_1,v)* para cada vértice *v* según recorremos los vértices de *P* en sentido *horario*. Sobre el polígono resultante *P'* aplicamos el algoritmo LEFTSCAN, calculando esta vez el camino convexo izquierdo *LCVX(v_0,v')* para cada vértice *v'* según recorremos los vértices de *P'* en sentido *contrahorario*. Los algoritmos RIGHTSCAN y LEFTSCAN son simétricos, es decir, que son idénticos considerando los giros derechos de uno como izquierdos en el otro y viceversa. Durante la aplicación de ambos algoritmos se descarta la parte de *bd(P)* y *bd(P')* que no es visible, borrándose algunos vértices de *P* y *P'* y añadiéndose otros nuevos. Obtenemos *así P''* que es el polígono de visibilidad buscado.

2.2.1 Algoritmo RIGHTSCAN

A continuación describiremos el algoritmo RIGHTSCAN.

Mantenemos un camino convexo derecho *RCVX(v_1,v)* para cada vértice *v*, guardando los vértices del camino en una pila *S*. Por razones de eficiencia la pila es implementada como una cola concatenable, que indicamos *Q(S)*. Para cada vértice analizado habrá una cola concatenable asociado con él. El manejo y propósito de estas colas se verá más adelante. Supongamos que el contenido de la pila en un determinado momento es *(u_0, u_1,..., u_m)*, donde $u_0=v_1$, u_m es la cima, y los u_i's están en *RCVX(u_0,u_m)* en ese orden. Por estar u_1, u_2,...,u_m en un camino convexo, sus interceptaciones derechas $R_{u_1}, R_{u_2}, ..., R_{u_m}$, están ordenadas de derecha a izquierda, es decir, $v_1 \geq R_{u_1} \geq R_{u_2} \geq ... \geq R_{u_m}$. El contenido actual de la pila, *S=(u_0, u_1,..., u_m)*, se conoce como el *estado en v*, que es el siguiente vértice a ser analizado. Seguidamente describimos el proceso para obtener el siguiente estado desde el actual. El proceso se repite hasta que el estado en v_0 es alcanzado. El estado inicial es *S=(v_1,v_2)* en v_3. Tenemos los siguientes tres casos dependiendo de la posición del próximo vértice v a ser analizado.

(i) u_{m-1},u_m,*v* es un giro derecho. En este caso distinguiremos dos subcasos.

(a) (Figura 6a). Asociado con u_m hay un lóbulo izquierdo *LLB(u_m,r)* para algún r sobre *LCH(u_m,v_1)* y *v* se encuentra dentro de *LLB(u_m,r)*. (La existencia de *r* se puede producir a partir del caso (iii)(a).) En este caso u_m es invisible así como los

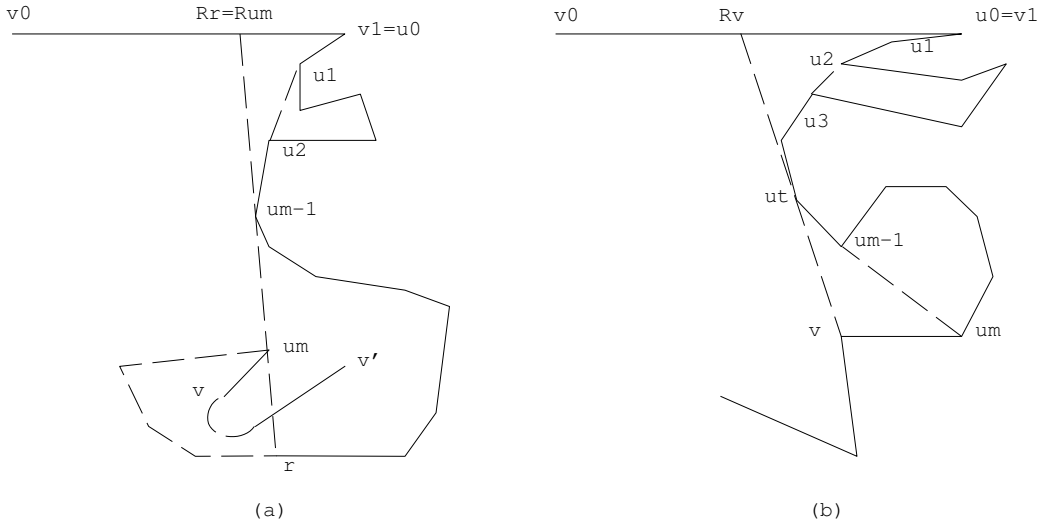

Figura 6. u_{m-1},u_m,v es un giro derecho.

vértices de P que siguen a v y se encuentran dentro de $LLB(u_m,r)$. Borramos esos vértices hasta encontrar un vértice v' que no está en $LLB(u_m,r)$. El estado en v' es $S'=(u_0, u_1,..., u_{m-1}, r)$ y tenemos el caso (iii)(a).

(b) (Figura 6b). En este caso el camino convexo necesita ser actualizado. Hacemos una búsqueda binaria sobre $Q(S)$ para encontrar el vértice que es visible desde v y está más próximo a v_1 por el camino convexo. Sea u_t tal vértice. El nuevo estado en el siguiente vértice a v en $RCH(v_1,v_0)$, es $S'=(u_0, u_1,..., u_t, v)$, y $\overline{u_t,v}$ es un pseudo-lado, es decir, u_t y v definen un lóbulo derecho $RLB(u_t,v)$. R_v es la intersección del ancla y $\overrightarrow{v,u_t}$. Por eficiencia la otra parte de la cola, es decir, $(u_{t+1},..., u_m)$, es almacenada en $Q(v)$. A partir de $Q(v)$ y $Q=Q(S')-(cima)$, la cola del nuevo estado con el elemento cima eliminado, podemos reproducir el contenido del estado previo concatenándolos, es decir, $(u_0, u_1,..., u_m)=Q \,||\, Q(v)$, donde el símbolo || significa concatenación.

Figura 7. u_{m-1},u_m,v son colineales.

(ii) (Figura 7). u_{m-1},u_m y v son colineales. Este caso se trata como si u_{m-1},u_m,v fuera un giro derecho. R_v es la misma que R_{u_m}, y el nuevo estado es $(u_0, u_1,..., u_{m-1}, v)$. (El caso en el que v se encuentra sobre $\overline{u_m,r}$ de $LLB(u_m,r)$ es considerado como si v estuviera dentro de $LLB(u_m,r)$ también.)

(iii) u_{m-1},u_m,v es un giro izquierdo. En este caso distinguimos dos subcasos.

(a) (Figura 8a). El segmento $\overline{u_{m-1},u_m}$ es un pseudo-lado y v se encuentra en $RLB(u_{m-1},u_m)$. u_m es invisible. Así, retrocedemos por $bd(P)$ empezando con u_m y para cada vértice w por el que pasamos hacemos lo siguiente. Restablecemos el

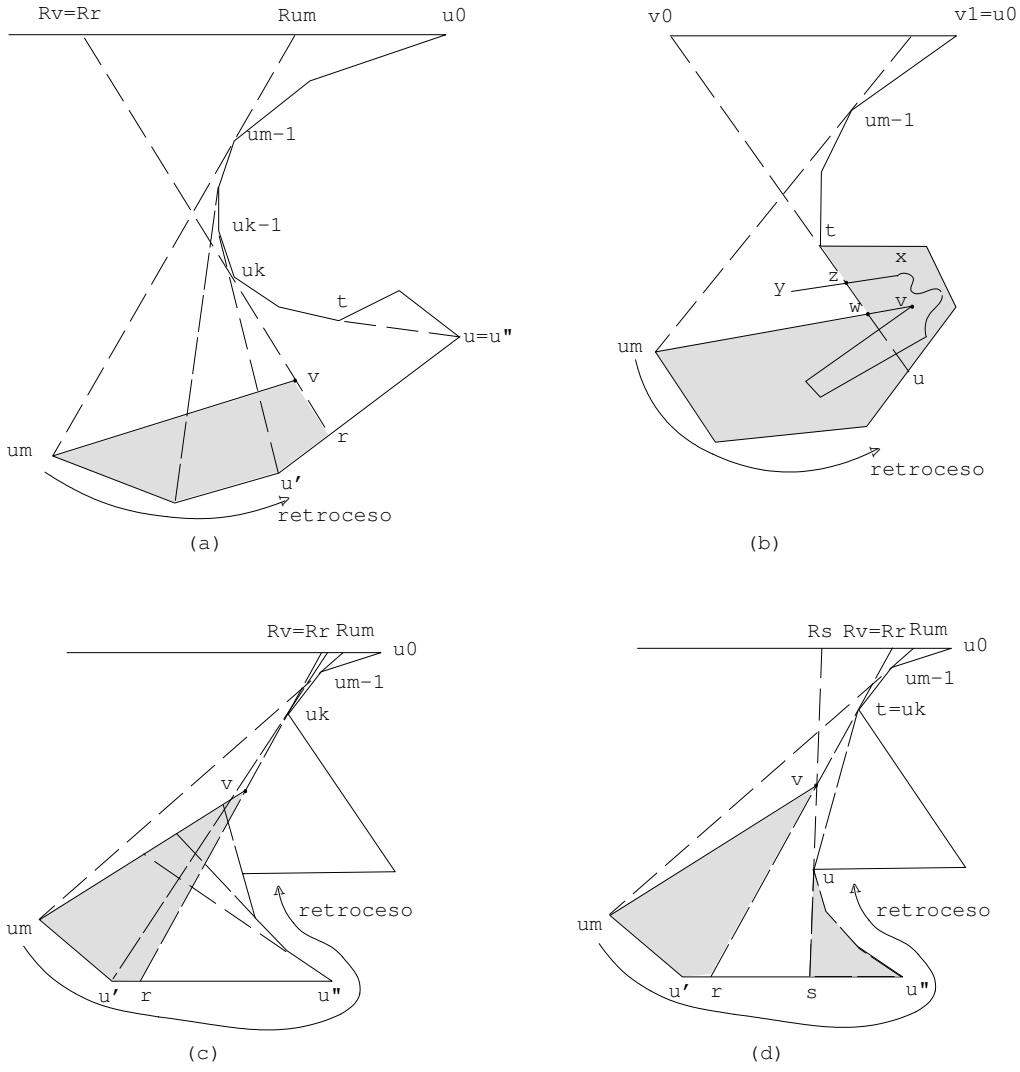

Figura 8. u_{m-1},u_m,v es un giro izquierdo y v en $RLB(u_{m-1},u_m)$. Lados en $LCH(u_m,u)$ invisibles cuando $\overline{t,u}$ corta $\overline{u_m,v}$ y cálculo de s. Partes sombreadas son eliminadas pues son invisibles.

estado en w concatenando $Q(w)$ y $Q=Q(S)-(cima)$. Sea el estado en w $S'=(u_0, u_1,..., t, u)$. Si t,u,v no es un giro derecho, repetimos el proceso de restablecimiento, excepto en el caso, mostrado en la Figura 8b, en el cual $\overline{t,u}$ es un pseudo-lado especial; es decir, la región definida por $RLB(t,u)$ es enteramente invisible (ver (iii)(b)). Los vértices por los que retrocedemos son invisibles y son borrados, pero los lados en $LCH(u_m,u)$ sólo podemos asegurar que son invisibles cuando $\overline{t,u}$ corta el lado $\overline{u_m,v}$. Sea u' el último vértice u en el que se produce este corte durante el retroceso. Sea u'' el vértice anterior a u', que a su vez es la cima del estado en u' (Figura 8c). Cuando t,u,v es un giro derecho, hacemos una búsqueda binaria sobre $Q(S')$ para encontrar el vértice u_k visible desde v y que está más próximo a v_1 por el camino convexo. Sea r la intersección de $\overline{u',u''}$ y $\overrightarrow{u_k,v}$. Entonces el siguiente estado es $S''=(u_0, u_1,..., u_k, v)$, $Q(v)=(r)$ y $Q(r)$ es $(u_{k+1},..., t, u)$ salvo si el vértice u tal que t,u,v es un giro derecho no es u''. En este caso sea s la intersección de $\overline{u',u''}$ y $\overrightarrow{v,u}$. $Q(r)$ es $(u_{k+1},..., t, u, s)$, $Q(s)=()$ y R_s la intersección de $\overrightarrow{u,v}$ y el ancla (Figura 8d). R_v y R_r toman el valor de la intersección de $\overrightarrow{v,u_k}$ y el ancla. (Aquí los puntos r,v y u_k son colineales y hemos tratado v como si estuviera a la derecha de $\overrightarrow{u_k,r}$.) Notar que tenemos un lóbulo izquierdo $LLB(v,r)$ asociado con el elemento cima de S'' (pudiendo ser origen de un caso (i)(a)). Los vértices $u_m,..., u'$ en $LLB(v,r)$ son invisibles desde el ancla y son borrados. Para el caso donde $\overline{t,u}$ es un pseudo-lado especial, hacemos lo siguiente. Encontramos la intersección w de $\overline{t,u}$ y $\overline{u_m,v}$ e ignoramos todos los vértices que siguen a v hasta encontrar un lado $\overline{x,y}$ que corta $\overline{t,w}$. Sea z la intersección de $\overline{x,y}$ y $\overline{t,w}$. De este modo tenemos un estado $(u_0, u_1,..., t, z)$ en y y tenemos el caso (i)(b).

(b) (Figura 9a). En este caso el estado S puede ser extendido a $S'=(u_0, u_1,..., u_m, v)$. No obstante, ya que sabemos que si $\overline{u_m,v}$ es un lado trasero, v puede ser borrado, podemos comprobar si v_0,u_m,v es un giro izquierdo (Figura 9b). Si no, podemos extender el estado como hemos comentado. De otra manera, borramos v y buscamos un lado $\overline{u,w}$ tal que v_0,u_m,u es un giro izquierdo pero v_0,u_m,w no lo es; es decir, el lado $\overline{u,w}$ corta $\overrightarrow{v_0,u_m}$. Sea t la intersección del lado $\overline{u,w}$ y $\overrightarrow{v_0,u_m}$. Tenemos un nuevo estado $S'=(u_0, u_1,..., u_m, t)$ en w y tenemos el caso (i)(b). R_t toma el valor V_0. Notar que en este caso $\overline{u_m,t}$ es un pseudo-lado especial en el que la región definida por $RLB(u_m,t)$ es enteramente invisible.

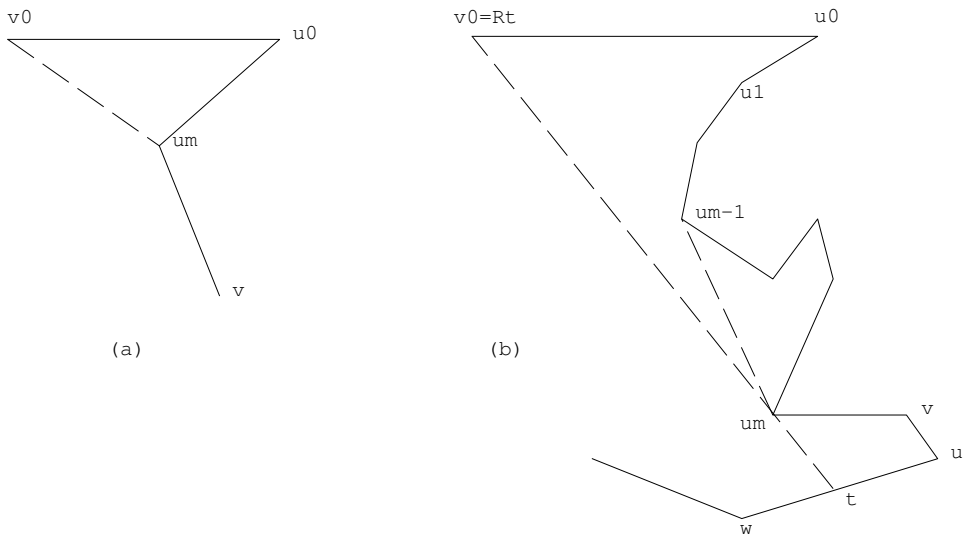

Figura 9. u_{m-1},u_m,v es un giro izquierdo y v no se encuentra en $RLB(u_{m-1},u_m)$.

2.2.2 Pseudo-código algoritmo RIGHTSCAN

A continuación hacemos una descripción detallada del anterior algoritmo. Empezamos explicando el pseudo-código utilizado. Presentamos después los bloques de código con las líneas numeradas correspondientes al algoritmo y subrutinas utilizadas seguido cada uno de sus comentarios particulares.

Los vértices del polígono de partida están enlazados en ambos sentidos, indicando $SIG(v_i)$ y $ANT(v_i)$ los vértices v_{i+1} y v_{i-1}, respectivamente, donde $i=1, 2,..., n-1$. Definimos las funciones $GIRO\_D(p_1,p_2,p_3)$ y $GIRO\_I(p_1,p_2,p_3)$ que devuelven valor verdadero si los puntos p_1,p_2,p_3 forman un giro derecho o izquierdo, respectivamente, y falso en caso contrario. La función $CORTE(p_1,p_2,p_3,p_4)$ devuelve el punto intersección del lado $\overline{p_1, p_2}$ de la línea prolongación del punto p_3 al punto p_4. La función $INSERTAR\_V(a,v,b)$ inserta un nuevo vértice v en el polígono entre los vértices a y b.

Sobre la pila S, que implementamos como cola concatenable $Q(S)$, tenemos las operaciones asociadas *PUSH* y *POP*. $PUSH(v,S)$ significa añadir v como cima de la pila S y $POP(S)$ borrar el elemento cima de S. Las correspondientes operaciones sobre colas concatenables son *insertar* el elemento v en $Q(S)$ y *borrar* el elemento cima de $Q(S)$, respectivamente. Asociadas a las colas concatenables tenemos las operaciones *SPLIT* y *MERGE*. La primera la unimos a la función *FIND*, de forma que $FIND\_SPLIT(v,S)$ significa encontrar el vértice u_t en $Q(S)=(u_0,..., u_t,..., u_m)$, que es visible desde v y está más próximo a u_0 por el camino convexo, haciendo $Q1=(u_0,..., u_t)$ y $Q2=(u_{t+1},..., u_m)$.

MERGE(Q1,Q2) significa concatenar las dos colas uniendo el último elemento de *Q1* con el primero de *Q2*. El símbolo *Q* indica una cola concatenable y *Q(v)=()* indica que la cola asociada a *v* está vacía.

```
Algoritmo RIGHTSCAN(v0)
01. Q(S)=(v1); PUSH(v2,S);
02. Q(v2)=(); R_v2=(v1);
03. v=SIG(v2);
04. existeLLBumr=FALSO;
05. mientras(v ≠ v0){
06.   si(no GIRO_I(um-1,um,v)){
07.         si(existeLLBumr y GIRO_D(umTHREE,um,v)){
08.               llamar ONE(v,S);
09.         }y si no{ llamar TWO(v,S);}
10.   }y si no{
11.         si(GIRO_D(ANT(ANT((v)),um,v)){
12.               umTHREE=um;
13.               llamar THREE(v,S,existeLLBumr);
14.               saltar a la condición del mientras;
15.         }y si no{ llamar FOUR(v,S,v0);}
16.   }
17.   existeLLBumr=FALSO;
18. }
```

Por claridad en la exposición agrupamos en cuatro subrutinas los distintos casos planteados. *ONE* cubre los casos (i)(a) y (ii)(a); *TWO* los casos (i)(b) y (ii)(b); *THREE* el caso (iii)(a); y *FOUR* el caso (iii)(b).

En la línea 6 está la condición que separa los casos (i) de los (iii). Al preguntar si no es un giro izquierdo estamos considerando el caso colineal junto con el caso giro derecho.

En la línea 7 tenemos que determinar si existe asociado a u_m un lóbulo izquierdo $LLB(u_m,r)$ y si v se encuentra dentro de él. El caso que puede conducir a que estas condiciones se cumplan es como hemos visto el (iii)(a). En la subrutina *THREE*, que cubre el caso, es donde se determina la existencia de dicho lóbulo. Para comprobar si v está en su interior debemos considerar el lado que forma u_m con el vértice invisible que le precedía. Debemos ayudarnos de una variable booleana $existeLLBu_mr$ y de otra para almacenar el punto del vértice invisible eliminado, que llamamos u_mTHREE por ser la cima en la pila *S* en dicho caso, pues una vez terminado el análisis del caso *THREE* no queda rastro de ambos datos. Si v cae a la derecha del lado $\overline{u_mTHREE,u_m}$ entonces es interior al lóbulo y

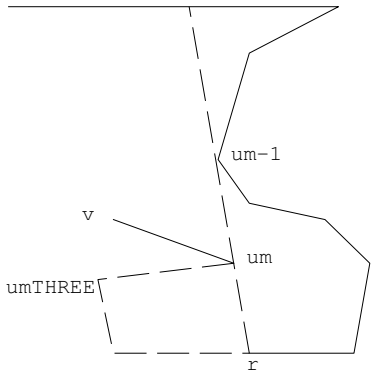

Figura 10. Caso en que *v* no se encuentra en el interior de *LLB(u_m,r)*.

si no exterior (Figura 10).

En la línea 11 tenemos que determinar si $\overline{u_{m-1},u_m}$ es un pseudo-lado y *v* se encuentra en *RLB(u_{m-1},u_m)*. Comprobamos en que semiplano cae de los definidos por el lado que forma u_m con el vértice anterior a éste, $\overline{ANT(ANT(v)),u_m}$. Si es en el de la derecha entonces estamos en el caso comentado.

```
Subrutina ONE(v,S)
01. borrar(u_m);
02. giroIv=FALSO;
03. mientras(no(giroIsigv=GIRO_I(u_m,r,SIG(v))) o giroIv o
      GIRO_I(v,SIG(v),r)){
04.   v=SIG(v); giroIv=giroIsigv;
05.   borrar(ANT(v));
06. }
07. Q(v)=Q(u_m);
08. ANT(v)=r; SIG(r)=v;
09. POP(S);
10. PUSH(v,S);
11. v=SIG(v);
```

En la línea 3 buscamos, a partir del vértice siguiente a *v*, el primer vértice que no esté dentro de *LLB(u_m,r)*. El problema se traduce en encontrar el lado $\overline{v,SIG(v)}$ que corta el segmento $\overline{u_m,r}$. Buscamos el corte preguntando si se dan tres giros entre las dos parejas de puntos. El cuarto giro como se puede ver en la Figura 11 es innecesario. Además, para tratar correctamente como ONE el caso (ii)(a) en que u_{m-1},u_m y *v* son colineales forzamos que *SIG(v)* no puede ser un punto del segmento $\overline{u_m,r}$. Así, comprobamos el corte permitiendo que *v* sea un punto de $\overline{u_m,r}$ pero *SIG(v)* no, es decir, que u_m,*r*,*v* no sea giro izquierdo y que u_m,*r*,*SIG(v)* sea un giro izquierdo.

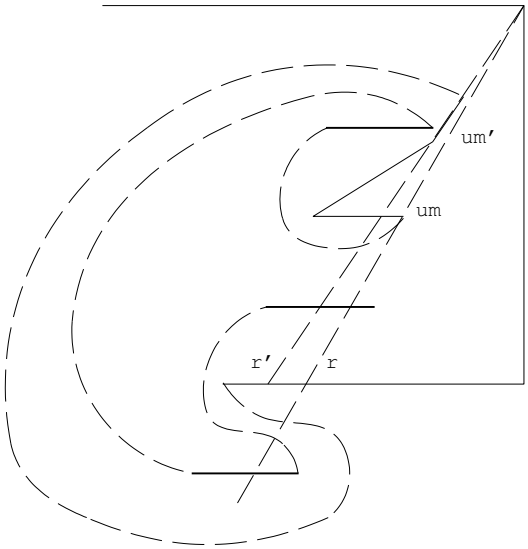

Figura 11. Condición de búsqueda del corte de *bd(P)* con el segmento $\overline{u_m, r}$.

El vértice *SIG(v)* encontrado es el siguiente vértice a analizar resultando ser un caso (iii)(a). Para que se reconozca como tal, según vimos, el vértice que le sigue debía caer en el semiplano de la derecha de los definidos por el lado $\overline{v, SIG(v)}$ por lo que no podemos borrar el vértice *v*. En lugar de calcular el corte de $\overline{u_m, r}$ y $\overline{v, SIG(v)}$ como punto que seguirá a *r* en *bd(P)*, simplificamos tomando *v* pues por ser invisible será borrado (Figura 12). La cola concatenable de tal punto es la misma que la del punto u_m (formada únicamente por *r*) y así se la asignamos a *v* con lo que el manejo de las colas concatenables para restablecer el estado en dicho punto y en *r* no se ve afectado por la simplificación.

En la línea 8 enlazamos convenientemente el polígono después de haber borrado vértices.

```
Subrutina TWO(v,S)
01. FIND_SPLIT(v,S); Q(S)=Q1; Q(v)=Q2;
02. R_v=CORTE(v_0,v_1,u_t,v);
```

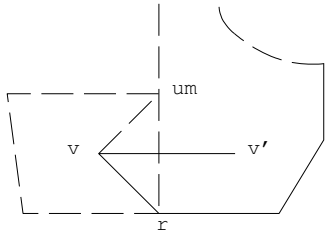

Figura 12. Lados en trazo continuo resultado de la subrutina ONE.

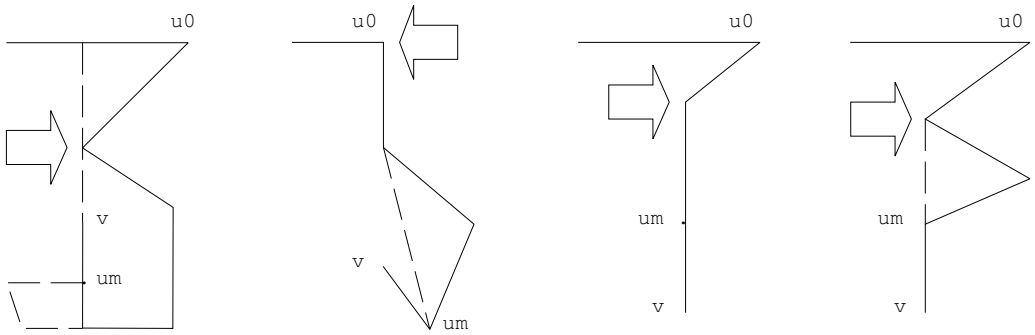

Figura 13. Flecha indica el vértice u_t.

```
03. PUSH(v,S);
04. v=SIG(v);
```

En la línea 1 hacemos una llamada a la función *FIND_SPLIT(v,S)* que a medida que busca el vértice u_t en $Q(S)=(u_0,..., u_t,..., u_m)$, visible desde v y más próximo a u_0 por el camino convexo, va separando las colas concatenables $Q1=(u_0,..., u_t)$ y $Q2=(u_{t+1},..., u_m)$. La visibilidad desde v para un vértice u_i en el camino convexo consiste en que el giro v,u_{i+1},u_i, $i=0, 1,..., m-1$, no sea un giro derecho, o bien, $i=m$. La condición de búsqueda que debe cumplir un vértice para ser u_t es, primero, que el vértice anterior en el camino convexo no sea visible desde v y, segundo, que él sí lo sea. Si todos fueran visibles tomaríamos u_0. En el caso en que v,u_{i+1},u_i son colineales el vértice u_i es visible, garantizando así que al añadir v al camino convexo no se incumpla la definición de camino convexo derecho por haber tres puntos consecutivos que no forman un giro a la izquierda. En la Figura 13 se pueden ver algunos casos en que v está alineado con dos puntos consecutivos del camino convexo y cómo se resuelve.

```
Subrutina THREE(v,S,giroDtuv)
01. u=ANT(v);
02. corteu_mvtu=VERDAD;
03. hacer{
04.   POP(S);
05.   MERGE(Q(S),Q(u));
06.   u=ANT(u);
07.   si(corteu_mvtu){u'=SIG(u); u"=u;}
08.   borrar(SIG(u));
09. }mientras no((giroDtuv=GIRO_D(t,u,v)) o
      ((corteu_mvtu=GIRO_I(u_m,v,t) y GIRO_D(u_m,v,u)) y ANT(u)=t));
10. si(giroDtuv){
11.   si(u ≠ u"){
12.         INSERTAR_V(u,s,v);
13.         s=CORTE(u',u",u,v);
```

```
14.         Q(s)=(); Rs=CORTE(v0,v1,u,v);
15.         PUSH(s,S);
16.    }y si no{ ANT(v)=u;};
17.    INSERTAR_V(ANT(v),r,v);
18.    FIND_SPLIT(v,S); Q(S)=Q1; Q(r)=Q2;
19.    r=CORTE(u',u",uk,v);
20.    Q(v)=(r); Rv=Rr=CORTE(v0,v1,uk,v);
21.    PUSH(v,S);
22.    v=SIG(v);
23. }y si no{
24.    w=CORTE(um,v,t,u);
25.    giroDv=FALSO;
26.    mientras(no(giroDsigv=GIRO_D(t,w,SIG(v))) o giroDv o
       GIRO_D(v,SIG(v),w)){
27.         v=SIG(v); giroDv=giroDsigv;
28.         borrar(ANT(v));
29.    }
30.    z=CORTE(v,SIG(v),t,w);
31.    Q(z)=Q(u); Rz=Ru;
32.    v=SIG(v);
33.    borrar(ANT(v));
34.    ANT(v)=u; SIG(u)=v;
35. }
```

En las líneas 4 y 5 restablecemos el estado en el vértice anterior en el polígono, que es la cima *u* en la pila *S* (estado en el vértice actual), borrando la cima de *S* y concatenando a *S* la cola de dicho vértice anterior. Dicho vértice anterior es invisible por lo que es borrado en la línea 8.

En la línea 7 comprobamos si en el estado anterior el lado $\overline{u_m,v}$ cortaba un pseudo-lado $\overline{t,u}$, en cuyo caso actualizamos el lado sobre el que va a caer el vértice r pues los lados en *LCH(u_m,u)* son invisibles.

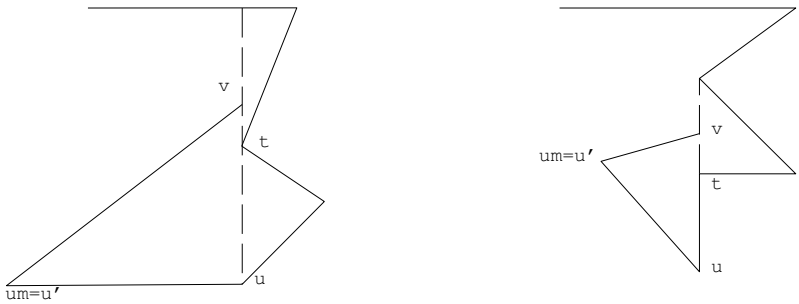

Figura 14. *t,u,v* colineales en el caso (iii)(a) y seguimos retrocediendo.

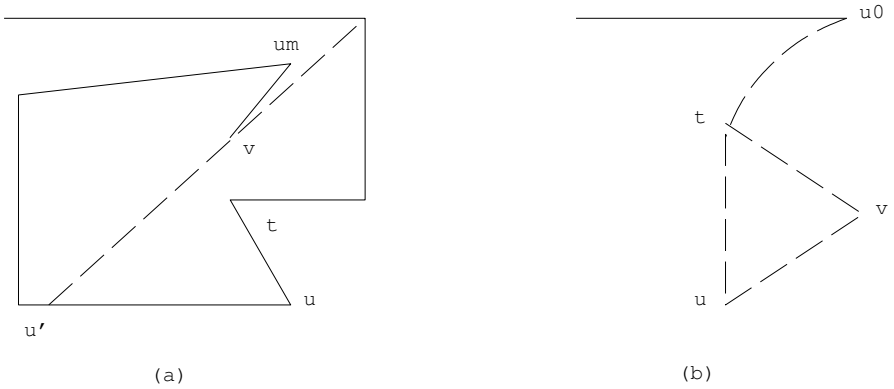

Figura 15. Condición de búsqueda del corte de *bd(P)* con el segmento $\overline{t,u}$.

En la línea 9 está la condición de fin del retroceso por bd(P). Primero comprobamos si *t,u,v* es un giro derecho, en cuyo caso *v* es visible y hemos finalizado el retroceso. Como se ve en la Figura 14 no basta que sean colineales para finalizar el retroceso, aunque sí lo sea para determinar que *v* es visible, pues hay más vértices invisibles que borrar. Después comprobamos si estamos en el caso en que entramos en un *RLB(t,u)* enteramente invisible, mirando si el lado $\overline{u_m,v}$ corta el segmento $\overline{t,u}$ y si $\overline{t,u}$ es actualmente un lado. Sabiendo que *t,u,v* no es un giro derecho el corte se produce si se da un giro izquierdo u_m,*v,t* y además un giro derecho u_m,*v,u*. Como se ve en la Figura 15a este último giro es necesario para asegurar que se da el corte. No es necesario comprobar el giro *t,u,*u_m, cuarto posible entre las dos parejas de puntos, porque habiendo comprobado los otros tres siempre es un giro derecho por la clasificación de casos que hemos hecho (Figura 15b, u_m caería si no en el triángulo *t,u,v* que se ve es un absurdo). Permitimos que *t,u,v* sean colineales pues se puede dar el caso, como se ve en la Figura 16.

En la línea 11 comprobamos si durante el retroceso hemos borrado ambos vértices del lado donde cae *r*, por ser invisibles. En cuyo caso creamos el vértice *s* en ese lado sobre la prolongación de $\overrightarrow{v,u}$, tal y como se puede ver en la Figura 17a. En la línea 12 insertamos convenientemente el vértice *s*. La parte del polígono que eliminamos es un lóbulo derecho

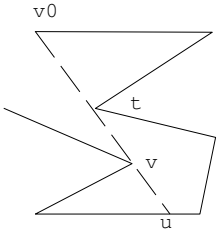

Figura 16. *t,u,v* colineales y entramos en un *RLB(t,u)* invisible.

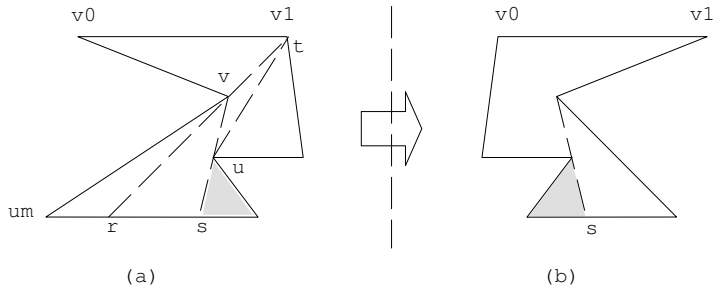

Figura 17. Eliminación de *RLB(u,s)*.

RLB(u,s) que resultaría igualmente invisible en el estudio simétrico que hacemos a continuación con el algoritmo LEFTSCAN, como se indica en la Figura 17b, pero que si no borráramos aquí conllevaría elevar la complejidad del algoritmo ya que podríamos llegar a retroceder varias veces por los mismos vértices. El punto *r* también lo creamos en ese lado $\overline{u', u''}$ sobre la prolongación del vértice buscado en la función FIND_SPLIT, u_k, cima de la pila *S*, y el vértice *v*. En la línea 17 insertamos el vértice *r*, teniendo en cuenta que no siempre se inserta el vértice *s*.

En la línea 26 buscamos el primer vértice *y*, a partir del vértice siguiente a *v*, de un lado $\overline{x, y}$ que corte el segmento $\overline{t, w}$ tal que dicho vértice no sea un punto de este segmento. Buscamos el corte preguntando si se dan tres giros entre las dos parejas de puntos. El cuarto giro como se puede ver en la Figura 18 es innecesario.

En las líneas 30 y 31 se ve que el vértice *u*, que es invisible, es sustituido en el polígono *P* por el vértice *z* simplemente cambiando las coordenadas.

En la línea 34 enlazamos convenientemente el polígono después de haber borrado

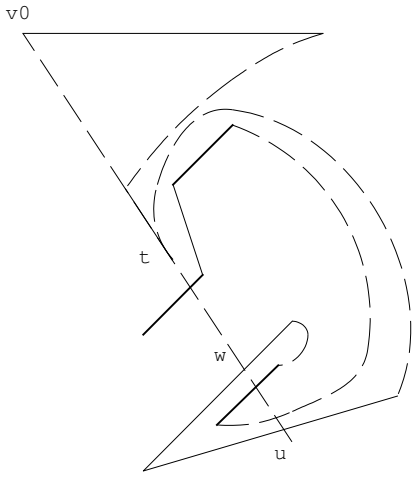

Figura 18. Condición de búsqueda del corte de *bd(P)* con el segmento $\overline{t, w}$.

vértices.

```
Subrutina FOUR(v,S,v0)
01. si(no GIRO_I(v0,um,v)){
02.   Q(v)=(); Rv=CORTE(v0,v1,v,um);
03. }y si no{
04.   um=ANT(v);
05.   mientras(GIRO_D(v0,um,SIG(v))){
06.         v=SIG(v);
07.         borrar(ANT(v));
08.   }
09.   ANT(v)=um; SIG(um)=v;
10.   t=CORTE(v,SIG(v),v0,um);
11.   Q(t)=Q(v); Rt=v0;
12. }
13. PUSH(v,S);
14. v=SIG(v);
```

En la línea 5 buscamos el primer vértice *w*, a partir del vértice siguiente a *v*, de un lado $\overline{u,w}$ que corte la línea $\overline{v_0,u_m}$ tal que dicho vértice no sea un punto de esta línea. Por estar el polígono *P* en forma estándar, podemos asegurar que el primer punto que caiga en el semiplano derecho de los que define la línea $\overline{v_0,u_m}$ es visible.

En la línea 9 enlazamos convenientemente el polígono después de haber borrado vértices.

En las líneas 10 y 11 se ve que el vértice *u*, que es invisible, es sustituido en el polígono *P* por el vértice *t* cambiando las coordenadas y la interceptación derecha.

2.2.3 Complejidad del algoritmo

Para calcular la complejidad, no es difícil ver que para cada vértice analizado gastamos como mucho *O(log n)* tiempo para actualizar el camino convexo (línea 1 de subrutina *TWO* o línea 18 de subrutina *THREE*), y que para cada vértice por el que retrocedemos gastamos como mucho *O(log n)* tiempo restableciendo el camino convexo (línea 5 de subrutina *THREE*). (Notar que la unión de dos colas concatenables puede hacerse en *O(log s)* tiempo, donde *s* es el tamaño total de las colas), y el restablecimiento se hace una vez por cada vértice. El vértice por el que se retrocede es borrado y no se vuelve a considerar. Así, el tiempo que implica el algoritmo RIGHTSCAN es *O(n log n)*.

Podemos entonces afirmar que el algoritmo descrito para calcular el Polígono de Visibilidad desde un Lado de un polígono P con n vértices tiene una complejidad $O(n \log n)$.

2.3. Polígono de Visibilidad de un Vértice

El algoritmo para calcular el *polígono de visibilidad desde un vértice* $VIS(v_0)$ de un polígono P en forma estándar consiste en aplicar el algoritmo SCANPTO según recorremos los vértices de P en sentido *horario*. Durante la aplicación del algoritmo se descarta la parte de $bd(P)$ que no es visible, borrándose algunos vértices y añadiéndose otros nuevos. Obtenemos *así* P' que es el polígono de visibilidad buscado. El algoritmo SCANPTO es un caso particular del algoritmo RIGHTSCAN en el que el ancla se ha reducido a un lado de un sólo punto, lo que implica una serie de simplificaciones que vemos a continuación.

2.3.1 Algoritmo SCANPTO

A continuación describiremos el algoritmo SCANPTO.

La simplificación principal del algoritmo es que el camino convexo derecho $RCVX(v_1,v)$ que manteníamos para cada vértice v en el algoritmo RIGHTSCAN se va a componer únicamente de (u_0,u_1), donde $u_0=v_0$ y u_1 es el vértice anterior en $bd(P)$. Por tanto no cabe hablar de guardar los vértices del camino en una pila S, ni de búsquedas sobre la misma. Las interceptaciones derechas sabemos que coinciden sobre v_0. Seguimos hablando del *estado en* v, compuesto de los dos vértices indicados. Seguidamente describimos el proceso para obtener el siguiente estado desde el actual. El proceso se repite hasta que el estado en v_0 es alcanzado. El estado inicial es $S=(v_0,v_1)$ en v_2. Tenemos los siguientes tres casos dependiendo de la posición del próximo vértice v a ser analizado.

(i) u_0,u_1,v es un giro derecho. En este caso distinguiremos dos subcasos.

> (a) (Figura 19a). Asociado con u_1 hay un lóbulo izquierdo $LLB(u_1,r)$ para algún r sobre $LCH(u_1,v_0)$ y v se encuentra dentro de $LLB(u_1,r)$. (La existencia de r se puede producir a partir del caso (iii)(a).) En este caso u_1 es invisible así como los vértices de P que siguen a v y se encuentran dentro de $LLB(u_1,r)$. Borramos esos vértices hasta encontrar un vértice v' que no está en $LLB(u_1,r)$. El estado en v' es $S'=(u_0,r)$ y tenemos el caso (iii)(a).

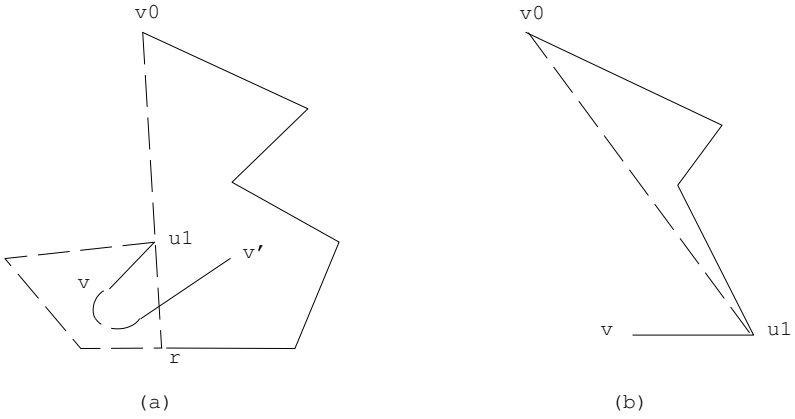

Figura 19. u_0,u_1,v es un giro derecho.

(b) (Figura 19b). En este caso el nuevo estado en el siguiente vértice a v en $RCH(v_1,v_0)$, es $S'=(u_0,v)$, y $\overline{u_0,v}$ es un pseudo-lado, es decir, u_0 y v definen un lóbulo derecho $RLB(u_0,v)$.

(ii) (Figura 20) u_0,u_1 y v son colineales. Este caso se trata como si u_0,u_1,v fuera un giro derecho. El nuevo estado es (u_0,v). (El caso en el que v se encuentra sobre $\overline{u_1,r}$ de $LLB(u_1,r)$ es considerado como si v estuviera dentro de $LLB(u_1,r)$ también.)

(iii) u_0,u_1,v es un giro izquierdo. En este caso distinguimos dos subcasos.

(a) (Figura 21a) $\overline{u_0,u_1}$ es un pseudo-lado y v se encuentra en $RLB(u_0,u_1)$. u_1 es invisible. Así, retrocedemos por los vértices u', de estado $S'=(u_0,u)$, de $bd(P)$ empezando con u_1 mientras u_0,u,v no sea un giro derecho, excepto en el caso, mostrado en la Figura 21b, en el cual $\overline{u,u'}$ es un pseudo-lado especial; es decir, la región definida por $RLB(u,u')$ es enteramente invisible (ver (iii)(b)). Los vértices por los que retrocedemos son invisibles y son borrados. Si no, sea r la intersección de $\overline{u',u}$ y $\overline{u_0,v}$. Entonces el siguiente estado es S"=(u_0,v). (Aquí los puntos r,v y

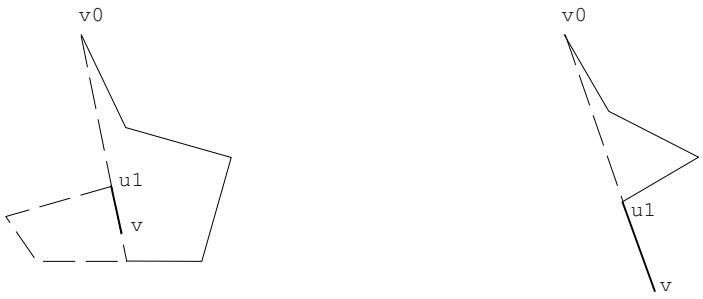

Figura 20. u_0,u_1,v son colineales.

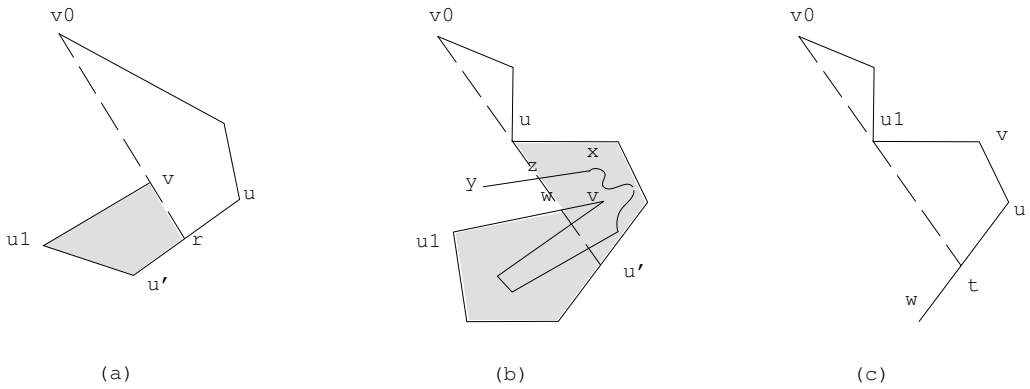

Figura 21. u_0,u_1,v es un giro izquierdo.

u_0 son colineales y hemos tratado v como si estuviera a la derecha de $\overline{u_0,r}$.) Notar que tenemos un lóbulo izquierdo *LLB(v,r)* asociado con el segundo vértice de *S''* (pudiendo ser origen de un caso (i)(a)). Los vértices u_1,..., u' en *LLB(v,r)* son invisibles desde el ancla y son borrados. Para el caso donde $\overline{u,u'}$ es un pseudo-lado especial, hacemos lo siguiente. Encontramos la intersección w de $\overline{u,u'}$ y $\overline{u_1,v}$ e ignoramos todos los vértices que siguen a v hasta encontrar un lado $\overline{x,y}$ que corta $\overline{u,w}$. Sea z la intersección de $\overline{x,y}$ y $\overline{u,w}$. De este modo tenemos un estado (u_0,z) en y y tenemos el caso (i)(b).

(b) (Figura 21c). En este caso borramos v y buscamos un lado $\overline{u,w}$ tal que v_0,u_1,u es un giro izquierdo pero v_0,u_1,w no lo es; es decir, el lado $\overline{u,w}$ corta $\overline{v_0,u_1}$. Sea t la intersección del lado $\overline{u,w}$ y $\overline{v_0,u_1}$. Tenemos un nuevo estado $S'=(u_0,t)$ en w y tenemos el caso (i)(b). Notar que en este caso $\overline{u_1,t}$ es un pseudo-lado especial en el que la región definida por $RLB(u_1,t)$ es enteramente invisible.

2.3.2 Pseudo-código algoritmo SCANPTO

A continuación hacemos una descripción detallada del anterior algoritmo. El pseudo-código utilizado es el ya explicado en el algoritmo RIGHTSCAN. Presentamos después los bloques de código con las líneas numeradas correspondientes al algoritmo y subrutinas utilizadas seguido cada uno de sus comentarios particulares.

```
Algoritmo SCANPTO(v0)
01. v=SIG(v1);
02. existeLLBu1r=FALSO;
03. mientras(v ≠ v0){
04.   si(no GIRO_I(v0,ANT(v),v)){
```

```
05.         si(existeLLBu1r y GIRO_D(u1THREE,ANT(v),v)){
06.               llamar ONE(v);
07.         }y si no{ v=SIG(v);}
08.   }y si no{
09.         si(GIRO_D(ANT(ANT((v)),ANT(v),v)){
10.               u1THREE=ANT(v);
11.               llamar THREE(v,existeLLBu1r);
12.               saltar a la condición del mientras;
13.         }y si no{ llamar FOUR(v,v0);}
14.   }
15.   existeLLBu1r=FALSO;
16. }
```

Por claridad en la exposición agrupamos en tres subrutinas los distintos casos planteados. *ONE* cubre los casos (i)(a) y (ii)(a); *THREE* el caso (iii)(a); y *FOUR* el caso (iii)(b).

En la línea 4 está la condición que separa los casos (i) de los (iii). Al preguntar si no es un giro izquierdo estamos considerando el caso colineal junto con el caso giro derecho.

En la línea 5 tenemos que determinar si existe asociado a u_1 un lóbulo izquierdo $LLB(u_1,r)$ y si v se encuentra dentro de él. El caso que puede conducir a que estas condiciones se cumplan es como hemos visto el (iii)(a). En la subrutina *THREE*, que cubre el caso, es donde se determina la existencia de dicho lóbulo. Para comprobar si v está en su interior debemos considerar el lado que forma u_1 con el vértice invisible que le precedía. Debemos ayudarnos de una variable booleana $existeLLBu_1r$ y de otra para almacenar el punto del vértice invisible eliminado, que llamamos u_1THREE por ser el segundo vértice en S en dicho caso, pues una vez terminado el análisis del caso *THREE* no queda rastro de ambos datos. Si v cae a la derecha de este lado, $\overline{u_1THREE,u_1}$, entonces es interior al lóbulo.

En la línea 9 tenemos que determinar si $\overline{u_0,u_1}$ es un pseudo-lado y v se encuentra en $RLB(u_0,u_1)$. Comprobamos en que semiplano cae de los definidos por el lado que forma u_1 con el vértice anterior a éste, $\overline{ANT(ANT(v)),u_1}$. Si es en el de la derecha entonces estamos en el caso comentado.

```
Subrutina ONE(v)
01. borrar(u1);
02. giroIv=FALSO;
03. mientras(no(giroIsigv=GIRO_I(u1,r,SIG(v))) o giroIv o
```

```
      GIRO_I(v,SIG(v),r)){
04.   v=SIG(v); giroIv=giroIsigv;
05.   borrar(ANT(v));
06. }
07. ANT(v)=r; SIG(r)=v;
08. v=SIG(v);
```

En la línea 3 buscamos, a partir del vértice siguiente a *v*, el primer vértice que no esté dentro de *LLB(u_1,r)*. El problema se traduce en encontrar el lado $\overline{v,SIG(v)}$ que corta el segmento $\overline{u_1,r}$. Buscamos el corte preguntando si se dan tres giros entre las dos parejas de puntos. El cuarto giro es innecesario. Además, para tratar correctamente como ONE el caso (ii)(a) en que u_0,u_1 y *v* son colineales forzamos que *SIG(v)* no puede ser un punto del segmento $\overline{u_1,r}$. Así, comprobamos el corte permitiendo que *v* sea un punto de $\overline{u_1,r}$ pero *SIG(v)* no, es decir, que u_1,r,v no sea giro izquierdo y que u_1,r,*SIG(v)* sea un giro izquierdo.

El vértice *SIG(v)* encontrado es el siguiente vértice a analizar resultando ser un caso (iii)(a). Para que se reconozca como tal, según vimos, el vértice que le sigue debía caer en el semiplano de la derecha de los definidos por el lado $\overline{v,SIG(v)}$ por lo que no podemos borrar el vértice *v*. En lugar de calcular el corte de $\overline{u_1,r}$ y $\overline{v,SIG(v)}$ como punto que seguirá a *r* en *bd(P)*, simplificamos tomando *v* pues por ser invisible será borrado.

En la línea 7 enlazamos convenientemente el polígono después de haber borrado vértices.

```
Subrutina THREE(v,giroDtuv)
01. u=ANT(v);
02. corteu_mvtu=VERDAD;
03. hacer{
04.   u=ANT(u);
05.   u'=SIG(u);
06.   borrar(SIG(u));
07. }mientras no((giroDtuv=GIRO_D(v_0,u,v)) o GIRO_I(u_1,v,u));
08. si(giroDtuv){
09.   INSERTAR_V(u,r,v);
10.   r=CORTE(u',u,v_0,v);
11. }y si no{
12.   w=CORTE(u_1,v,u,v_0);
13.   giroDv=FALSO;
14.   mientras(no(giroDsigv=GIRO_D(u,w,SIG(v))) o giroDv o
      GIRO_D(v,SIG(v),w)){
```

```
15.         v=SIG(v); giroDv=giroDsigv;
16.         borrar(ANT(v));
17.    }
18.    z=CORTE(v,SIG(v),u,w);
19.    ANT(v)=u; SIG(u)=v;
20. }
21. v=SIG(v);
```

En la línea 7 está la condición de fin del retroceso por bd(P). Primero comprobamos si v_0,u,v es un giro derecho, en cuyo caso v es visible y hemos finalizado el retroceso. No basta que sean colineales para finalizar el retroceso, aunque sí lo sea para determinar que v es visible, pues hay más vértices invisibles que borrar. Después comprobamos si estamos en el caso en que entramos en un *RLB(t,u)* enteramente invisible, mirando si el lado $\overline{u_1,v}$ corta el segmento $\overline{u,u'}$. Sabiendo que v_0,u,v no es un giro derecho el corte se produce si se da un giro izquierdo u_1,v,u. Permitimos que t,u,v sean colineales pues se puede dar el caso.

En la línea 14 buscamos el primer vértice y, a partir del vértice siguiente a v, de un lado $\overline{x,y}$ que corte el segmento $\overline{u,w}$ tal que dicho vértice no sea un punto de este segmento. Buscamos el corte preguntando si se dan tres giros entre las dos parejas de puntos. El cuarto giro es innecesario.

En la línea 18 el último vértice invisible, es sustituido en el polígono P por el vértice z simplemente cambiando las coordenadas.

En la línea 19 enlazamos convenientemente el polígono después de haber borrado vértices.

```
Subrutina FOUR(v,v0)
01. um=ANT(v);
02. mientras no(GIRO_D(v0,u1,SIG(v))){
03.   v=SIG(v);
04.   borrar(ANT(v));
05. }
06. ANT(v)=um; SIG(um)=v;
07. t=CORTE(v,SIG(v),v0,u1);
08. v=SIG(v);
```

En la línea 2 buscamos el primer vértice w, a partir del vértice siguiente a v, de un lado $\overline{u,w}$ que corte la línea $\overline{v_0,u_1}$ tal que dicho vértice no sea un punto de esta línea. Por estar el polígono P en forma estándar, podemos asegurar que el primer punto que caiga en el

semiplano derecho de los que define la línea $\overline{v_0, u_1}$ es visible.

En la línea 6 enlazamos convenientemente el polígono después de haber borrado vértices.

En la línea 7 el vértice *u*, que es invisible, es sustituido en el polígono *P* por el vértice *t* simplemente cambiando las coordenadas.

2.3.3 Complejidad del algoritmo

Para calcular la complejidad, no es difícil ver que para cada vértice analizado gastamos como mucho *O(1)* tiempo para actualizar el estado (la línea de subrutina *TWO* o línea 10 de subrutina *THREE*), y que para cada vértice por el que retrocedemos gastamos como mucho *O(1)* tiempo restableciendo el estado (línea 5 de subrutina *THREE*), y el restablecimiento se hace una vez por cada vértice. El vértice por el que se retrocede es borrado y no se vuelve a considerar. Así, el tiempo que implica el algoritmo SCANPTO es *O(n)*.

Podemos entonces afirmar que el algoritmo descrito para calcular el Polígono de Visibilidad desde un Vértice de un polígono *P* con n vértices tiene una complejidad *O(n)*.

2.4. Algoritmo Forma Estándar

Un polígono orientado *P*, de vértices $v_0, v_1, \ldots, v_n=v_0$, donde $\overline{v_0, v_1}$ es el ancla, está en forma estándar si ninguno de los vértices, v_i, $i=2, 3, \ldots, n-1$, está por encima de la línea que contiene $\overline{v_0, v_1}$.

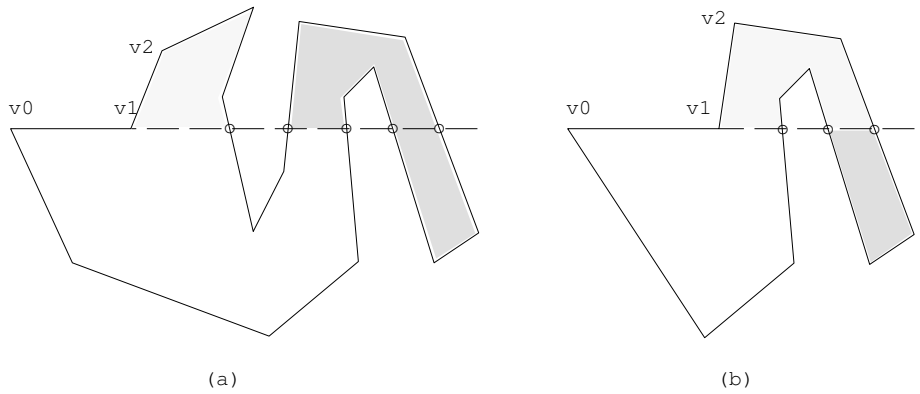

Figura 22. Los trozos rayados son invisibles y en los punteados calculamos el polígono de visibilidad desde un vértice.

El algoritmo para calcular la forma estándar consiste en dibujar una línea a través de $\overline{v_0, v_1}$ cortando P en trozos que quedan a ambos lados de la misma y eliminar los quedan en el lado del ancla que da al exterior de P. Para identificar los trozos calculamos los cortes del contorno del polígono *bd(P)* con la línea que contiene $\overline{v_0, v_1}$ insertando nuevos vértices en P cuando dichos cortes pertenezcan a un punto interior de un lado. Después ordenamos los vértices de P que caen sobre ésta línea, los que están antes que v_0 entre ellos y según la distancia a v_0, y los que están a partir de v_1 entre ellos y según la distancia a v_1. Entonces, si v_0,v_1,v_2 es un giro izquierdo (Figura 22a) borramos todos los vértices desde v_2 hasta el vértice anterior al que está sobre la línea y le sigue según el orden definido. Si no, seguimos el orden en *bd(P)* hasta el siguiente vértice que caiga sobre la línea. Vamos realizando ambas operaciones, eliminar trozo y seguir por *bd(P)*, alternativamente hasta llegar al vértice v_0.

En los casos en que el primer trozo eliminado de P contenga el vértice v_1 o el último trozo eliminado contenga el vértice v_0, calculamos el polígono de visibilidad del trozo desde el vértice v_1 o v_0 correspondiente. Para pasar estos trozos del polígono P a forma estándar aprovechamos que tenemos calculados los cortes sobre la línea que contiene el ancla. En este caso sobre dichos polígonos debemos eliminar los trozos que caen en el lado del ancla que da al interior de P (Figura 22b).

El algoritmo no reviste mayor complicación si los vértices v_i de P no caen sobre la línea que contiene el ancla. Si no, debemos considerar donde caen los vértices anterior v_{i-1} y siguiente v_{i+1} al que está sobre dicha línea para determinar si en el vértice puede empezar o terminar un trozo que haya que eliminar. A continuación, hacemos un análisis exhaustivo de todos los casos posibles y exponemos las conclusiones.

Si cambiamos de lado del ancla al pasar de v_{i-1} a v_{i+1} simplemente saltamos el paso de introducir un nuevo vértice en P y debemos considerar el corte.

Si ambos vértices, v_{i-1} y v_{i+1}, caen en el mismo lado del ancla tenemos las siguientes ocho posibilidades que indicamos en la Figura 23. En las figuras indicamos con un círculo negro sobre el vértice que debemos considerar un corte en la línea. La flecha paralela al ancla indica que hemos empezado a borrar vértices, de un trozo a eliminar, desde el vértice donde nace la flecha hasta el primero que se encuentre sobre la línea en el sentido que señala. La flecha vertical señala, para estos casos, por donde tendrá que cortar la línea el contorno del polígono *bd(P)* en algún momento.

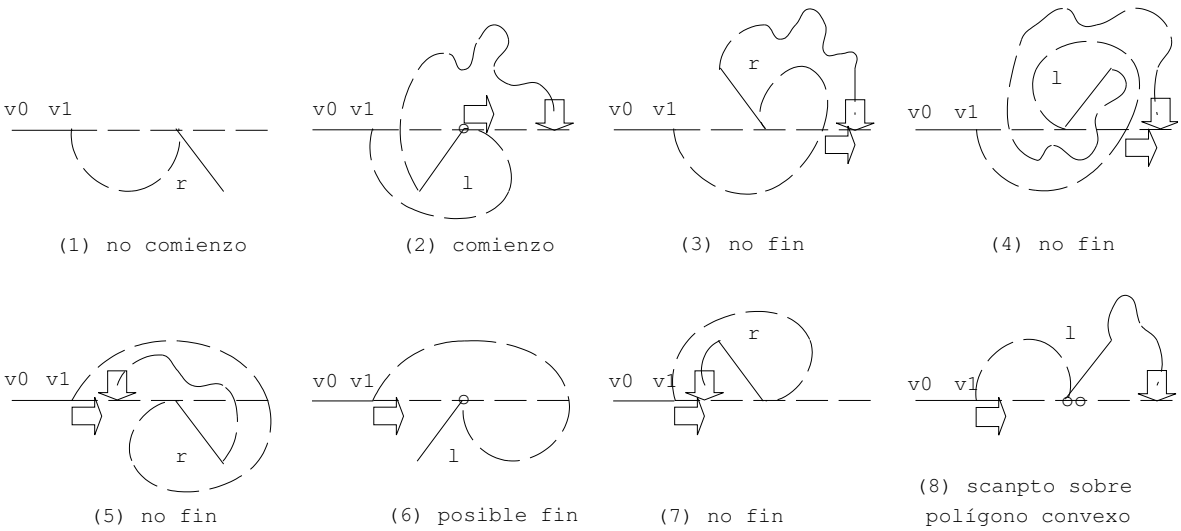

Figura 23. v_i cae en la línea que contiene el ancla y v_{i-1}, v_{i+1} caen en el mismo semiplano de los que define el ancla.

En el primer caso vemos que el vértice no supone un comienzo de trozo a eliminar, por lo que debemos no considerar un corte. En el segundo vemos que supone un comienzo de trozo a eliminar, y debemos considerar un corte. En el tercero, cuarto, quinto y séptimo, cuando llegamos al vértice sobre la línea que contiene el ancla v_i, ya venimos eliminando vértices desde un corte anterior y vemos que en ninguno de los casos el vértice supone un fin de trozo a eliminar, por lo que no es necesario considerar un corte. En el sexto caso también venimos buscando un fin de trozo desde un corte anterior y vemos que sí se puede producir ese fin sobre el vértice, por lo que debemos considerar un corte. En el caso octavo ocurre otra vez lo mismo y no supone un fin del trozo, pero vemos que para el caso de que el primer subtrozo contenga el vértice v_1 o el segundo subtrozo contenga el vértice v_0, el vértice v_i supone el fin del pedazo sobre el que calcular el polígono de visibilidad desde el vértice correspondiente por lo que consideramos dos cortes indicando con ello que suponemos un fin de trozo a eliminar e inmediatamente otro comienzo de trozo a eliminar.

Si uno de los dos vértices, v_{i-1} o v_{i+1}, cae en la línea que contiene el ancla y el otro no, tenemos los dieciséis casos de la Figura 24. Los ocho primeros están creados a partir de los del párrafo anterior añadiendo un segundo vértice a continuación del que está sobre el ancla. Son totalmente análogos y no los comentaremos más. Los siguientes ocho casos en la figura están creados a partir de estos ocho primeros pero cambiando de lado el vértice siguiente al último de los que están sobre la línea que contiene el ancla.

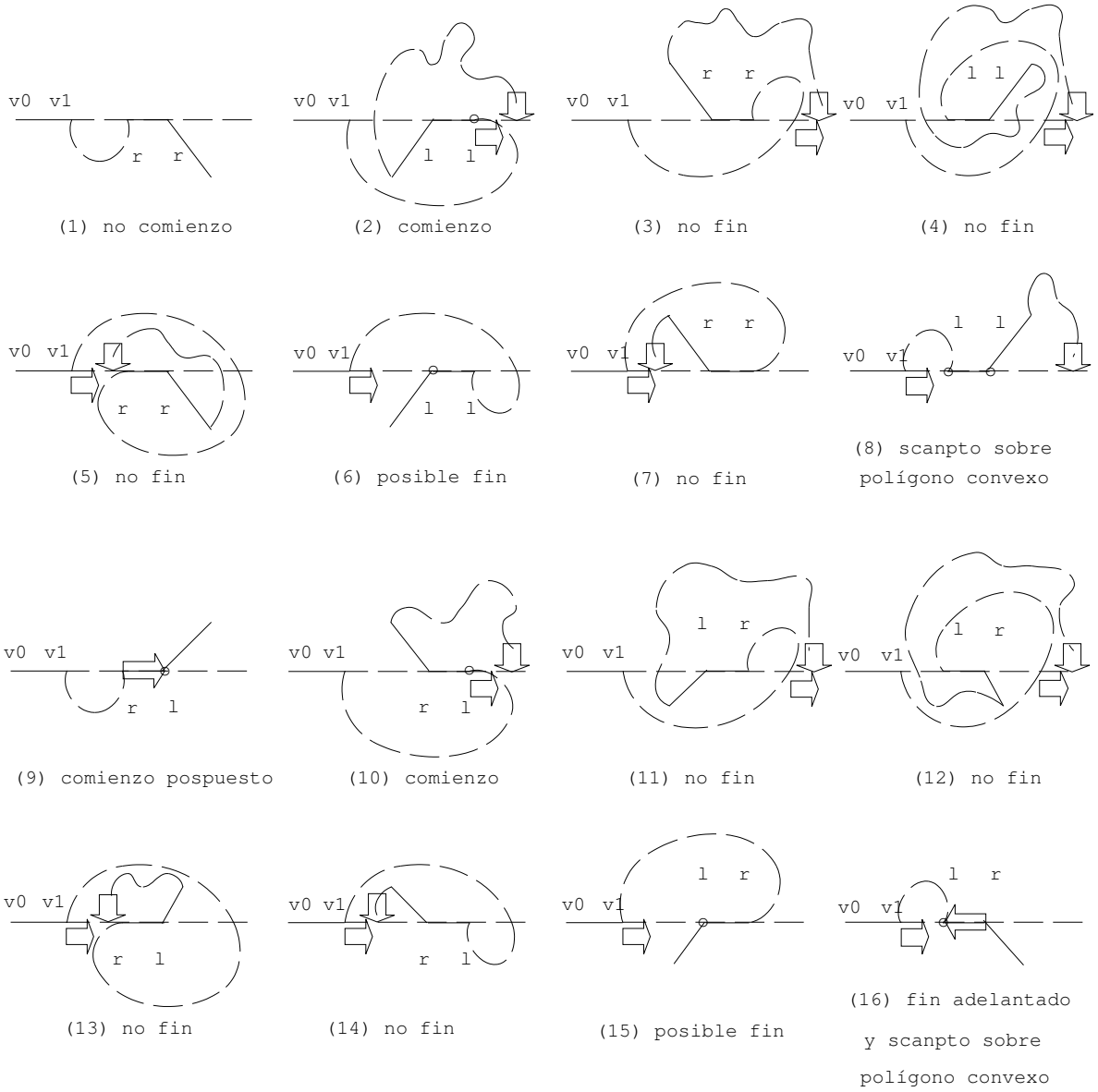

Figura 24. v_i cae en la línea que contiene el ancla y v_{i-1} o v_{i+1} también.

En el noveno caso vemos que el primer vértice sobre la línea supone un comienzo de trozo a eliminar debiendo considerar un corte, pero lo consideramos sobre el último. En el décimo caso el primer vértice supone un comienzo de trozo, por lo que debemos considerar un corte. En los casos decimoprimero a decimocuarto venimos eliminando vértices desde un corte anterior y vemos que en ninguno de los casos el vértice supone un fin de trozo a eliminar, por lo que no es necesario considerar cortes. En el decimoquinto caso ocurre lo mismo pero esta vez es posible que el último vértice sea un fin de trozo, por lo que debemos considerar un corte. En el decimosexto caso vemos que el último vértice supone un fin de trozo por lo que debemos considerar un corte, pero lo consideramos

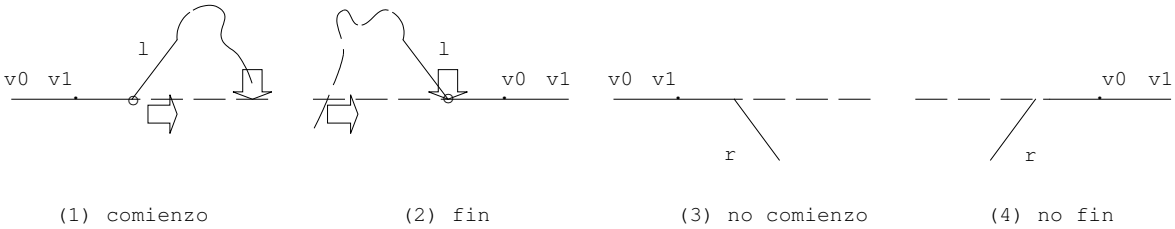

Figura 25. v_i cae en la línea que contiene el ancla y es el siguiente a v_1 o el anterior a v_0.

sobre el primer vértice ya que éste puede suponer a la vez un fin de trozo sobre el que calcular un polígono de visibilidad desde el vértice v_1.

En el caso particular de que el vértice que cae sobre la línea que contiene el ancla, v_i, sea el siguiente a v_1 o el anterior a v_0 tenemos las cuatro posibilidades de la Figura 25. En el primer caso vemos que supone un comienzo de trozo a eliminar, por lo que debemos considerar un corte. En el segundo caso vemos que supone un fin de trozo, por lo que debemos considerar un corte. En los últimos dos casos el vértice no supone comienzo o fin de trozo, por lo que debemos no considerar un corte.

Cuando tenemos más de dos vértices consecutivos sobre el ancla los vértices de los extremos son los únicos sobre los que pueden empezar o finalizar un trozo a eliminar por lo que debemos no considerar cortes en los vértices interiores.

Hasta aquí hemos repasado todos los casos posibles cuando un vértice cae sobre la línea que contiene el ancla. Hemos visto que en algunos de ellos debemos considerar un corte, en otros debemos no considerarlo y en un tercer grupo es indiferente. Para simplificar el tratamiento agrupamos los casos de forma que consideramos un corte allí donde comienza o finaliza un trozo y aseguramos que los cortes considerados no suponen un comienzo o fin de trozo allí donde lo pueda buscar el algoritmo. Resultan entonces los siguientes tres grupos.

Si cambiamos de lado del ancla al pasar del vértice anterior al siguiente, v_{i-1} a v_{i+1}, consideramos un corte. Si uno de los dos vértices cae sobre la línea que contiene el ancla y entre los tres, v_{i-1},v_i,v_{i+1}, forman un giro izquierdo consideramos un corte. Si ambos caen en el mismo lado del ancla y entre los tres, v_{i-1},v_i,v_{i+1}, forman un giro izquierdo consideramos el corte dos veces seguidas siguiendo el sentido en $bd(P)$ para definir el anterior y siguiente en la cadena ordenada de cortes. Lo hacemos así para que cuando se deba considerar un solo corte el algoritmo siempre encuentre el segundo de ellos. No consideramos el grupo de cortes dobles sólo cuando v_{i-1} e v_{i+1} están en el lado del ancla

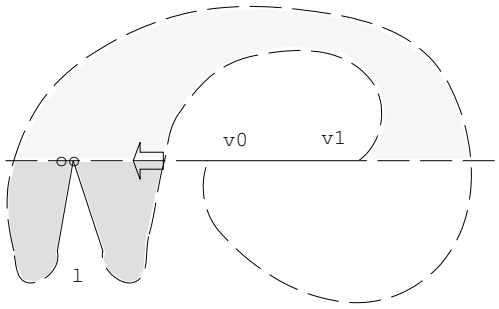

Figura 26. Corte doble con v_{i-1} y v_{i+1} en el lado del ancla que da al interior de P.

que da al exterior de P, porque reutilizamos los cortes para pasar a forma estándar los trozos sobre los que calcular los polígonos de visibilidad en v_0 y v_1, y según vimos eliminamos trozos del lado del ancla que da al interior de P, como se ve en la Figura 26.

2.5. Algoritmo Orientación

Para determinar la orientación de los vértices de un polígono P, tomamos un vértice de abscisa mínima v_i. Aseguramos que el vértice considerado no está alineado con los vértices anterior y siguiente comprobando que la abscisa del vértice que le sigue v_{i+1} es distinta (Figura 27). Sabiendo que el interior cae a la derecha del vértice con abscisa mínima calculamos el signo del determinante que describe el giro v_{i-1},v_i,v_{i+1}. Si es menor que cero, es un giro derecho y el polígono está orientado. En caso contrario el estudio es simétrico y basta considerar los giros derechos como izquierdos y viceversa.

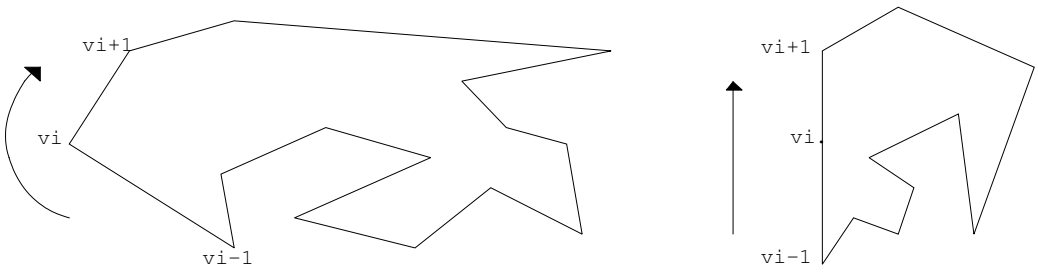

Figura 27. Determinación del vértice de abscisa mínima v_i no alineada.

3. ESTRUCTURAS DE DATOS

En este capítulo repasamos todas las estructuras de datos utilizadas, exponiendo su definición y comentando las operaciones que se realizan sobre ellas, la información que contienen y para qué se utilizan.

3.1. Punto

Es una de las estructuras de datos básicas en Geometría Computacional. Lo utilizamos para almacenar las coordenadas, horizontales y verticales, de los vértices.

Las operaciones definidas son: cálculo de la distancia entre dos puntos, determinación de si dos puntos son iguales, cálculo del punto de corte entre las rectas definidas por dos pares de puntos, cálculo del signo del determinante que define el giro entre tres puntos, determinación de si tres puntos forman un giro izquierdo o un giro derecho.

Definimos la estructura de datos punto a continuación.

```
typedef struct{float x, y;}punto;
```

3.2. Polígono (simplemente enlazado)

Es otra de las estructuras de datos que subyace en muchos problemas que se plantean en Geometría Computacional. Para poder almacenar un polígono debemos guardar el orden

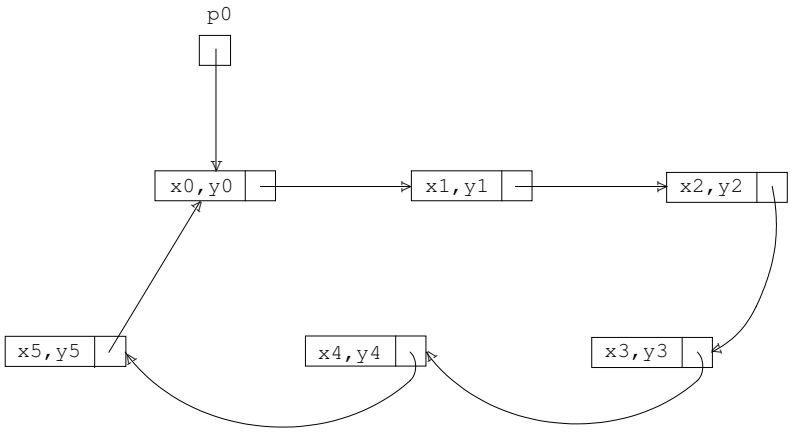

Figura 28. Polígono (simplemente enlazado).

```
typedef struct vsimple *pvsimple;
struct vsimple{  punto pto;
                 pvsimple sig;};
```

en que se suceden los vértices. Nos apoyamos en la estructura de datos vértice simple que permite almacenar el vértice siguiente en el polígono y que definimos debajo de la Figura 28, en la que representamos un polígono simplemente enlazado construido a partir de vértices simples.

3.3. Cola Concatenable

Es una estructura de datos definida sobre conjuntos de elementos entre los que tenemos definido un orden lineal ≤, que puede procesar las operaciones de *insertar, borrar, buscar, partir* y *concatenar*, que comentamos a continuación. *Insertar(a,S)*, inserta un nuevo elemento *a* en un conjunto *S*. *Borrar(a,S)*, borra un elemento *a* de un conjunto *S*. *Buscar(S)*, busca un elemento que cumpla una determinada condición en un conjunto *S*. *Partir(a,S)*, parte un conjunto *S* en dos conjuntos S_1 y S_2 tal que $S_1=\{b|b\leq a \text{ y } b \in S\}$ y $S_2=\{b|b>a \text{ y } b \in S\}$. *Concatenar*($S_1$,$S_2$), toma como entradas dos conjuntos S_1 y S_2 tales que cada elemento de S_1 es menor que cada elemento de S_2, y produce como salida el conjunto concatenado S_1S_2.

Utilizamos las colas concatenables para almacenar los caminos convexos derechos de los vértices en el algoritmo RIGHTSCAN que usamos para calcular el Polígono de Visibilidad desde un Lado. El comportamiento de un camino convexo en cuanto a las operaciones de inserción y borrado es el de una pila. Es decir, que al insertar un vértice en el camino convexo siempre lo añadimos al final del mismo y al borrar un vértice siempre borramos el último insertado. En adelante, llamaremos a estas dos operaciones *push* y *pop*, respectivamente. El último elemento insertado en una pila es la *cima*.

Hemos utilizado dos estructuras de datos distintas para implementar las colas concatenables atendiendo a dos necesidades distintas. Por un lado necesitamos una estructura de datos que pueda procesar *n* instrucciones en un tiempo $O(n \log n)$ para conseguir una implementación óptima del algoritmo. Por otro lado necesitamos poder recorrer los elementos de un conjunto *S* secuencialmente para poder dibujar en pantalla cada lado del camino convexo que representa. Esto será necesario cuando ejecutemos paso a paso los algoritmos para poder ver los estados intermedios. La estructura que utilizamos en el primer caso gasta un tiempo $O(\log n)$ para cada una de las operaciones cumpliendo el objetivo. La estructura que utilizamos en el segundo caso gasta un tiempo $O(n)$ para las operaciones *buscar* y *partir*, y un tiempo $O(1)$ para el resto. Estas estructuras de datos son *árboles-pila binarios* y *listas* respectivamente, que analizamos a continuación.

3.3.1 Árboles-pila binarios

Un árbol-pila binario es un *árbol balanceado*, queriendo decir con esto que la altura es aproximadamente igual al logaritmo (en base dos) del número de vértices en el árbol. Un árbol balanceado es fácil de construir inicialmente. El problema es prevenir que el árbol deje de estar balanceado tras ejecutar una serie de instrucciones de inserción y borrado.

Definición. Un *árbol-pila binario* es un árbol en el cual cada vértice que no es una hoja tiene dos hijos excepto el vértice más a la derecha del árbol que puede tener o dos hijos o uno izquierdo, y en el que los caminos de la raíz a las hojas de todo subárbol izquierdo son de igual longitud. Notar que un árbol de una sola hoja es un árbol-pila binario. En la Figura 29a se representa un árbol-pila binario en el que todos los vértices que no son hojas tienen dos hijos excepto el vértice 18 que es el vértice más a la derecha y tiene un hijo izquierdo. Además los caminos de la raíz a las hojas del subárbol izquierdo con raíz el vértice 8 tienen todos longitud 4, los del vértice 4 tienen longitud 3, los de los vértices 2 y 10 tienen longitud 2 y los de los vértices 1, 5, 9, 13 y 17 tienen longitud 1.

Los vértices de un árbol-pila binario guardan un orden inherente entre ellos expresado por la posición que ocupan en el árbol. Cada posición representa un número de puesto que va de *1* a *n*, siendo *n* el número de vértices del árbol. De ésta afirmación se deduce que sólo existe un único árbol-pila binario posible para cada número *n* de vértices. Partiendo desde la raíz del árbol en su subárbol izquierdo están los vértices con un número de puesto menor que el de la raíz, y en el subárbol derecho los que tienen un puesto mayor. Siempre que vayamos por una rama izquierda vamos a encontrar vértices con números de puesto menores y si vamos por una rama derecha vamos a encontrar vértices con números de puesto mayores. Si representamos el número de puesto en binario, un vértice y los vértices

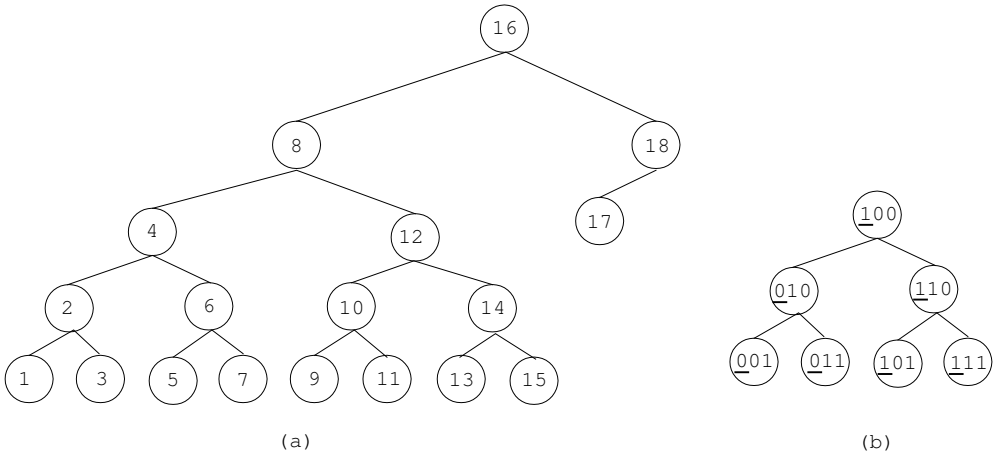

Figura 29. Número de puesto en árbol-pila binario.

de su subárbol derecho tienen en común un uno donde los vértices del subárbol izquierdo tienen un cero. Esta información de ceros y unos nos proporciona el número de puesto de cada vértice según navegamos por el árbol, o el camino que debemos seguir para llegar a un vértice a partir de su puesto. En el subárbol derecho de la raíz como puede estar incompleto tenemos en cuenta el número de puesto de la cima para corregir el cálculo. En la Figura 29b se puede ver que los vértices 4 a 7 tienen un uno en el tercer bit de su representación en binario, mientras que en los vértices 1 a 3 tienen un cero.

En la estructura de datos almacenamos la raíz y cima del árbol-pila binario, y el puesto de la cima. Dada la condición de búsqueda de nuestro problema particular, en cada vértice del árbol debemos tener acceso a las coordenadas del punto que corresponda, así como las de los puntos anterior y siguiente en el camino convexo. Definimos dicha estructura que a su vez se apoya en la estructura de datos vértice de árbol debajo de la Figura 30, en la que representamos un árbol-pila binario con cinco vértices, en la que indicamos con una *N* los punteros con valor nulo y en los nodos del árbol los números representan punteros a puntos (el número indica el orden del punto en el camino convexo).

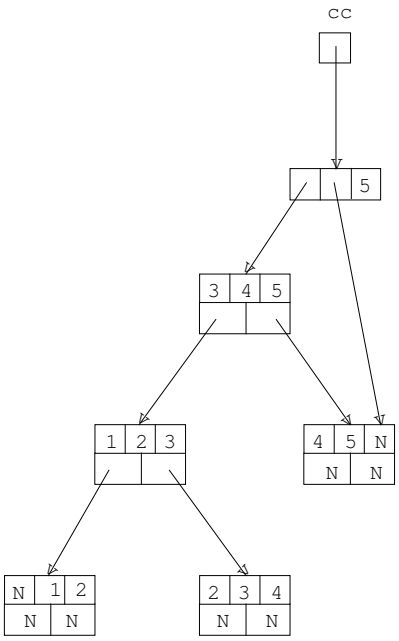

Figura 30. Árbol-pila binario.

```
typedef struct varbol *pvarbol;
struct varbol{   punto *este, *ant, *sig;
                 pvarbol hijoizdo, hijodcho;};
typedef struct{  pvarbol raiz, cima;
                 int puesto;}ccarbol;
```

Seguidamente, exponemos los algoritmos de las operaciones de *push, pop, buscar, partir* y *concatenar* sobre árboles-pila binarios.

3.3.1.1 Operaciones push y pop

Para insertar un nuevo elemento en un árbol-pila binario calculamos el número puesto que le corresponde que será uno mayor al del vértice cima. Si es un número impar se cuelga como hijo derecho del vértice cima. Si es par, el número de unos menos uno del número en binario de puesto del vértice a añadir indica el número de veces a desplazarnos por un hijo derecho desde la raíz, y el subárbol al que llegamos se cuelga como hijo izquierdo del nuevo y el nuevo toma la posición de la raíz del subárbol. En la Figura 31a insertamos el vértice 13 como hijo derecho de la cima. En la Figura 31b insertamos el vértice 14 (1110 en binario) en el lugar del vértice 13 al que llegamos desplazándonos desde la raíz dos veces por los hijos derechos, que pasa a ser su hijo izquierdo.

En cada vértice del árbol introducimos un puntero al punto correspondiente y jugando con la cima introducimos los punteros a los puntos anterior y siguiente en el camino convexo. El vértice insertado es la nueva cima.

Para borrar el último elemento insertado, la cima, buscamos en el árbol el padre de la cima bajando desde la raíz por los vértices más a la derecha en cada nivel del árbol. Si el nº de puesto de la actual cima es impar este vértice padre pasa a ser la nueva cima y el elemento a borrar es su hijo derecho. Si es par, colgamos el subárbol hijo izquierdo de la cima como hijo derecho del padre de la cima encontrado en el árbol. El vértice más a la derecha en este subárbol pasa a ser la nueva cima. En la Figura 31a borramos el vértice 13 que es el hijo derecho del 12, la nueva cima. En la Figura 31b borramos el vértice 14 y colgamos el vértice 13, su subárbol hijo izquierdo, como hijo derecho del vértice 12. El

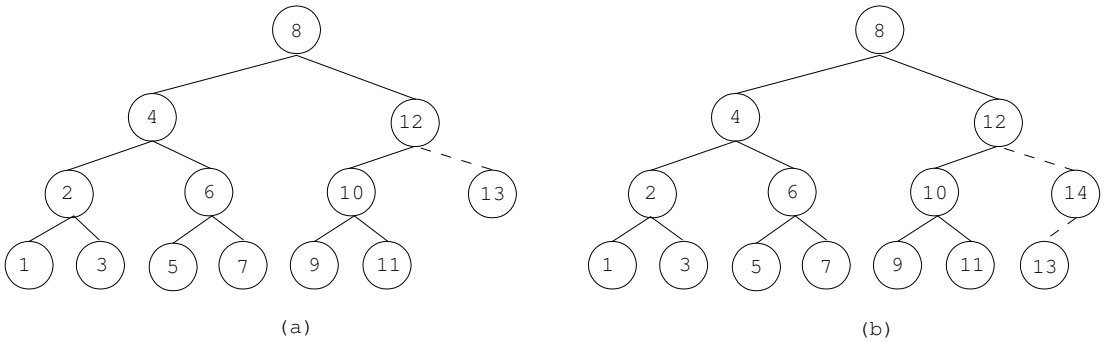

Figura 31. Operaciones de inserción y borrado en árbol-pila binario.

vértice 13, como vértice más a la derecha de este subárbol (y único) es la nueva cima.

3.3.1.2 Operaciones buscar, partir y concatenar

Las operaciones buscar y partir aparecen siempre seguidas en nuestro problema particular por lo que según buscamos un vértice del camino convexo podemos ir partiendo el árbol. El vértice que buscamos va a ser la nueva cima del árbol y su puesto lo calculamos a medida que bajamos por el árbol desde la raíz, aumentando dicho valor en una cantidad en función de la altitud de los vértices por los que pasamos. Cada vez que tomamos una rama a la derecha el número de puesto en binario tendrá un 1, y cuando llegamos al vértice buscado un último 1. Mientras sigamos el árbol desde la raíz por la derecha podemos llegar a un subárbol incompleto, que debemos tener en cuenta para actualizar la altitud. Esta información la obtenemos del número de puesto en binario de la cima actual analizando los unos y ceros.

Los subárboles que separamos porque no van a pertenecer al camino convexo los guardamos en otro árbol junto con el resto de información necesaria para restablecer el árbol-pila binario inicial. Concretamente si el vértice buscado está en el subárbol cuya raíz es el hijo izquierdo, guardamos dicho vértice con su subárbol derecho, y si el vértice buscado está en el subárbol cuya raíz es el hijo derecho, sabemos que dicho vértice y su subárbol izquierdo estarán en el nuevo estado. La información a guardar es el siguiente del vértice encontrado, la cima antigua y su número de puesto. En la Figura 32a representamos mediante flechas la búsqueda en un árbol-pila binario de 17 vértices el recorrido que hacemos hasta llegar al vértice 11 buscado. El cálculo del número de puesto de la nueva cima sería 16*0 + 8*1 + 4*0 + 2*1 + 1*1. En la Figura 32b se muestra el árbol resultado

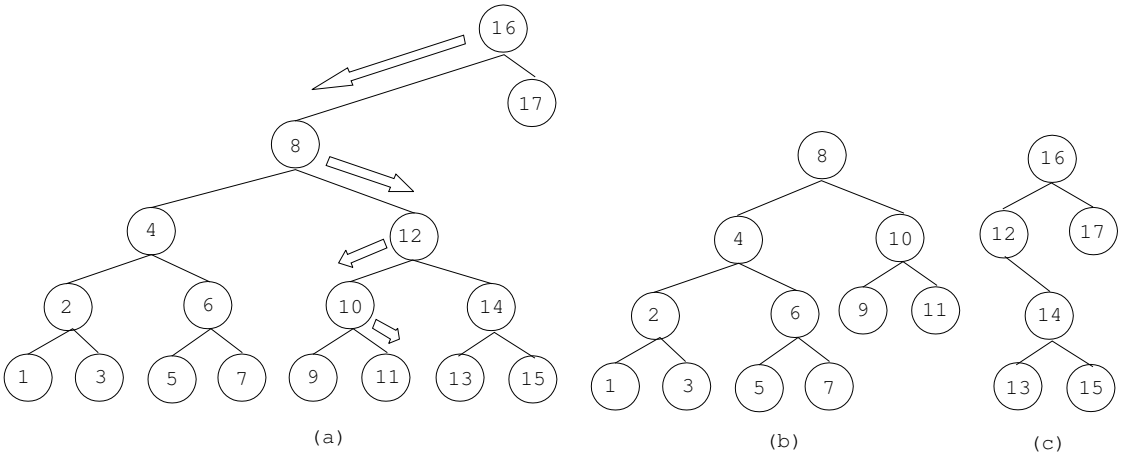

Figura 32. Operaciones buscar, partir y concatenar en árbol-pila binario.

en el que se han eliminado los subárboles derechos con raíz 16 y 12 por sendos desplazamientos por un hijo izquierdo, que guardamos en el árbol de la Figura 32c.

El proceso para concatenar dos árboles-pila binarios es totalmente inverso y sólo tenemos que desandar los pasos dados para partirlo. Para regenerar el árbol en Figura 32a a partir de los de la Figuras 32b y 32c utilizamos la información que aporta el número de puesto en binario tanto de la cima actual como la del árbol a restablecer.

3.3.2 Listas

En una lista el acceso a los vértices del camino convexo que representa es secuencial, es decir, primero leemos la cima, luego el vértice anterior y así hasta llegar a los demás vértices. A cambio, la implementación de las operaciones es muy sencilla. Además de la cima, guardamos cual es el primer vértice del camino, dato necesario en las operaciones de *buscar-partir* y *concatenar*. Definimos dicha estructura que a su vez se apoya en la estructura de datos vértice de lista debajo de la Figura 33, en la que representamos una lista con cuatro vértices, en la que indicamos con una *N* los punteros con valor nulo (almacenamos una copia de las coordenadas de los puntos y no un puntero a los mismos).

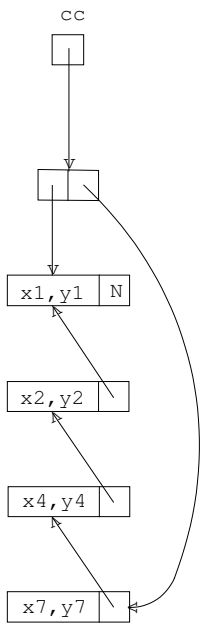

Figura 33. Lista.

```
typedef struct vlista *pvlista;
struct vlista{   punto pto;
                 pvlista ant;};
typedef struct{pvlista primero, cima;}cclista;
```

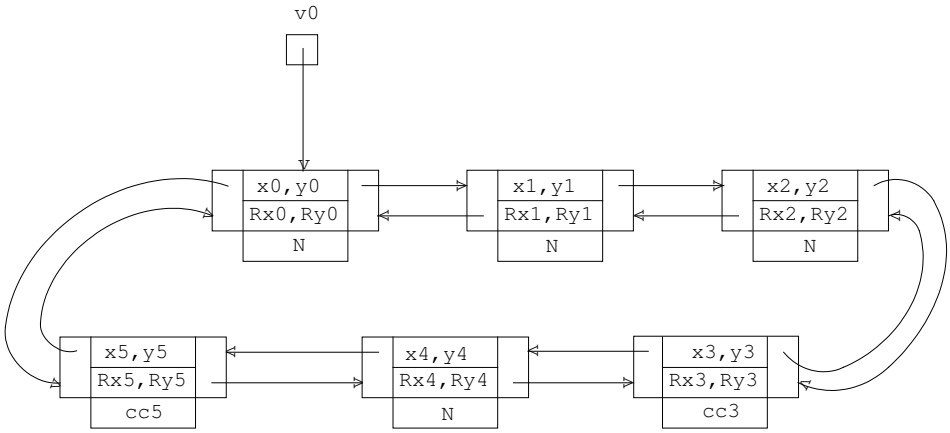

Figura 34. Polígono doblemente enlazado.

```
typedef struct vdoble *pvdoble;
struct vdoble{   punto pto, intercept;
                 pvdoble ant, sig;
                 union{      ccarbol *log;
                             cclista *sec;}ccv;};
```

3.4. Polígono doblemente enlazado

Para aplicar el algoritmo RIGHTSCAN sobre un polígono necesitamos que los vértices estén doblemente enlazados, como ya hemos visto, para poder acceder tanto al vértice anterior como al siguiente. Además por cada vértice visible tenemos otros dos datos a almacenar que son su interceptación derecha y la cola concatenable (en una de sus dos implementaciones). Para ello, nos apoyamos en la estructura de datos vértice doble que permite almacenar toda esta información y que definimos debajo de la Figura 34, en la que representamos un polígono construido a partir de vértices dobles, en la que indicamos con una *N* los punteros con valor nulo y en los vértices representamos con las letras *cc* y un número el puntero a su cola concatenable (el número indica el orden del vértice en el polígono inicial).

3.5. Cadena

Al aplicar el algoritmo que calcula la forma estándar de un polígono *P*, necesitamos una estructura de datos que permita ordenar los cortes del contorno del polígono *bd(P)* con la línea que contiene el ancla. Además, debe guardar la información sobre el orden lineal definido entre los vértices del polígono afectados, es decir la distancia a los vértices del ancla. Para simplificar la implementación también vamos almacenando en la estructura el orden entre los vértices expresado en *bd(P)*, y en lugar de almacenar las coordenadas de

los vértices guardamos un puntero al mismo vértice doble que constituye el polígono. Definimos dicha estructura que a su vez se apoya en la estructura de datos vértice de corte debajo de la Figura 35, en la que representamos una cadena con cuatro vértices de corte, en la que indicamos con una *N* los punteros con valor nulo y en la parte superior de los vértices de corte los números representan punteros a vértices dobles (el número indica el orden del mismo en el polígono inicial).

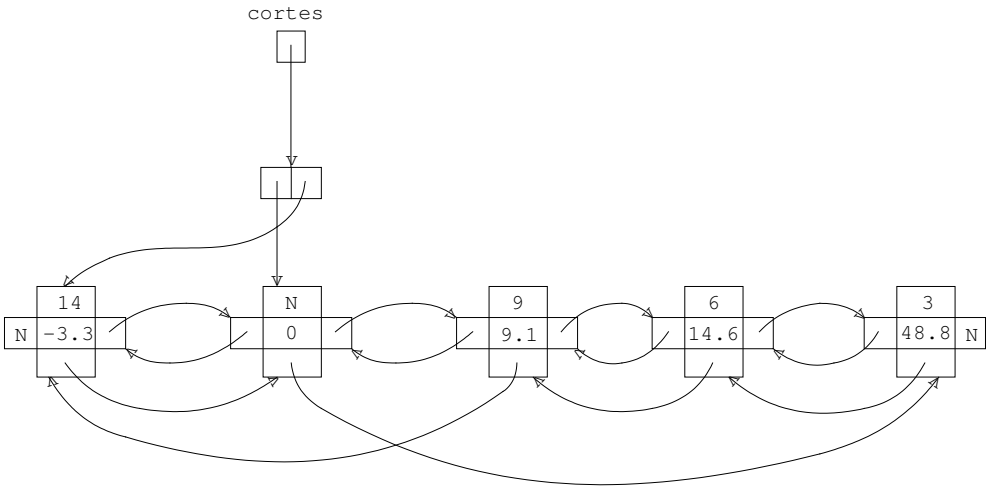

Figura 35. Cadena.

```
typedef struct vcorte *pvcorte;
struct vcorte{   float d;
                 pvdoble vd;
                 pvcorte ant, sig, orden;};
typedef struct{pvcorte v0v1, ultimo;}cadena;
```

4. IMPLEMENTACIÓN

En este capítulo se comentan aspectos del desarrollo de la aplicación que implementa los algoritmos y estructuras de datos expuestos en los capítulos anteriores. Comenzamos explicando las razones para la elección del lenguaje de programación. Seguidamente se relacionan los pasos realizados para convertir el código original en C a C++ para incorporar un interfaz gráfico tipo Windows. Por último se presentan las funciones desarrolladas clasificadas según la estructura de datos que las utilizan.

4.1. Elección del lenguaje

Un requerimiento básico a la hora de elegir un lenguaje apropiado para el desarrollo de un Trabajo Fin de Carrera es que esté ampliamente difundido en la comunidad universitaria. Esto facilita su manejo y reutilización en otros trabajos e investigaciones. Adicionalmente sería deseable un lenguaje simple que permita orientar todo el esfuerzo de programación a los problemas inherentes del algoritmo que queremos implementar. En particular, y dadas las estructuras de datos complejas utilizadas en este Trabajo Fin de Carrera, como por ejemplo los árboles-pila binarios, requeríamos un lenguaje que permitiera el uso de punteros para implementar de una forma natural tanto estas estructuras como las operaciones que las manejaban.

La elección original fue utilizar el lenguaje de programación C. Otra ventaja que proporciona esta elección es la portabilidad del código, dado que es probablemente el lenguaje con mayor número de compiladores y para las más diversas plataformas que exista. Tampoco hay que olvidar la alta eficiencia de los programas que se generan al utilizar este lenguaje y que pueden ser necesarias cuando se manejan polígonos con gran número de vértices.

Posteriormente se vio la necesidad de utilizar un lenguaje que facilitara la creación de un entorno gráfico amigable. Aunque se hizo un desarrollo en Visual Basic que llamaba al ejecutable original en C con los parámetros necesarios, finalmente se optó por utilizar el lenguaje C++ que posibilita una integración total del propio código en C. De esta forma utilizamos un solo entorno de desarrollo integrado y se facilita su estudio.

Cabe finalmente destacar la disponibilidad de entornos de desarrollo integrado libres para su uso no comercial tanto para el lenguaje C, como para C++. En ellos disponemos de todas las herramientas necesarias para crear una aplicación como son el editor avanzado

de código, el diseñador de ventanas, el compilador, el administrador de proyectos, y el ejecutor y depurador de programas. En el caso de C++ proporciona la biblioteca de clases con todos los componentes de las aplicaciones Windows que queremos incorporar al interfaz gráfico. Este marco de trabajo basado en componentes con propiedades, eventos y métodos ya escritos aceleran el desarrollo de aplicaciones con un esfuerzo mínimo. Además proporcionan un entorno visual amigable, con un sistema de ayuda de todas las funciones del entorno, e incluso de los lenguajes de programación.

4.2. Conversión de C a C++

Aunque C++ está basado en C y es totalmente compatible hay distintos aspectos, por ejemplo las convenciones de llamada a funciones, que dependen del compilador utilizado por lo que como punto de partida utilizamos compiladores del mismo fabricante. Como no queremos traducir nuestras estructuras de datos y funciones en C a la estructura de clases de C++, mezclamos ambos lenguajes. Para ello, introducimos todas las líneas de los archivos cabecera y de código de C salvo las directivas de preprocesamiento “include” entre los siguientes grupos de sentencias.

```
#ifdef __cplusplus
extern "C" {
#endif
(..)
#ifdef __cplusplus
}
#endif
```

Incluimos además los ficheros cabecera de C en los de C++ para poder llamar a las funciones de C desde C++.

El control de la aplicación se realiza desde C++ por lo que la función “main” del código original se renombra como una función más y en un segundo paso trasladamos el código directamente a funciones C++ desde las que llamaremos al resto de funciones C originales.

Las funciones de gestión de memoria de C las sustituimos por las de C++, esto es, “malloc” y “free” por “new” y “delete” respectivamente.

El tratamiento de gráficos cambia conceptualmente en una aplicación Windows ya que es un componente más que se inicializa en C++ y que deberemos pasar como un parámetro, “TPaintBox*”, a cada función C que necesite dibujar. Incorporamos la librería

C++ de este control, <ExtCtrls.hpp>, en los ficheros cabecera de C. Cambiamos además la librería no estándar de C <graphics.h> por la librería gráfica de C++ <Graphics.hpp> y con ella los nombres de las funciones, que ahora contienen letras mayúsculas, y los parámetros, que algunos pasan a ser propiedades. Para el borrado de la pantalla cambiamos la función C "cleardevice" por el método C++ "Repaint".

El manejo de eventos cambia también conceptualmente en aplicaciones Windows. El sistema operativo informa a las aplicaciones de los eventos que las aplicaciones le requieren y ya no son las aplicaciones las que manejan los eventos. El manejo del ratón y de las pulsaciones de teclas requieren ahora dividir las funciones C que las utilizaban en distintos trozos y crear variables de estado para poder interpretar un mismo evento según el momento de la ejecución en que nos encontramos. Ligamos mediante el inspector de objetos los eventos que nos interesan a estas nuevas funciones. Eliminamos las funciones de manejo del ratón de C y la librería no estándar de C, <conio.h>, utilizada para las llamadas a la función C "getch" para leer una pulsación del teclado.

4.3. Funciones por estructura de datos

Empezamos introduciendo funciones generales y seguidamente en cada subsección las que afectan a cada una de las estructuras de datos expuestas en el capítulo 3. En los casos más interesantes se detalla el código utilizado en subsecciones dedicadas.

4.3.1 Funciones generales

Todas las funciones en esta subsección corresponden a controles definidos en C++ y son creadas automáticamente a partir de la creación de los mismos, así como de los eventos que definimos sobre estos controles. Salvo las funciones de esta subsección que están definidas en C++, el resto de funciones están definidas en C.

Las siguientes funciones se corresponden con las distintas opciones de submenús definidas en la ventana principal de la aplicación.

```
void __fastcall FileNew1Execute(TObject *Sender);
void __fastcall FileOpen1Execute(TObject *Sender);
void __fastcall FileSave1Execute(TObject *Sender);
void __fastcall Salvarresultado1Click(TObject *Sender);
void __fastcall FileExit1Execute(TObject *Sender);
void __fastcall Orientacin1Click(TObject *Sender);
void __fastcall FormaEstndar1Click(TObject *Sender);
```

```
  void __fastcall Visibilidaddesdepuntosv0yv11Click(TObject
*Sender);
  void __fastcall RangosVisibles1Click(TObject *Sender);
  void __fastcall Pasoapaso1Click(TObject *Sender);
  void __fastcall Directa1Click(TObject *Sender);
  void __fastcall Contenido1Click(TObject *Sender);
  void __fastcall HelpAbout1Execute(TObject *Sender);
```

Las siguientes funciones se corresponden con los eventos de pulsación de tecla en el teclado y en el ratón así como de movimiento del ratón. Los eventos de manejo del ratón los utilizamos para introducir un polígono nuevo. El modo de dibujo que utilizamos es "pmNotXor" para conseguir el efecto de arrastrar el último lado introducido sobre los anteriores y que no se vayan borrando, ya que tiene en cuenta el color del lienzo sobre el que se escribe además del color del lápiz. Para borrar basta dibujar de nuevo la línea.

```
  void __fastcall FormKeyPress(TObject *Sender, char &Key);
  void __fastcall FormMouseDown(TObject *Sender, TMouseButton
Button, TShiftState Shift, int X, int Y);
  void __fastcall FormMouseMove(TObject *Sender, TShiftState
Shift, int X, int Y);
  void __fastcall FormMouseUp(TObject *Sender, TMouseButton
Button, TShiftState Shift, int X, int Y);
```

4.3.2 Funciones sobre puntos

Creamos las siguientes funciones para dibujar un segmento desde la posición actual del cursor gráfico a un punto, cambiar el cursor gráfico y para dibujar una línea entre dos puntos sin modificar la posición del cursor gráfico.

```
  void lineaa(TPaintBox*, punto);
  void movera(TPaintBox*, punto);
  void linea(TPaintBox*, punto, punto);
```

Creamos las siguientes funciones para cada una de las operaciones definidas entre puntos en la sección *3.1. Punto*.

```
  float distancia(punto, punto);
  char igual(punto, punto);
  punto corte(punto, punto, punto, punto);
  char turn(punto, punto, punto);
  char lturn(char, punto, punto, punto);
  char rturn(char, punto, punto, punto);
```

4.3.2.1 Función lturn

Exponemos el código comentado de la función que determina si hay un giro izquierdo (left turn) entre 3 puntos.

```
char lturn(char rlscan, punto a, punto b, punto c){
/* rlscan indica cómo considerar los signos de los determinantes según la orientación del polígono y teniendo en cuenta la puntualización hecha en 4.3.3.1. Función orientacionps sobre las coordenadas. Si el polígono está orientado (al avanzar por bd(P) el interior queda a la derecha) el análisis es un rightscan. En el leftscan rlscan tiene signo contrario. */
 return(((a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y))*rlscan)>0);
}
```

4.3.3 Funciones sobre polígonos (simplemente enlazados)

Creamos las siguientes funciones para dibujar el contorno de un polígono y el propio polígono de un color determinado.

```
void pintarps(TPaintBox*, pvsimple, TColor);
void rellenarps(TPaintBox*, pvsimple, TColor);
```

Creamos las siguientes funciones para liberar la memoria de un polígono simplemente enlazado y para crear un polígono doblemente enlazado a partir de uno simplemente enlazado.

```
void liberarmps(pvsimple);
void copiarpsapd(pvsimple, pvdoble);
```

Creamos la siguiente función para implementar el algoritmo definido en *2.5. Algoritmo Orientación*. Todos los algoritmos trabajan sobre polígonos doblemente enlazados excepto este que aplicamos sobre cada polígono simplemente enlazado que introducimos con el ratón o recuperamos de un fichero.

```
char orientacionps(pvsimple);
```

4.3.3.1 Función orientacionps

Exponemos el código comentado de la función que calcula la orientación de un polígono (simplemente enlazado).

```
char orientacionps(pvsimple xminant){
/* xminant apunta al vértice anterior a uno de abscisa mínima. Aseguramos que el vértice de x mínima no está alineado con los vértices anterior y siguiente comprobando que la x
```

del vértice que le sigue es distinta. Esta comprobación sólo se puede realizar una vez leído todo el polígono. */

```
 while(xminant->sig->pto.x==xminant->sig->sig->pto.x)
  xminant=xminant->sig;
```

/* Sabiendo que el interior cae a la derecha del vértice con x mínima calculamos el signo del determinante que describe el giro. Si es menor que cero, el giro es a la derecha y el polígono está orientado. Caso contrario el estudio es simétrico y basta considerar los giros derechos como izquierdos y viceversa. Dado que en la pantalla las ordenadas siguen orden inverso al que sería normal en coordenadas cartesianas, se deben considerar los signos de los determinantes a la inversa para trasladar los cálculos a la misma y que un giro a la derecha se vea como tal (multiplicamos por -1). */

```
 return(-1*turn(xminant->pto, xminant->sig->pto, xminant->sig-
>sig->pto));
}
```

4.3.4 Funciones sobre árboles-pila binarios

Creamos las siguientes funciones para gestionar la inicialización y liberación de memoria de esta estructura de datos.

```
  void inicializarccarbol(ccarbol**, int, punto*, punto*);
  void liberarmvarbol(pvarbol);
  void liberarmccarbol(ccarbol*);
```

Creamos las siguientes funciones para implementar las operaciones definidas en la subsección *3.3.1. Árboles-pila binarios*.

```
  void pushlog(punto*, ccarbol*);
  void poplog(ccarbol*);
  ccarbol *findsplitlog(punto, ccarbol*, char);
  void mergelog(ccarbol*, ccarbol*);
```

4.3.4.1 Funciones pushlog, poplog, findsplitlog y mergelog

Exponemos el código comentado de las funciones que implementan las operaciones básicas sobre una cola concatenable implementada con un árbol-pila binario. El nombre termina en "log" porque la complejidad de todas estas funciones es *O(log n)*.

```
void pushlog(punto *pto, ccarbol *ccs){
 pvarbol nuevo, *pva;
 int puestocima=++ccs->puesto;
 nuevo=new struct varbol;
```

/* En cada vértice del árbol introducimos un puntero al punto correspondiente y jugando con la cima introducimos los punteros a los puntos anterior y siguiente en el camino convexo */

```
 nuevo->este=ccs->cima->sig=pto;
```

```
 nuevo->ant=ccs->cima->este;
 nuevo->sig=NULL;
 nuevo->hijodcho=NULL;
 if(puestocima&1){          // si el número de puesto del vértice a añadir es impar
  nuevo->hijoizdo=NULL;
  ccs->cima->hijodcho=nuevo;     // se cuelga como hijo derecho del vértice cima
 }else{
   /* Despreciamos todos los ceros por la derecha del número en binario de puesto del vértice a añadir */
   do puestocima>>=1; while(!(puestocima&1));
   pva=&ccs->raiz;
   /* El número de unos menos uno del número en binario de puesto del vértice a añadir indica el número de veces a desplazarnos por un hijo derecho desde la raíz */
   while(puestocima>>=1) if(puestocima&1) pva=&(*pva)->hijodcho;
   /* El subárbol al que llegamos se cuelga como hijo izquierdo del nuevo y el nuevo toma su posición en el árbol */
   nuevo->hijoizdo=*pva;
   *pva=nuevo;
 }
 ccs->cima=nuevo;
}

void poplog(ccarbol *ccs){
 pvarbol va, *pva;
 if(ccs->puesto--&1){       // si el número de puesto del vértice a borrar es impar
  va=ccs->raiz;
  /* Buscamos el padre de la cima */
  while(va->hijodcho!=ccs->cima) va=va->hijodcho;
  va->hijodcho=NULL;                    // su hijo derecho es la cima a borrar
 }else{
  pva=&ccs->raiz;
  /* Buscamos el padre de la cima */
  while(*pva!=ccs->cima) pva=&(*pva)->hijodcho;
  va=*pva=(*pva)->hijoizdo;          // su hijo izquierdo le sustituye en el árbol
  while(va->hijodcho!=NULL) va=va->hijodcho;    // buscamos la nueva cima
 }
 delete(ccs->cima);
 ccs->cima=va;
 va->sig=NULL;
}

ccarbol *findsplitlog(punto v, ccarbol *ccs, char rl){
 /* En la cola concatenable v (ccv) guardamos la información necesaria para restablecer el estado en v, contenido de la cola concatenable S (ccs) antes de ejecutar la función */
 ccarbol *ccv;
```

/* Calculamos el puesto del vértice buscado según recorremos el árbol binario aumentando dicho valor en una cantidad en función de la altitud del vértice, expresada en la variable incremento */

```
int incremento=1;
char tododerechas=VERDAD, sentido=COMIENZO;
pvarbol va, *percheroccs, *percheroccv;

ccv=new ccarbol;
ccv->puesto=ccs->puesto;
ccv->cima=ccs->cima;
```

/* El incremento inicial viene en función de la altitud del árbol, que podemos deducir del puesto de la cima de S */

```
while(ccs->puesto>>=1) incremento<<=1;
```

/* Empezamos la búsqueda por la raíz del árbol. va apunta siempre al vértice candidato del camino convexo de v_1 a u, a ser el primer vértice visible desde v */

```
va=ccs->raiz;
percheroccs=&ccs->raiz;
percheroccv=&ccv->raiz;
do{
```

/* Miramos si el punto anterior a va es visible desde v (el anterior a la cima siempre lo es) */

```
  if(va==ccs->cima || !rturn(rl, v, *va->este, *va->ant)){
   tododerechas=FALSO;
```

/* Si el vértice buscado está en el subárbol cuya raíz es el hijo izquierdo, guardamos en la cola de v dicho vértice con su subárbol derecho */

```
  if(sentido!=IZQUIERDA){
      sentido=IZQUIERDA;
      *percheroccv=va;
  }
  percheroccv=&va->hijoizdo;                    // tomamos una rama a la izquierda
```

/* Seguimos buscando por el camino convexo dirección hacia v_1 un punto cuyo anterior es no visible desde v */

```
  va=va->hijoizdo;
 }else
```

/* Miramos si va es visible desde v */

```
  if(!rturn(rl, v, *va->sig, *va->este)) break;
  else{
```

/* Cada vez que tomamos un rama a la derecha el puesto será mayor */

```
      ccs->puesto+=incremento;
```

/* Mientras sigamos el árbol desde la raíz por la derecha podemos llegar a un subárbol incompleto en cuyo caso corregimos el incremento en función de la altitud de dicho subárbol */

```
      if(tododerechas)
       while(ccs->puesto+(incremento>>1) > ccv->puesto)
incremento>>=1;
```

```
        /* Si el vértice buscado está en el subárbol cuya raíz es el hijo derecho, sabemos
que dicho vértice y su subárbol izquierdo estarán en el nuevo estado */
        if(sentido!=DERECHA){
         sentido=DERECHA;
         *percheroccs=va;
        }
        percheroccs=&va->hijodcho;              // tomamos una rama a la derecha
        /* Seguimos buscando por el camino convexo dirección hacia u un punto va
visible desde v */
        va=va->hijodcho;
      }
     incremento>>=1;
    /* incremento=1 implica que estamos en una hoja del árbol que será el vértice buscado */
    }while(incremento>1);
    /* Al encontrar el vértice terminamos de calcular el puesto */
    ccs->puesto+=incremento;
    if(sentido==IZQUIERDA) *percheroccs=va;
    if(incremento>1){                           // si no hemos llegado a un vértice hoja
     /* Guardamos el subárbol hijo derecho del vértice va encontrado */
     *percheroccv=va->hijodcho;
     va->hijodcho=NULL;
     /* percheroccv apunta a un hijo izquierdo que ha dejado de serlo */
    }else *percheroccv=NULL;
    /* Guardamos el siguiente del vértice encontrado sobre el valor NULL del siguiente de la
cima antigua de S */
    ccv->cima->sig=va->sig;
    va->sig=NULL;
    ccs->cima=va;                           // el vértice va encontrado es la nueva cima
    return ccv;
}

void mergelog(ccarbol *ccs, ccarbol *ccv){
/* Restablecemos el estado en v con la información de ccv */
     /* La información que aporta el puesto actual de la cima en forma de unos y ceros, la
extraemos restando una cantidad en función de la altitud del vértice, que expresamos en la
variable decremento. */
    int decremento=1;
     /* La variable puestomaximo guarda información contenida en el número de puesto de la
cima del árbol a restablecer, sobre la altitud del subárbol derecho de la raíz. */
    int puestomaximo=ccv->puesto;
    char tododerechas=VERDAD, sentido=COMIENZO;
    pvarbol perchaccs, perchaccv, *pva;

     /* El decremento inicial viene en función de la altitud del árbol que queremos
restablecer, que deducimos del puesto de su cima */
```

```
 while(!((decremento<<=1) > ccv->puesto));
 pva=&ccs->raiz;
 perchaccs=ccs->raiz;
 perchaccv=ccv->raiz;
 /* Que sean iguales equivale en el proceso inverso, findsplit, a que se encontró el vértice
cima */
 while((decremento>>=1)!=ccs->puesto)
 if(decremento > ccs->puesto){   // si en el proceso inverso tomamos rama izda
  tododerechas=FALSO;
  if(sentido!=IZQUIERDA){
      sentido=IZQUIERDA;
      *pva=perchaccv;
  }
  pva=&perchaccv->hijoizdo;
  perchaccv=perchaccv->hijoizdo;
 }else{                          // si en el proceso inverso tomamos rama derecha
  /* La información en forma de unos en el puesto actual se borra a medida que se lee */
  ccs->puesto-=decremento;
  /* Mientras sigamos el árbol desde la raíz por la derecha podemos llegar a un subárbol
incompleto en cuyo caso corregimos el decremento en función de la altitud de dicho
subárbol */
  if(tododerechas){
      puestomaximo-=decremento;
      while(puestomaximo-(decremento>>1) < 0) decremento>>=1;
  }
  if(sentido!=DERECHA){
   sentido=DERECHA;
   *pva=perchaccs;
  }
  pva=&perchaccs->hijodcho;
  perchaccs=perchaccs->hijodcho;
 }
 if(sentido==IZQUIERDA) *pva=perchaccs;
 if(decremento>1) perchaccs->hijodcho=perchaccv;
 perchaccs->sig=ccv->cima->este;
 ccv->cima->sig=NULL;
 ccs->puesto=ccv->puesto;
 ccs->cima=ccv->cima;
 delete ccv;
 ccv=NULL;
}
```

4.3.5 Funciones sobre listas

Creamos las siguientes funciones para gestionar la inicialización y liberación de memoria de esta estructura de datos.

```
void inicializarcclista(cclista**, punto);
void liberarmcclista(cclista*);
```

Creamos las siguientes funciones para implementar las operaciones definidas en la subsección *3.3.2. Listas*.

```
void pushsec(punto, cclista*);
void popsec(cclista*);
cclista *findsplitsec(TPaintBox*, punto, cclista*, char, char);
void mergesec(cclista*, cclista*);
```

4.3.5.1 Funciones pushsec, popsec, findsplitsec y mergesec

Exponemos el código comentado de las funciones que definen las operaciones básicas sobre una cola concatenable implementada con una lista. El nombre termina en "sec" porque accedemos secuencialmente a cada elemento de la pila. Se aprecia que la implementación es mucho más simple que en el caso de los árboles-pila binarios, pero a cambio la complejidad es *O(n)* en la función "findsplitsec".

```
void pushsec(punto pto, cclista *ccs){
 pvlista nuevo;
 nuevo=new struct vlista;
 nuevo->pto=pto;
 nuevo->ant=ccs->cima;
 ccs->cima=nuevo;
}

void popsec(cclista *ccs){
 pvlista vl=ccs->cima;
 ccs->cima=ccs->cima->ant;
 delete(vl);
}

cclista *findsplitsec(TPaintBox *PaintBox, punto v, cclista
*ccs, char rl, char borra){
 cclista *ccv;
 ccv=new cclista;
 ccv->cima=ccs->cima;
 /* Buscamos el vértice siguiente al primero no visible desde v del camino convexo de v1
a u, o en su defecto v1 */
 do{
  /* El primer vértice de la cola concatenable de v es el segundo vértice visible desde v
del camino convexo de v1 a u. La cima de S es el vértice anterior ut, primero visible. */
  ccv->primero=ccs->cima;
  ccs->cima=ccs->cima->ant;
```

```
   /* Borramos la parte del camino convexo visible desde v, es decir, desde u_t a u_m */
   if(borra) lineaa(PaintBox, ccs->cima->pto);
  }while(ccs->cima!=ccs->primero &&
         !rturn(rl, v, ccs->cima->pto, ccs->cima->ant->pto));
  ccv->primero->ant=NULL;
  return ccv;
}

void mergesec(cclista *ccs, cclista *ccv){
  ccv->primero->ant=ccs->cima;
  ccs->cima=ccv->cima;
  delete(ccv);
}
```

4.3.6 Funciones sobre polígonos doblemente enlazados

Creamos las siguientes funciones para dibujar el contorno de un polígono y el propio polígono de un color determinado.

```
  void pintarpd(TPaintBox*, pvdoble, TColor);
  void rellenarpd(TPaintBox*, pvdoble, pvdoble, TColor);
```

Creamos las siguientes funciones para gestionar la inicialización y liberación de memoria de esta estructura de datos. En la última función creamos un polígono doblemente enlazado invirtiendo el sentido de los enlaces a partir de otro doblemente enlazado. Esta opción es más sencilla que duplicar todas las funciones invirtiendo el tratamiento de los punteros a los vértices anterior por siguiente y viceversa.

```
  pvdoble crearinsertarvdoble(pvdoble, pvdoble);
  void cortarbolsillo(TPaintBox*, pvdoble, pvdoble, char*);
  void liberarmpdccarbol(pvdoble);
  void liberarmpdcclista(pvdoble);
  void copiarinvertirpdapd(pvdoble, pvdoble, pvdoble, pvdoble,
pvdoble*, pvdoble*);
```

Creamos las siguientes funciones para implementar los algoritmos definidos en las secciones *2.4. Algoritmo Forma Estándar*, *2.2. Polígono de Visibilidad desde un Lado* y *2.3. Polígono de Visibilidad desde un Vértice*.

```
  char formaestandar(TPaintBox*, pvdoble, pvdoble*, pvdoble*,
char);
  pvdoble onelog(pvdoble, ccarbol*, char);
  pvdoble twolog(pvdoble, ccarbol*, char);
  pvdoble threelog(pvdoble, ccarbol*, char, char*);
```

```
  pvdoble fourlog(pvdoble, ccarbol*, pvdoble, char);
  void scanlog(char, pvdoble, pvdoble, pvdoble);
  pvdoble onesec(TPaintBox*, pvdoble, cclista*, char);
  pvdoble twosec(TPaintBox*, pvdoble, cclista*, pvdoble, char);
  pvdoble threesec(TPaintBox*, pvdoble, cclista*, pvdoble, char,
char*);
  pvdoble foursec(TPaintBox*, pvdoble, cclista*, pvdoble, char);
  void scansec(TPaintBox*, char, pvdoble, pvdoble, pvdoble);
  pvdoble onepto(pvdoble, char);
  pvdoble threepto(pvdoble, pvdoble, char, char*);
  pvdoble fourpto(pvdoble, pvdoble, char);
  void scanpto(char, pvdoble, pvdoble, pvdoble);
```

Creamos la siguiente función para mostrar el rango visible sobre el ancla de cada uno de los vértices visibles. Las interceptaciones izquierdas y derechas de los vértices, que delimitan el rango visible, están en ambos polígonos resultado de las dos pasadas RIGHTSCAN y LEFTSCAN. Las guardamos durante la ejecución del algoritmo paso a paso únicamente para mostrarlas con esta función.

```
  void rangosvisibles(TPaintBox*, pvdoble, pvdoble, pvdoble,
pvdoble);
```

4.3.6.1 Función formaestándar

Exponemos el código comentado de la función que implementa el algoritmo Forma Estándar.

```
char formaestandar(TPaintBox *PaintBox, pvdoble v0, pvdoble
*v1sigp1, pvdoble *v0antp1, char rl){
 cadena *cortes;
 pvcorte vc;
 char vdborrado=FALSO;
 inicializarcadena(&cortes);
 cortesrectav0v1(PaintBox, cortes, v0, rl);
 vc=cortes->v0v1;
 /* Si el vértice siguiente a v1 o el anterior a v0 son los definidos en bd(P) las variables
v1sigp1 y v0antp1 seguirán a NULL */
 *v1sigp1=*v0antp1=NULL;
 /* Si el vértice siguiente a v1 cae en el lado del ancla que da al exterior, tenemos que
calcular el polígono de visibilidad desde el punto v1 */
 if(lturn(rl, v0->pto, v0->sig->pto, v0->sig->sig->pto)){
  vc=vc->orden;
  while(vc->ant!=cortes->v0v1){
   /* Eliminamos los trozos del polígono desde v1 hasta v1->sig, en el polígono en forma
estándar, que caen en el lado del ancla que da al interior */
```

```
    cortarbolsillo(PaintBox, vc->vd, vc->ant->vd, &vdborrado);
    vc=vc->ant->orden;
   }
   *v1sigp1=vc->vd;          // guardamos en v1sigp1 el v1->sig en el polígono en f.e.
  }
  vc=vc->orden;
  while(vc!=cortes->v0v1 && vc->sig!=cortes->v0v1){
   /* Eliminamos los trozos del polígono que caen en el lado de ancla que da al exterior */
   cortarbolsillo(PaintBox, vc->vd, vc->sig->vd, &vdborrado);
   vc=vc->sig->orden;
  }
  /* Análogamente para v0 */
  if(rturn(rl, v0->sig->pto, v0->pto, v0->ant->pto)){
   *v0antp1=vc->vd;
   vc=vc->orden;
   while(vc!=cortes->v0v1){
    cortarbolsillo(PaintBox, vc->vd, vc->ant->vd, &vdborrado);
    vc=vc->ant->orden;
   }
  }
  liberarmcadena(cortes);
  return vdborrado;
}
```

4.3.6.2 Funciones onelog, twolog, threelog, fourlog y scanlog

Exponemos el código comentado de las funciones que implementan el algoritmo RIGHTSCAN y los 4 casos en que se subdivide su análisis en la versión de ejecución directa. El nombre termina en "log" porque las colas concatenables en que se apoyan para almacenar el camino convexo en cada vértice son árboles-pila binarios y las funciones que implementan las operaciones que las manejan tienen complejidad *O(log n)*.

```
pvdoble onelog(pvdoble v, ccarbol *ccs, char rl){
 ccarbol *ccv;
 pvdoble r=v->ant->ant;
 punto um=*ccs->cima->este;
 char lturnvsig, lturnv=FALSO;
 ccv=v->ant->ccv.log;                    // guardamos la cola concatenable de um
 delete(v->ant);                         // borramos um que es invisible
 /* Buscamos, a partir del vértice siguiente a v, el primer vértice v' que no esté dentro de
LLB(um, r). Forzamos que v' no puede ser un punto del segmento u_m,r y por eso la
condición que comprueba el corte de u_m,r con el lado v'->ant,v' permite que v'->ant
sea un punto de u_m,r pero v' no. De esta forma se trata correctamente como onelog() el
```

caso en que u_{m-1}, u_m y v son colineales. */

```
 while(!(lturnvsig=lturn(rl, um, r->pto, v->sig->pto)) || lturnv
       || lturn(rl, v->pto, v->sig->pto, r->pto)){
  v=v->sig;
  delete(v->ant);
  lturnv=lturnvsig;
 }
```

/* Hemos borrado los vértices invisibles excepto el anterior a v'. En lugar de calcular el corte de $\overline{u_m, r}$ y $\overline{\text{v'->ant, v'}}$ como punto que seguirá a r en bd(P), simplificamos tomando v'->ant pues por ser invisible será borrado. La cola concatenable de tal punto es la misma que la del punto um (formada únicamente por r) y así se la asignamos a v'->ant con lo que el manejo de las pilas concatenables para restablecer el estado en dicho punto y en r no se ve afectado por la simplificación. */

```
 v->ccv.log=ccv;                    // asignamos la cola concatenable de u_m a v'->ant
 v->ant=r;
 r->sig=v;
 poplog(ccs);
 pushlog(&v->pto, ccs);
 return v->sig;
}

pvdoble twolog(pvdoble v, ccarbol *ccs, char rl){
 v->ccv.log=findsplitlog(v->pto, ccs, rl);
 pushlog(&v->pto, ccs);
 return v->sig;
}

pvdoble threelog(pvdoble v, ccarbol *ccs, char rl, char
*rturntuv){
 pvdoble u=v->ant, vd;
 punto um=u->pto, up, upp, t, w;
 char corteumvtu=VERDAD, rturnvsig, rturnv=FALSO;
 do{
```

/* Restablecemos el estado en el vértice anterior en el polígono, que es la cima u en la cola S (estado en el vértice actual), borrando la cima de S y concatenando a S la cola de dicho vértice anterior */

```
 poplog(ccs);
 if(u->ccv.log!=NULL) mergelog(ccs, u->ccv.log);
 u=u->ant;                          // la nueva cima es el vértice anterior a la antigua
```

/* Si en el estado anterior el lado $\overline{u_m, v}$ cortaba un pseudo-lado $\overline{t, u}$ cambiamos el lado sobre el que va a caer el vértice r pues dicho lado es enteramente invisible */

```
 if(corteumvtu){
  up=u->sig->pto;
  upp=u->pto;
```

```
 }
 delete(u->sig);                          // borramos la antigua cima que es invisible
 /* Condición de fin del backtrack por bd(P) */
        /* v es visible sólo si t, u y v forman un giro a la derecha */
 }while(!(*rturntuv=rturn(rl, t=*ccs->cima->ant, u->pto, v->pto))
        /* El resto de la condición comprueba si estamos en el caso en que entramos en
un RLB(t, u) enteramente invisible. Comprobamos si el lado u_m,v corta el segmento t,u
y si t,u es actualmente un lado. Si rturntuv==VERDAD el corte se produce si se dan esos
dos giros. */
        && (!(corteumvtu=lturn(rl, um, v->pto, t) &&
        rturn(rl, um, v->pto, u->pto)) || !igual(u->ant->pto,t)));
 if(*rturntuv){
  /* Si hemos borrado ambos vértices del lado donde cae r, al ser invisibles, creamos el
vértice s en ese lado sobre la prolongación de v,u */
  if(!igual(u->pto, upp)){
   vd=crearinsertarvdoble(u, v);
   vd->pto=corte(up, upp, u->pto, v->pto);
   pushlog(&vd->pto, ccs);
  }else v->ant=u;                         // si no creamos s, el anterior a v no está unido
  vd=crearinsertarvdoble(v->ant, v);
  vd->ccv.log=findsplitlog(v->pto, ccs, rl);
  /* Creamos el vértice r sobre la prolongación de u_k,v, la cima que devuelve la función
findsplit es u_k */
  vd->pto=corte(up, upp, *ccs->cima->este, v->pto);

  inicializarccarbol(&v->ccv.log, ccs->puesto+1, ccs->cima->este,
&vd->pto);
  pushlog(&v->pto, ccs);
  v=v->sig;
 }else{
  /* El punto w es la intersección del lado u_m,v con el pseudo-lado t,u */
  w=corte(um, v->pto, t, u->pto);
  /* Buscamos el primer vértice y, a partir del vértice siguiente a v, de un lado x,y que
corte el segmento t,w tal que dicho vértice no sea un punto de este segmento. */
  while(!(rturnvsig=rturn(rl, t, w, v->sig->pto)) || rturnv ||
        rturn(rl, v->pto, v->sig->pto, w)){
   v=v->sig;
   delete(v->ant);
   rturnv=rturnvsig;
  }
  /* Creamos sobre el vértice invisible u el punto visible z, intersección del lado x,y con
t,w */
```

```
  u->pto=corte(v->pto, v->sig->pto, t, w);
  v=v->sig;
  delete(v->ant);
  v->ant=u;
  u->sig=v;
 }
 return v;
}

pvdoble fourlog(pvdoble v, ccarbol *ccs, pvdoble v0, char rl){
 punto umpto=*ccs->cima->este;
 pvdoble um;
 if(lturn(rl, v0->pto, umpto, v->pto)){
  um=v->ant;
```

/* Buscamos el primer vértice w, a partir del vértice siguiente a v, de un lado $\overline{u,w}$ que corte la línea $\overline{v_0,u_m}$ tal que dicho vértice no sea un punto de esta línea. */

```
  while(!rturn(rl, v0->pto, umpto, v->sig->pto)){
   v=v->sig;
   delete(v->ant);
  }
  um->sig=v;
  v->ant=um;
```

/* Creamos sobre el vértice invisible u el punto visible t que cae sobre la prolongación de v_0 a u_m en el lado $\overline{u,w}$ */

```
  v->pto=corte(v->pto, v->sig->pto, v0->pto, umpto);
 }
 pushlog(&v->pto, ccs);
 return v->sig;
}

void scanlog(char rl, pvdoble v0, pvdoble v, pvdoble vfin){
 ccarbol* ccs;              // cola concatenable de S, estado en el vértice en curso
 punto um3;                                          // um del último threelog()
```

/* Variable booleana que indica si existe un LLB(u_m, r) como consecuencia de un threelog() en el v anterior */

```
 char llbumr=FALSO;

 inicializarccarbol(&ccs, 1, NULL, &v0->sig->pto);
 pushlog(&v->pto, ccs);
 v=v->sig;
 while(v!=vfin){
```

/* Caso colineal es tratado como un rturn */

```
  if(!lturn(rl, *ccs->cima->ant, *ccs->cima->este, v->pto))
   if(llbumr && rturn(rl, um3, *ccs->cima->este, v->pto))
```

```
      v=onelog(v, ccs, rl);
    else v=twolog(v, ccs, rl);
  else
    if(rturn(rl, v->ant->ant->pto, *ccs->cima->este, v->pto)){
      um3=*ccs->cima->este;
      v=threelog(v, ccs, rl, &llbumr);
      continue;
    }else v=fourlog(v, ccs, v0, rl);
  llbumr=FALSO;
 }
 liberarmccarbol(ccs);
}
```

4.3.6.3 Funciones onesec, twosec, threesec, foursec y scansec

Exponemos el código comentado de las funciones que implementan el algoritmo RIGHTSCAN y los 4 casos en que se subdivide su análisis en la versión de ejecución paso a paso. El nombre termina en "sec" porque las colas concatenables en que se apoyan para almacenar el camino convexo en cada vértice son listas y en las funciones que implementan las operaciones que las manejan el acceso es secuencial.

```
pvdoble onesec(TPaintBox *PaintBox, pvdoble v, cclista *ccs, char
rl){
 cclista *ccv;
 pvdoble r=v->ant->ant;
 punto um=ccs->cima->pto;
 char lturnvsig, lturnv=FALSO;
 ccv=v->ant->ccv.sec;
 delete(v->ant);
 PaintBox->Canvas->Pen->Color= FONDO;
 linea(PaintBox, v->pto, um);                    // borramos el lado u_m,v
 while(!(lturnvsig=lturn(rl, um, r->pto, v->sig->pto)) || lturnv
       || lturn(rl, v->pto, v->sig->pto, r->pto)){
  v=v->sig;
  delete(v->ant);
  lturnv=lturnvsig;
  lineaa(PaintBox, v->pto);          // borramos los lados desde v hasta v'->ant
 }
 v->ccv.sec=ccv;
 v->ant=r;
 r->sig=v;
 movera(PaintBox, r->pto);
 lineaa(PaintBox, um);     // borramos el lado r,u_m creado en el último threesec()
 lineaa(PaintBox, ccs->cima->ant->pto); // borramos pseudo-lado u_{m-1},u_m
```

```
 PaintBox->Canvas->Pen->Color= RLCVX;
 lineaa(PaintBox, v->pto);          // pintamos el nuevo pseudo-lado u_{m-1},v
 PaintBox->Canvas->Pen->Color= P2;
 linea(PaintBox, v->pto, r->pto);          // pintamos el nuevo lado r,v
 popsec(ccs);
 pushsec(v->pto, ccs);
 return v->sig;
}

pvdoble twosec(TPaintBox *PaintBox, pvdoble v, cclista *ccs,
pvdoble v0, char rl){
 pvdoble vd;
 PaintBox->Canvas->Pen->Color= FONDO;
 movera(PaintBox, ccs->cima->pto);
 /* Mandamos borrar en la función findsplit la parte del camino convexo de S que no va
estar en el camino convexo del estado en v->sig */
 v->ccv.sec=findsplitsec(PaintBox, v->pto, ccs, rl, VERDAD);
 PaintBox->Canvas->Pen->Color= P2;
 /* Repintamos la parte del polígono actualmente visible que pueda haber coincidido con
el camino convexo borrado. Puesto que no disponemos de un puntero al vértice doble u_t,
sólo el valor del punto, repintamos desde v hasta llegar a u_t */
 movera(PaintBox, (vd=v)->pto);
 do lineaa(PaintBox, (vd=vd->ant)->pto); while(!igual(ccs->cima-
>pto, vd->pto));
 PaintBox->Canvas->Pen->Color= RLCVX;
 lineaa(PaintBox, v->pto);          // pintamos el pseudo-lado u_t,v
 /* Calculamos R_v sobre la prolongación de v,u_t */
 v->intercept=corte(v0->pto, v0->sig->pto, ccs->cima->pto, v-
>pto);
 pushsec(v->pto, ccs);
 return v->sig;
}

pvdoble threesec(TPaintBox *PaintBox, pvdoble v, cclista *ccs,
pvdoble v0, char rl, char *rturntuv){
 pvdoble u=v->ant, vd;
 punto um=u->pto, um_1=ccs->cima->ant->pto, up, upp, t, w;
 char corteumvtu=VERDAD, rturnvsig, rturnv=FALSO;
 pvlista vl;
 PaintBox->Canvas->Pen->Color= FONDO;
 linea(PaintBox, um_1, um);          // borramos el pseudo-lado u_{m-1},u_m
 do{
  lineaa(PaintBox, u->pto);      // borramos los lados desde el último u' hasta v
```

```
  popsec(ccs);
  if(u->ccv.sec!=NULL) mergesec(ccs, u->ccv.sec);
  u=u->ant;
  if(corteumvtu){
   up=u->sig->pto;
   upp=u->pto;
  }
  delete(u->sig);
 }while(!(*rturntuv=rturn(rl, t=ccs->cima->ant->pto, u->pto, v-
        >pto)) && (!(corteumvtu=lturn(rl, um, v->pto, t) &&
        rturn(rl, um, v->pto, u->pto)) || !igual(u->ant->pto,t)));
 lineaa(PaintBox, u->pto);                          // borramos el lado u,u'
 if(*rturntuv){
  PaintBox->Canvas->Pen->Color= P2;
  if(!igual(u->pto, upp)){
   vd=crearinsertarvdoble(u, v);
   /* Pintamos el nuevo lado u,s */
   lineaa(PaintBox, vd->pto=corte(up, upp, u->pto, v->pto));
   /* Calculamos Rs sobre la prolongación de u,v */
   vd->intercept=corte(v0->pto, v0->sig->pto, u->pto, v->pto);
   pushsec(vd->pto, ccs);
  }else v->ant=u;
  vd=crearinsertarvdoble(v->ant, v);
  /* En el último parámetro de la función findsplitsec() decimos que no borre la parte del
camino convexo que va a eliminar de S, ya que durante el backtrack no lo fuimos pintando
según restablecíamos el estado */
  vd->ccv.sec=findsplitsec(PaintBox, v->pto, ccs, rl, FALSO);
  /* Pintamos el lado que llega a r, ya sea desde u o desde s */
  lineaa(PaintBox, vd->pto=corte(up, upp, ccs->cima->pto, v-
>pto));
  lineaa(PaintBox, v->pto);                         // pintamos el lado r,v
  PaintBox->Canvas->Pen->Color= RLCVX;
  lineaa(PaintBox, (vl=ccs->cima)->pto);   // pintamos el pseudo-lado u_k,v
  /* Pintamos el camino convexo desde u_{m-1} a u_k */
  while(!igual(um_1, vl->pto)) lineaa(PaintBox, (vl=vl->ant)-
>pto);
  movera(PaintBox, v->pto);
  /* Calculamos Rr sobre la prolongación de v a u_k */
  vd->intercept=corte(v0->pto, v0->sig->pto, ccs->cima->pto, v-
>pto);

  inicializarcclista(&v->ccv.sec, vd->pto);
  v->intercept=vd->intercept;                       // Rv = Rr
  pushsec(v->pto, ccs);
```

```
  v=v->sig;
 }else{
  lineaa(PaintBox, t);                                  // borramos el lado t,u
  movera(PaintBox, v->pto);
  w=corte(um, v->pto, t, u->pto);
  while(!(rturnvsig=rturn(rl, t, w, v->sig->pto)) || rturnv ||
        rturn(rl, v->pto, v->sig->pto, w)){
   v=v->sig;
   delete(v->ant);
   rturnv=rturnvsig;
   lineaa(PaintBox, v->pto);                  // borramos los lados desde v a x
  }
  /* Creamos z sobre u, actualizando la copia de la cima ya que no es un puntero al punto
z sino una copia */
  ccs->cima->pto=u->pto=corte(v->pto, v->sig->pto, t, w);
  v=v->sig;
  lineaa(PaintBox, v->pto);                             // borramos el lado x, y
  PaintBox->Canvas->Pen->Color= RLCVX;
  movera(PaintBox, (vl=ccs->cima)->pto);
  /* pintamos el camino convexo desde u_{m-1} hasta z */
  do lineaa(PaintBox, (vl=vl->ant)->pto); while(!igual(um_1, vl-
>pto));
  movera(PaintBox, u->pto);
  delete(v->ant);
  v->ant=u;
  u->sig=v;
 }
 return v;
}

pvdoble foursec(TPaintBox *PaintBox, pvdoble v, cclista *ccs,
pvdoble v0, char rl){
 punto umpto=ccs->cima->pto;
 pvdoble um;
 if(!lturn(rl, v0->pto, umpto, v->pto))
  /* Calculamos R_v sobre la prolongación de v a u_m */
  v->intercept=corte(v0->pto, v0->sig->pto, umpto, v->pto);
 else{
  PaintBox->Canvas->Pen->Color= FONDO;
  linea(PaintBox, umpto, v->pto);                       // borramos el lado u_m,v
  um=v->ant;
  while(!rturn(rl, v0->pto, umpto, v->sig->pto)){
   v=v->sig;
   delete(v->ant);
```

```
    lineaa(PaintBox, v->pto);                    // borramos los lados desde v a u
   }
   lineaa(PaintBox, v->sig->pto);                // borramos el lado u,w
   um->sig=v;
   v->ant=um;
   movera(PaintBox, v->pto=corte(v->pto, v->sig->pto, v0->pto,
umpto));
   v->intercept=v0->pto;
  }
  PaintBox->Canvas->Pen->Color= RLCVX;
  linea(PaintBox, umpto, v->pto);
  pushsec(v->pto, ccs);
  return v->sig;
}

void scansec(TPaintBox *PaintBox, char rl, pvdoble v0, pvdoble
v, pvdoble vfin){
  cclista* ccs;
  punto um3;
  char llbumr=FALSO;

  inicializarcclista(&ccs, v0->sig->pto);
  v->intercept=v0->sig->pto;
    // Rv2 = v1
  pushsec(v->pto, ccs);
  PaintBox->Canvas->Pen->Color= RLCVX;
  movera(PaintBox, v0->sig->pto);
  lineaa(PaintBox, v->pto);
  v=v->sig;                          // pintamos el camino convexo inicial v1,v2
  while(v!=vfin){
   PaintBox->Canvas->Pen->Color= V;
   lineaa(PaintBox, v->pto);      // pintamos lado del siguiente vértice a analizar
/* En la ejecución paso a paso paramos una vez por cada caso one, two, three y four en
este momento para mostrar el resultado */
   if(!lturn(rl, ccs->cima->ant->pto, ccs->cima->pto, v->pto))
    if(llbumr && rturn(rl, um3, ccs->cima->pto, v->pto))
v=onesec(PaintBox, v, ccs, rl);
    else v=twosec(PaintBox, v, ccs, v0, rl);
   else
    if(rturn(rl, v->ant->ant->pto, ccs->cima->pto, v->pto)){
       um3=ccs->cima->pto;
       v=threesec(PaintBox, v, ccs, v0, rl, &llbumr);
       continue;
    }else v=foursec(PaintBox, v, ccs, v0, rl);
   llbumr=FALSO;
```

```
 }
 liberarmcclista(ccs);
}
```

4.3.6.4 Funciones onepto, threepto, fourpto y scanpto

Exponemos el código comentado de las funciones que implementan el algoritmo SCANPTO y los 3 casos en que se subdivide su análisis, que calcula el polígono de visibilidad desde un vértice. El nombre de las funciones termina en "pto" porque un vértice es un punto y para diferenciarlos de sus análogos que se calculan para un lado.

```
pvdoble onepto(pvdoble v, char rl){
 pvdoble r=v->ant->ant;
 punto u1=v->ant->pto;
 char lturnvsig, lturnv=FALSO;
 delete(v->ant);
 while(!(lturnvsig=lturn(rl, u1, r->pto, v->sig->pto)) || lturnv
       || lturn(rl, v->pto, v->sig->pto, r->pto)){
  v=v->sig;
  delete(v->ant);
  lturnv=lturnvsig;
 }
 v->ant=r;
 r->sig=v;
 return v->sig;
}

pvdoble threepto(pvdoble v, pvdoble v0, char rl, char
*rturnv0uv){
 pvdoble u=v->ant, vd;
 punto u1=u->pto, up, w;
 char rturnvsig, rturnv=FALSO;
 do{
  up=u->pto;                                    // guardamos u' para calcular r
  u=u->ant;
  delete(u->sig);                               // borramos el vértice invisible u'
 }while(!(*rturnv0uv=rturn(rl, v0->pto, u->pto, v->pto)) &&
```

/* El resto de la condición comprueba si estamos en el caso en que entramos en un RLB(t, u) enteramente invisible. Comprobamos si el punto u está en el mismo semiplano que v_0 respecto al lado $\overline{u_1, v}$. */

```
        !lturn(rl, u1, v->pto, u->pto));
 if(*rturnv0uv){
```

/* Creamos el vértice r en el lado $\overline{u, u'}$ sobre la prolongación de v_0 a v */

```
  vd=crearinsertarvdoble(u, v);
```

```
   vd->pto=corte(up, u->pto, v0->pto, v->pto);
  }else{
   w=corte(u1, v->pto, u->pto, v0->pto);
   while(!(rturnvsig=rturn(rl, u->pto, w, v->sig->pto)) || rturnv
         || rturn(rl, v->pto, v->sig->pto, w)){
    v=v->sig;
    delete(v->ant);
    rturnv=rturnvsig;
   }
   u->sig=v;
   v->ant=u;
```

/* Creamos sobre el último vértice invisible el punto visible z intersección del lado $\overline{x,y}$ con $\overline{u,w}$ */

```
   v->pto=corte(v->pto, v->sig->pto, u->pto, w);
  }
  return v->sig;
}

pvdoble fourpto(pvdoble v, pvdoble v0, char rl){
 pvdoble u1=v->ant;
 while(!rturn(rl, v0->pto, u1->pto, v->sig->pto)){
  v=v->sig;
  delete(v->ant);
 }
 u1->sig=v;
 v->ant=u1;
 v->pto=corte(v->pto, v->sig->pto, v0->pto, u1->pto);
 return v->sig;
}

void scanpto(char rl, pvdoble v0, pvdoble v, pvdoble vfin){
```

/* El estado en v está formado por v_0 y v->ant */

```
 punto vantptoant;
 char llbu1r=FALSO;
 while(v!=vfin){
  if(!lturn(rl, v0->pto, v->ant->pto, v->pto))
   if(llbu1r && rturn(rl, vantptoant, v->ant->pto, v->pto))
v=onepto(v, rl);
   else v=v->sig;                    // twopto() queda reducido a esta instrucción
  else
   if(rturn(rl, v->ant->ant->pto, v->ant->pto, v->pto)){
      vantptoant=v->ant->pto;
      v=threepto(v, v0, rl, &llbu1r);
      continue;
   }else v=fourpto(v, v0, rl);
```

```
  llbu1r=FALSO;
 }
}
```

4.3.6.5 Función rangosvisibles

Exponemos el código comentado de la función que muestra los rangos visibles de cada vértice.

```
void rangosvisibles(TPaintBox *PaintBox, pvdoble v0, pvdoble v,
pvdoble vfin, pvdoble vp1){
```
/* Seguimos el sentido de la segunda pasada, que es donde están los vértices visibles definitivos. Las interceptaciones izquierda y derecha de cada vértice están en ambos polígonos P_1 y P_2. Se muestra el rango visible de todos los vértices del polígono en forma estándar exceptuando los dos del ancla */
```
 punto vp1intercept;
```
/* Los pasos a seguir para mostrar el rango visible en v_1->sig del polígono en forma estándar cuando no coincide con v_1->sig del polígono inicial son distintos al pintar y borrar el lado en estudio */
```
 if(v!=v0->sig->sig){
  PaintBox->Canvas->Pen->Color= V;
  movera(PaintBox, v->pto);
  lineaa(PaintBox, v0->sig->pto);                    // pintamos el lado
  PaintBox->Canvas->Pen->Color= VR;
  lineaa(PaintBox, vp1->intercept);          // pintamos el rango visible
  vp1=vp1->ant;
```
/* Si se ha seleccionado ejecución paso a paso en este momento paramos para mostrar el rango visible de v */
```
  PaintBox->Canvas->Pen->Color= FONDO;
```
/* Borramos el ancla con el rango visible pintado */
```
  lineaa(PaintBox, v->intercept);
  lineaa(PaintBox, v->pto);                          // borramos el lado
  v=v->sig;
 }else{
  PaintBox->Canvas->Pen->Color= FONDO;
```
/* El ancla empieza borrada en cada iteración */
```
  linea(PaintBox, v0->pto, v0->sig->pto);
 }
```
/* El cursor gráfico empieza en v_1 en cada iteración */
```
 movera(PaintBox, v0->sig->pto);
 while(v!=vfin){
  PaintBox->Canvas->Pen->Color= V;
```
/* Pintamos el lado del siguiente vértice */
```
  linea(PaintBox, v->ant->pto, v->pto);
  PaintBox->Canvas->Pen->Color= ANCLA;
  lineaa(PaintBox, v->intercept);       // pintamos el ancla desde v1 hasta Rv
```

```
  if(igual(v->pto, vp1->pto)){          // si el vértice está en los dos polígonos
   vp1intercept=vp1->intercept;
   vp1=vp1->ant;
  }else{
   /* Si el vértice v fue creado en la segunda pasada buscamos v->sig despreciando los
vértices invisibles. En estos puntos Rv = Lv. */
   do vp1=vp1->ant; while(!igual(v->sig->pto, vp1->pto));
   vp1intercept=v->intercept;
  }
  PaintBox->Canvas->Pen->Color= VR;
  lineaa(PaintBox, vp1intercept);              // pintamos el rango visible
  PaintBox->Canvas->Pen->Color= ANCLA;
  lineaa(PaintBox, v0->pto);          // pintamos el ancla desde Lv hasta v0
  PaintBox->Canvas->Pen->Color= VR;
  PaintBox->Canvas->Ellipse(vp1intercept.x, vp1intercept.y,
vp1intercept.x, vp1intercept.y);                          // por si Rv = Lv
  /* Si se ha seleccionado ejecución paso a paso en este momento paramos para mostrar
el rango visible de v */
  PaintBox->Canvas->Pen->Color= FONDO;
  /* Borramos el ancla con el rango visible pintado */
  lineaa(PaintBox, vp1intercept);
  lineaa(PaintBox, v->intercept);
  lineaa(PaintBox, v0->sig->pto);
  PaintBox->Canvas->Pen->Color= P2;
  /* Pintamos el lado del vértice del color de P */
  linea(PaintBox, v->ant->pto, v->pto);
  v=v->sig;
 }
 PaintBox->Canvas->Pen->Color= ANCLA;
 lineaa(PaintBox, v0->pto);                          // pintamos el ancla
}
```

4.3.7 Funciones sobre cadenas

Creamos las siguientes funciones para gestionar la inicialización y liberación de memoria de esta estructura de datos.

```
  void inicializarcadena(cadena**);
  void crearinsertarvcorte(float, pvdoble, pvcorte, pvcorte,
cadena*);
  void liberarmcadena(cadena*);
```

Creamos las siguientes funciones para implementar las operaciones definidas en la sección *2.4. Algoritmo Forma Estándar*.

```
  void insertarcorte(pvdoble, cadena*, pvdoble, char);
```

```
  void cortesrectav0v1(TPaintBox*, cadena*, pvdoble, char);
```

4.3.7.1 Funciones insertarcorte y cortesrectav0v1

Exponemos el código comentado de las funciones en que se apoya la función formaestandar para calcular los cortes del contorno del polígono con la recta que pasa por el ancla.

```
void insertarcorte(pvdoble vd, cadena *cortes, pvdoble v0, char
doble){
 pvcorte vc=cortes->v0v1;
 float dv0, dv1;
 dv0=distancia(vd->pto, v0->pto);
 dv1=distancia(vd->pto, v0->sig->pto);
 if(dv0 > dv1){
  /* Esta búqueda es en tiempo lineal */
  while(vc->sig!=NULL && vc->sig->d < dv1) vc=vc->sig;
  crearinsertarvcorte(dv1, vd, vc, vc->sig, cortes);
  /* Al insertar un corte doble seguimos el sentido en bd(P) para determinar el anterior y
siguiente en la cadena de cortes */
  if(doble>0) crearinsertarvcorte(dv1, vd, vc->sig, vc->sig->sig,
cortes);
  if(doble<0) crearinsertarvcorte(dv1, vd, vc, vc->sig, cortes);
 }else{
  while(vc->ant!=NULL && vc->ant->d > -dv0) vc=vc->ant;
  crearinsertarvcorte(-dv0, vd, vc->ant, vc, cortes);
  if(doble>0) crearinsertarvcorte(-dv0, vd, vc->ant, vc, cortes);
  if(doble<0) crearinsertarvcorte(-dv0, vd, vc->ant->ant, vc-
>ant, cortes);
 }
}

void cortesrectav0v1(TPaintBox *PaintBox, cadena *cortes,
pvdoble v0, char rl){
 pvdoble v, vd;
 /* s es el signo del determinante que define el giro v0, v1, v */
 char santant=0, sant, s;
 /* sant en v1->sig */
 sant=turn(v0->pto, v0->sig->pto, (v=v0->sig->sig)->pto);
 while((v=v->sig)!=v0->sig){     // s desde v3 a v0, sant desde v1->sig a v0->ant
  /* Si hemos cambiado al otro lado del ancla */
  if(sant*(s=turn(v0->pto, v0->sig->pto, v->pto)) < 0){
   PaintBox->Canvas->Pen->Color= FONDO;
   movera(PaintBox, v->pto);
   lineaa(PaintBox, v->ant->pto); // borramos el lado que corta la línea v0,v1
```

/* Añadimos un vértice sobre el lado donde corta la línea $\overline{v_0, v_1}$ */

```
   vd=crearinsertarvdoble(v->ant, v);
   vd->pto=corte(v->ant->ant->pto, v->pto, v0->pto, v0->sig-
>pto);
   PaintBox->Canvas->Pen->Color= P0;
```

/* Pintamos los dos lados surgidos al añadir el vértice */

```
   lineaa(PaintBox, vd->pto);
   lineaa(PaintBox, v->pto);
   insertarcorte(v->ant, cortes, v0, FALSO);
  }else
```

/* Si el punto anterior caía en la línea $\overline{v_0, v_1}$ consideramos donde caen el punto anterior (santant) y siguiente (s) a éste */

```
   if(!sant)
    switch(santant*s){
     case -1:            // si cambiamos de lado del ancla al pasar de uno al otro
      insertarcorte(v->ant, cortes, v0, FALSO);
      break;
```

/* Si uno de los dos cae en la línea $\overline{v_0, v_1}$ y el otro no y entre los tres forman un giro a la izquierda */

```
     case 0:
      if((santant || s) && lturn(rl, v->ant->ant->pto, v->ant-
>pto, v->pto))
       insertarcorte(v->ant, cortes, v0, FALSO);
      break;
```

/* Si ambos caen en el mismo lado de la línea $\overline{v_0, v_1}$ y entre los tres forman un giro a la izquierda. Insertamos el corte dos veces seguidas siguiendo el sentido en bd(P) para definir el anterior y siguiente en la cadena de cortes */

```
     case 1:
      if(lturn(rl, v->ant->ant->pto, v->ant->pto, v->pto))
       insertarcorte(v->ant, cortes, v0, s*rl);
    }
  santant=sant;
  sant=s;
 }
 cortes->ultimo->orden=cortes->v0v1;
}
```

5. MANUAL DE USUARIO

En este capítulo se explica el funcionamiento de la aplicación desarrollada. En cada sección se explica cada uno de los menús que contiene, además de una sección inicial para mostrar los elementos de la ventana principal.

5.1. Ventana Principal

La ventana principal de la aplicación consta de los siguientes elementos, como se puede apreciar en la Figura 36:

- **Barra de título**, con el título del algoritmo central del trabajo.
- **Barra de menús**, para acceder a las distintas funciones de la aplicación.

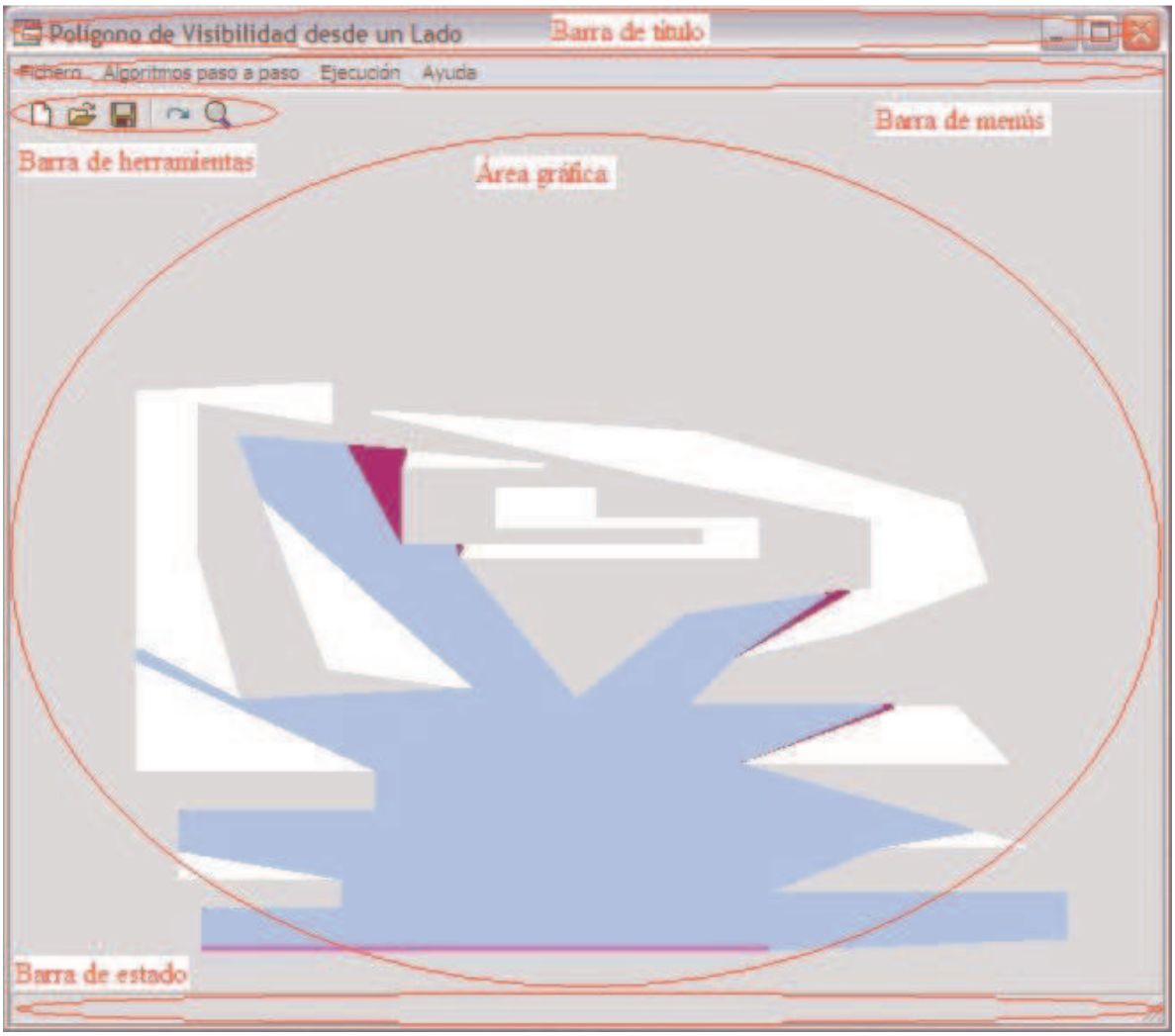

Figura 36. Ventana principal de la aplicación.

- **Barra de herramientas**, para un acceso rápido mediante un botón a las principales funciones de la aplicación.

- **Área gráfica**, para representar el polígono sobre el que vamos a aplicar los algoritmos desarrollados.

- **Barra de estado**, en la que se indican mensajes de ayuda sobre el uso de los elementos que se van apuntando con el ratón.

5.1.1 Guía de utilización rápida

La secuencia de utilización típica de la aplicación consistiría en:

- Introducir el polígono que vayamos a analizar desde alguna de las opciones del menú "Fichero". Desde ese momento se mostrará siempre en pantalla el polígono con el que vamos a trabajar.

- Seleccionar el lado desde el que vamos a realizar el estudio. Con cada pulsación de una tecla del teclado avanzamos un vértice por el contorno del polígono. En

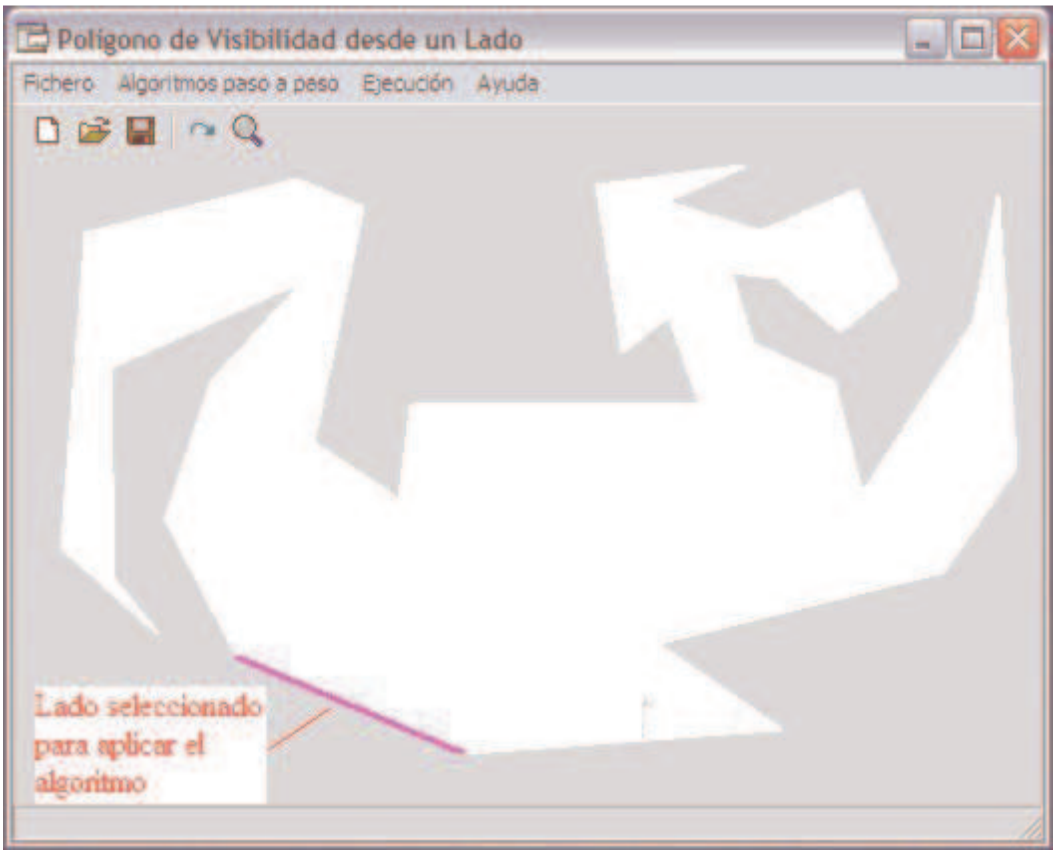

Figura 37. Modo de selección de lado.

la Figura 37 mostramos este modo de selección de lado.

- Seleccionar en el menú "Algoritmos paso a paso" los algoritmos que queramos mostrar además del algoritmo central del trabajo en la ejecución paso a paso.

- Ejecutar los algoritmos de una forma directa o paso a paso desde el menú "Ejecución". En la opción directa se muestra el resultado final de ejecución de los algoritmos desarrollados, y con cada pulsación de la barra espaciadora se analiza el siguiente lado siguiendo el contorno del polígono. En la opción paso a paso avanzamos de un estado a otro para poder analizar los estados intermedios. En los algoritmos previos al central del trabajo se muestran los pasos intermedios separados por una pausa de un segundo. En el algoritmo central y para ver los rangos visibles pasamos de un estado a otro pulsando cualquier tecla del teclado salvo la de escape, "Esc", con la que salimos del modo de ejecución paso a paso.

- Eventualmente podemos seleccionar las opciones en el menú de Ayuda para consultar información detallada de la aplicación o del propio trabajo.

- Notar que la tecla de escape, "Esc", además de salir de los dos modos de ejecución sirve igualmente para redibujar el polígono en el caso de que se borre accidentalmente del área gráfica, por ejemplo, al redimensionar la ventana principal o mover otra ventana por delante.

5.2. Menú Fichero

Este menú da acceso a comandos relacionados con ficheros. Consta de las siguientes entradas, como se puede ver en la Figura 38:

Figura 38. Menú Fichero.

- **Nuevo**, (Figura 39) para introducir un nuevo polígono mediante el ratón. Una vez seleccionada esta entrada, pinchar en el "Área gráfica" con el botón izquierdo del ratón para introducir el primer vértice que se indicará con un punto. A continuación vamos introduciendo el resto de vértices del polígono pinchando de nuevo con el botón izquierdo. Cuando los hayamos introducido todos pulsar el botón derecho del ratón para cerrar el polígono uniendo el primer y último vértices introducidos. Notar que cada vértice se toma cuando dejamos de pulsar el botón izquierdo del ratón y que desde que pulsamos dicho botón podemos mover por pantalla el ratón para arrastrar el lado que forma con el penúltimo vértice introducido.

- **Abrir**, para recuperar un polígono que tengamos almacenado en un fichero.

- **Salvar original**, para guardar el polígono original de entrada en un fichero.

- **Salvar resultado**, para guardar el polígono de visibilidad resultado de aplicar el

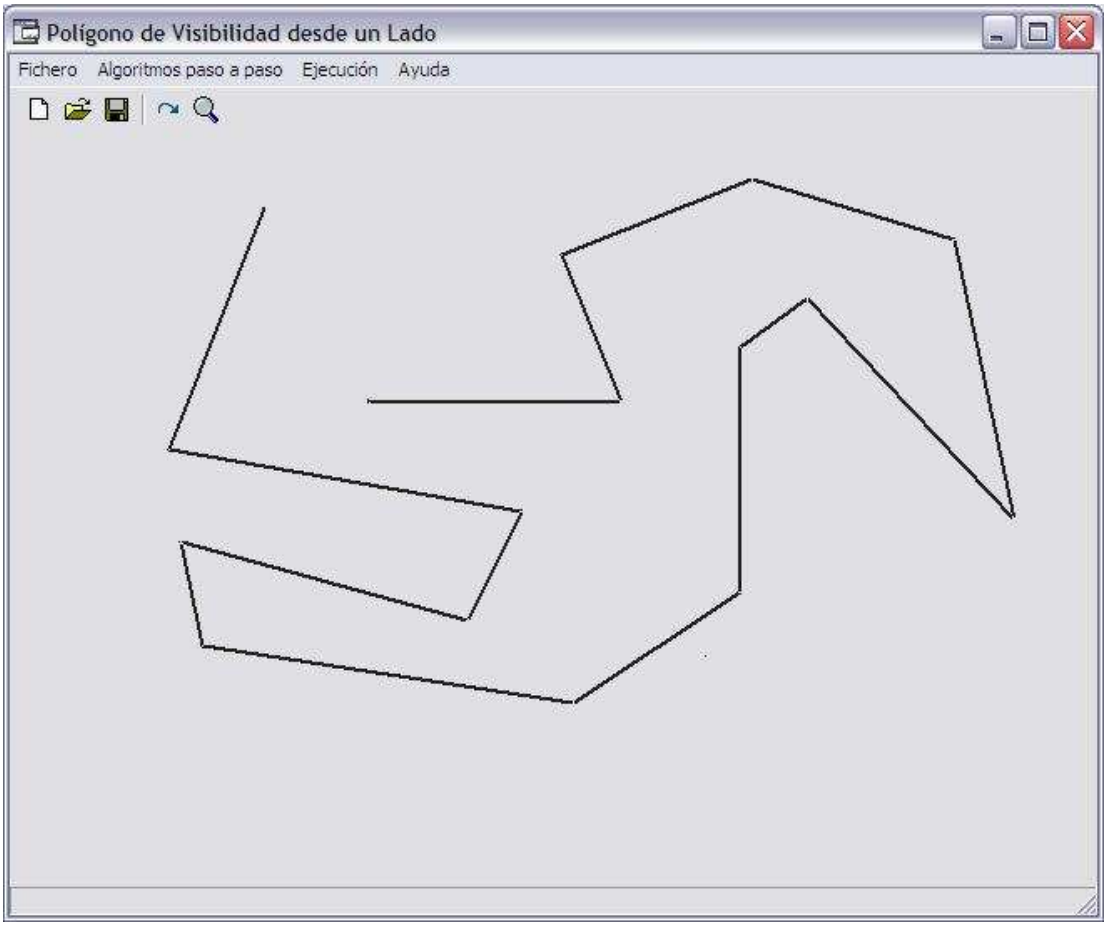

Figura 39. Entrada de menú Nuevo para introducir un polígono con el ratón.

algoritmo.

- **Salir**, para terminar la aplicación, cerrando la ventana principal.

5.3. Menú Algoritmos paso a paso

En este menú se seleccionan los algoritmos que queremos mostrar en la ejecución paso a paso y que disparamos desde el menú "Ejecución". Cada entrada del menú es un algoritmo que vamos a ir aplicando sobre el polígono a analizar, y en el mismo orden de arriba a abajo en el que se van a ir mostrando. Cada vez que seleccionamos una entrada del menú, se marca o desmarca con el símbolo "√". Si aparece marcado un algoritmo se mostrarán los pasos intermedios en el modo de ejecución paso a paso, y si no, se mostrará el resultado final y se pasará al siguiente algoritmo.

En el estado inicial aparece únicamente marcada la entrada del menú "Visibilidad desde un Lado". Esta entrada además no se puede seleccionar, por lo que aparecerá siempre marcada.

Los algoritmos indicados en este menú, como se puede ver en la Figura 40, son:

- **Orientación**, calculamos si el polígono está orientado, esto es, si al avanzar por su contorno el interior queda a la derecha. En la ejecución paso a paso se muestra en pantalla el vértice de abcisa mínima no alineada con los vértices anterior y siguiente, que se utiliza para determinar esta condición. Para poder ver en pantalla el sentido que sigue el contorno del polígono al pasar por estos tres vértices se van marcando con un círculo a intervalos de un segundo (Figura 41).

- **Forma Estándar**, pasamos a forma estándar el polígono, es decir, despreciamos la parte del polígono que determina el semiplano que forma el ancla y que da al

Figura 40. Menú Algoritmos paso a paso.

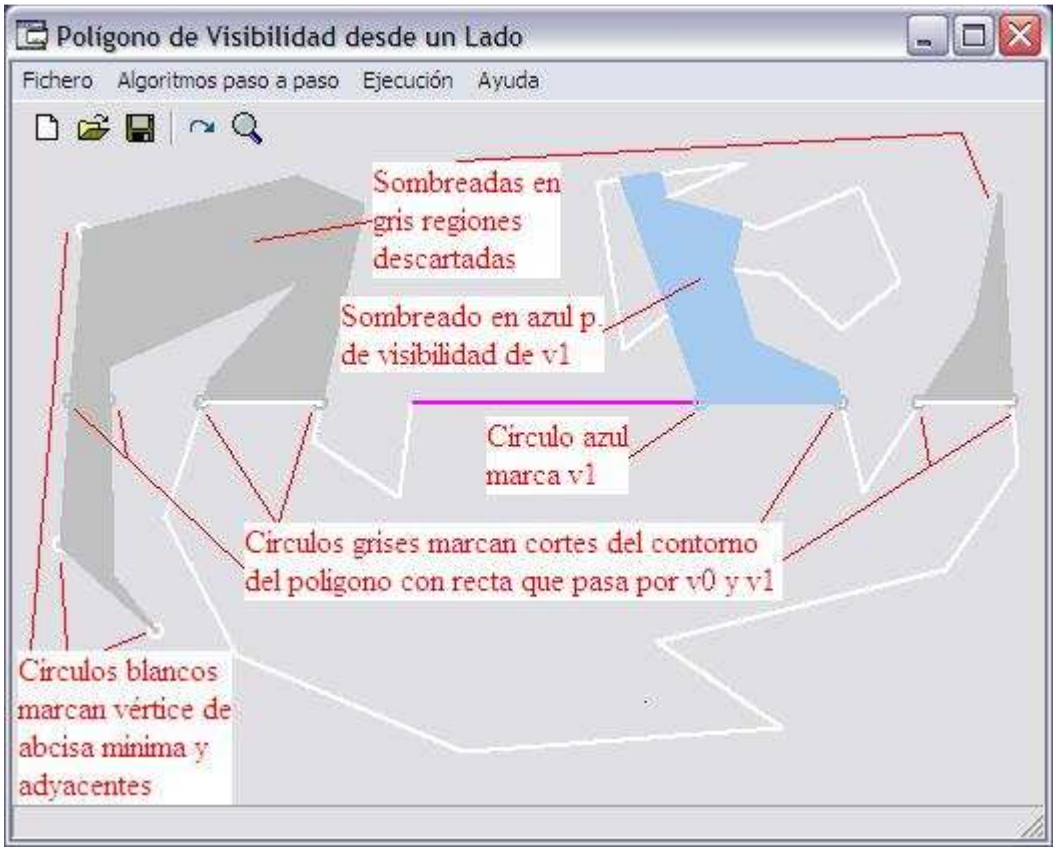

Figura 41. Marcado en ejecución paso a paso en algoritmos previos al algoritmo central.

exterior del polígono. En la ejecución paso a paso se muestran los cortes que se consideran en el algoritmo según recorremos el polígono y se marca en otro color los "bolsillos" que vamos despreciando. Cada corte simple que consideramos se marca con un círculo y cada corte doble con dos círculos concéntricos. La secuencia en que se marcan los cortes y eliminan los "bolsillos" en el algoritmo se muestra a intervalos de un segundo (Figura 41).

- **Visibilidad desde Puntos v_0 y v_1**, aplicamos el algoritmo polígono de visibilidad desde un punto sobre los vértices del ancla en el caso de que haya un trozo del polígono original que al pasarlo a forma estándar ha sido despreciado y que sin embargo es visible desde estos puntos. Estos polígonos forman parte del polígono de visibilidad desde el ancla. En la ejecución paso a paso se marca con un círculo cada vértice del ancla que efectivamente está en este caso y dejando una pausa de un segundo se muestra el polígono de visibilidad desde ese punto. En la Figura 41 marcamos únicamente v_1 y el polígono de visibilidad de v1 en la región descartada del polígono.

Figura 42. Marcado en ejecución paso a paso del algoritmo polígono de visibilidad de un lado.

- **Visibilidad desde un Lado**, es la única entrada que no podemos desmarcar. En la ejecución paso a paso se muestran los estados intermedios de análisis por los que vamos pasando al aplicar el algoritmo. Se marca en distintos colores el ancla, el polígono, el vértice en análisis, el camino convexo en cada vértice y el polígono de visibilidad según se va calculando. A su vez vamos borrando los vértices del polígono original que no forman parte del polígono de visibilidad (Figura 42). Para pasar de un estado a otro basta pulsar cualquier tecla del teclado salvo la de escape, "Esc", con la que salimos del modo de ejecución paso a paso.

- **Rangos Visibles (función)**, muestra el rango visible sobre el ancla de cada uno de los vértices del polígono a excepción de los dos del ancla, v_0 y v_1. Cambiamos de vértice pulsando cualquier tecla del teclado salvo la de escape, "Esc", con la que salimos del modo de ejecución paso a paso. En cada paso se marca en otro color el lado hasta el vértice que está siendo analizado según el sentido de la segunda pasada del algoritmo RIGHTSCAN. El rango visible se indica en otro color sobre el ancla (Figura 43). En el nombre de esta entrada en

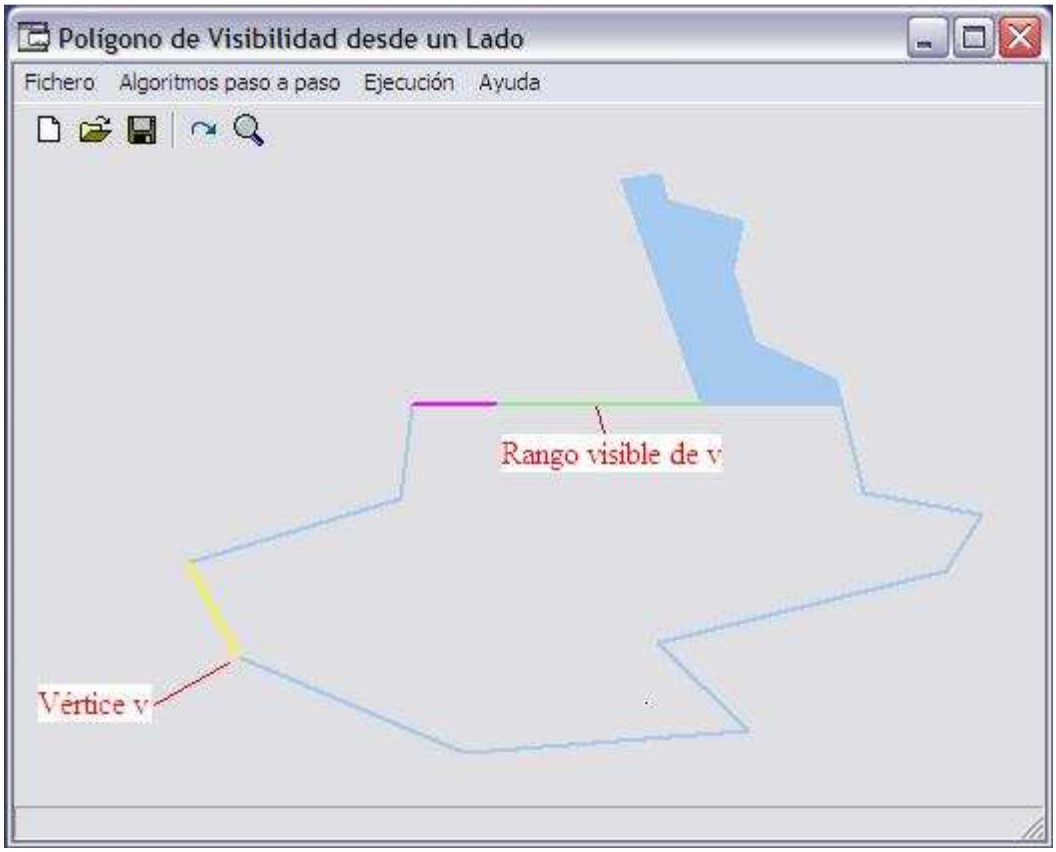

Figura 43. Marcado en ejecución paso a paso del rango visible de v.

el menú indicamos entre paréntesis que es una "función", porque en realidad no es un algoritmo que resuelva un problema.

5.4. Menú Ejecución

Este menú contiene las dos opciones de ejecución de los algoritmos. Consta de las siguientes entradas, como se puede ver en la figura 44:

- **Directa (árboles-pila binarios)**, se muestra directamente el resultado de todos los algoritmos, y con cada pulsación de la barra espaciadora se analiza el siguiente lado siguiendo el contorno del polígono. Con cualquier otra tecla pasamos al modo de selección del lado.

  El resultado de la primera pasada del algoritmo aparece debajo del resultado final tras las dos pasadas y debajo el polígono original. Si al pasarlo a forma estándar se descartó alguna región del polígono aparece en un cuarto color (Figura 36).

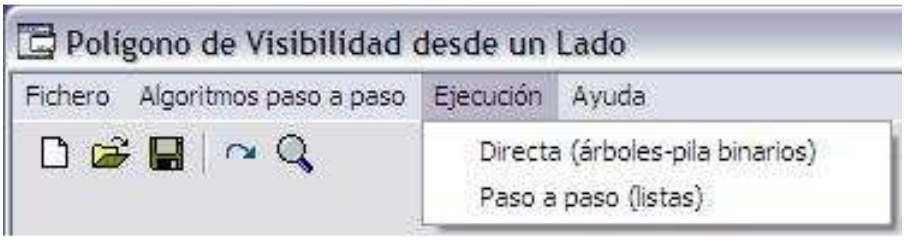

Figura 44. Menú Ejecución.

El algoritmo "Polígono de Visibilidad desde un Lado" utiliza la estructura de datos "árbol-pila binario" para implementar la cola concatenable en la que guardamos el camino convexo derecho que constituye el estado en cada vértice. Con esta implementación cada búsqueda en este camino convexo tiene una complejidad *O(log n)* con lo que la complejidad del algoritmo es *O(n log n)*.

- **Paso a paso (listas)**, se muestran los pasos intermedios más relevantes de los algoritmos seleccionados en el menú "Algoritmos paso a paso". En los algoritmos previos al central del trabajo se hace una pausa de un segundo en cada paso y en el resto se espera una pulsación de tecla para cambiar de estado. Con la tecla de escape, "Esc", salimos del modo de ejecución paso a paso. Una vez termina el último algoritmo pasamos al modo de selección del lado.

  El algoritmo "Polígono de Visibilidad desde un Lado" utiliza la estructura de datos "lista" para implementar la cola concatenable en la que guardamos el camino convexo derecho que constituye el estado en cada vértice. Con está implementación cada búsqueda en este camino convexo tiene una complejidad *O(n)* dado que analizamos cada vértice del camino secuencialmente. A la vez que recorremos estos vértices vamos dibujándolos en pantalla para de esta forma mostrar los pasos intermedios en la aplicación del algoritmo.

5.5. Menú Ayuda

Este menú da acceso a los temas de ayuda de la aplicación. Consta de las siguientes

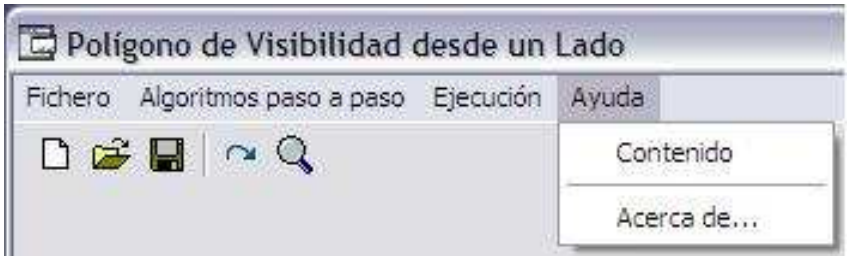

Figura 45. Menú Ayuda.

entradas, como se puede ver en la figura 45:

- **Contenido**, muestra la ayuda sobre el uso de la aplicación así como información detallada de la memoria del trabajo fin de carrera realizado.
- **Acerca de…**, muestra una ventana informativa de la versión, título y copyright de la aplicación.

6. RESULTADOS Y CONCLUSIONES

La parte fundamental del trabajo ha sido el desarrollo de una aplicación con la que poder entender de una forma práctica el algoritmo de *polígono de visibilidad desde un lado* planteado por Lee y Lin. En las distintas pruebas de funcionamiento con ejemplos concretos de polígonos pudimos ver que en algunos casos no se llegaba a la solución correcta, por lo que fue necesario refinar algunos puntos del algoritmo para cubrirlos. En particular Lee y Lin no consideraron un caso degenerado en su subcaso (iii)(a) y que corregimos en la descripción del algoritmo en esta memoria. En la descripción del pseudocódigo del algoritmo RIGHTSCAN en esta memoria exponemos más detalles que en la publicación original para cubrir el caso degenerado comentado y el tratamiento de casos en los que hay puntos alineados.

Para cumplir el objetivo de conseguir una implementación óptima del algoritmo partíamos de la necesidad de utilizar una estructura de datos con unas operaciones básicas que se amoldara a la propia naturaleza del problema. Esta estructura es la cola concatenable. A la hora de implementarla se consideraron las diversas opciones de árboles binarios documentadas con las que realizar las búsquedas binarias. Finamente se desarrolló la implementación que más se ajustaba a las operaciones básicas de una pila. Por la misma razón se decidió utilizar para la implementación un lenguaje de programación que usara punteros con los que implementar de forma eficiente y simple un árbol binario.

Para poder estudiar detenidamente un algoritmo en geometría computacional un objetivo más era desarrollar una implementación paso a paso con la que estudiar detenidamente los estados intermedios. Para conseguirlo ha sido necesario desarrollar una segunda implementación de cola concatenable, con la que pasáramos por cada punto del camino convexo en las operaciones de búsqueda y de esta forma poder dibujar el estado intermedio.

Finalmente es necesario incorporar otros algoritmos en la aplicación para poder aplicar el algoritmo central del trabajo de una forma flexible a cualquier polígono introducido y desde cualquiera de sus lados. El primero de ellos es crear un algoritmo para pasar a forma estándar. Aunque en principio es un problema fácil de resolver, de nuevo los casos particulares requieren hacer un análisis exhaustivo y crear una estructura de datos que se adapte a la naturaleza del problema y que permita una implementación eficiente y simple.

Otro algoritmo que fue necesario desarrollar para apoyar el central fue el polígono de visibilidad desde un punto que se aplica en ambos extremos del lado elegido. Para ello hicimos una analogía del estudio de subcasos planteados para la visibilidad desde un lado y creamos el algoritmo de visibilidad para un punto simplificando cada subcaso. Igualmente para la implementación del segundo algoritmo simplificamos las funciones desarrolladas para el primero.

Adicionalmente, la realización de este trabajo fin de carrera me ha hecho profundizar en aspectos diversos que son necesarios en el desarrollo en una aplicación:

- investigar en conocimientos teóricos específicos de un campo científico concreto, el de la visibilidad y la geometría computacional;
- ver la importancia de elegir un buen algoritmo y una estructura de datos idónea a la hora de diseñar una aplicación;
- documentar en una memoria todo el planteamiento teórico y práctico:
- practicar el uso de lenguajes de programación; y
- practicar el manejo de un entorno de programación.

El resultado ha sido positivo porque he afianzado conocimientos y ganado experiencia en todos estos aspectos del desarrollo de aplicaciones.

7. BIBLIOGRAFÍA

[1] A.V. Aho, J.D. Ullman, J.E. Hopcroft, The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Company, 1974, 108-157 (chapter Data Structures for set manipulation problems)

[2] D. Avis and G.T. Toussaint, An Optimal Algorithm for Determining the Visibility of a Polygon from an Edge, *IEEE Transactions on Computers* **C-30, 12**, 1981, 910-914

[3] B. Chazelle, Triangulating a Simple Polygon in Linear Time, *Discrete Comput. Geom.* **6**, 1991, 485-524. Prelim. version in FOCS 1990

[4] B. Chazelle and L.J. Guibas, Visibility and Intersection Problems in Plane Geometry, *Discrete Comput. Geom.* **4**, 1989, 551-581. Prelim. version in SOCG 1985

[5] L. Davis and M. Benedikt, Computational models of space: Isovists and isovist fields, *Comput. Graph. Image Process* **11**, 1979, 49-72

[6] H.A. El Gindy, An Efficient Algorithm for Computing the Weak Visibility Polygon from an Edge in Simple Polygons, Technical Report, School of Computer Science, McGill University, 1984

[7] H.A. El-Gindy and D. Avis, "A Linear Algorithm for Computing the Visibility Polygon from a Point", J. Algorithms **2**, 1981, 186-197

[8] L.J. Guibas, J. Hershberger, D. Leven, M. Sharir, R.E. Tarjan, Linear time algorithm for visibility and shortest path problems inside simple polygons, *Proc. 2nd ACM Symp. On Computational Geometry*, 1986, 1-13

[9] B. Joe and R.B. Simpson "Corrections to Lee's Visibility Polygon Algorithm". BIT **27**, 1987, 458-473

[10] D.T. Lee, Visibility of a simple polygon, *Computer Vision, Graphics, Image Processing* **22**, 1983, 207-221

[11] D.T. Lee and A.K. Lin, Computing the Visibility Polygon from an Edge, *Computer Vision, Graphics, Image Processing* **34**, 1986, 1-19

[12] M. McKenna, Worst-case optimal hidden-surface removal, *ACM Transactions on Graphics (TOG)* **6, 1**, 1987, 19-28

[13] J. O'Rourke, Art Gallery: Theorems and algorithms, Oxforf University Press, 1987

[14] F.P. Preparata and M.I. Shamos, Computational Geometry, an Introduction, Springer Verlaf, 1985

[15] T.C. Shermer, Recent Results in Art Galleries, *Proceedings of the IEEE*, vol. **80**, 1992, 1384-1399