UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Conversor BRep - STL

Proyecto Final de Carrera

Ingeniería Informática

Autora: Isabel María Gracián Torrent

Director: Eduardo Vendrell Vidal

Septiembre de 2015

Agradecimientos

Quiero agradecer al señor Vendrell que aceptara dirigir este proyecto, así como su amable atención y la paciencia demostradas durante el proceso de desarrollo.

En segundo lugar doy las gracias a Jorge Izquierdo. El más cercano experto en OpenSceneGraph que conozco, y que me ha inspirado para aprender sobre este motor de gráficos aplicándolo a este proyecto.

Y en último lugar, a mis padres, Carlos y Amparo, sin cuyo cariño, apoyo y ánimos hasta en los peores tiempos, jamás hubiese llegado hasta aquí. Hoy soy quien soy gracias a vosotros.

A todos, compañeros, amigos y familiares: gracias por estar ahí.

Resumen

Este proyecto final de carrera trata sobre el proceso de creación de una herramienta software centrada en la conversión del formato de representación de objetos 3D BRep al formato STL, que también define objetos tridimensionales. Esta memoria, trata de exponer el objetivo, análisis del problema, desarrollo de la solución y conclusiones del mismo.

Palabras clave: BRep, Boundary Representation, STL, Stereo Lithography, conversor, modelado 3D, Winged Edge, Half Edge, Winged Triangle, Maximal Element, OSG, OpenSceneGraph, Delaunay, triangulación, ear clipping.

Tabla de contenidos

Tabla de figuras

1. Introducción

CAD, o *Computer-Aided Design* es lo que en español se conoce como el *Diseño Asistido por Computador*. Desde la invención del CAD en 1966, en los últimos años están cobrando protagonismo los programas de modelado 3D, tanto por su utilidad para diseñar y simular objetos, como para una aplicación muy concreta que promete derivar en muchos usos: la impresión 3D.

Las impresoras 3D han hecho posible convertir de forma rápida, económica y sencilla un diseño hecho por computador en algo tangible. Incluso existen ya impresoras 3D de uso doméstico. Con diferentes materiales es posible "imprimir" piezas de repuesto, esculturas, joyería, comida,... Pero todo ello necesita un prototipo. Hay muchos tipos de formatos 3D, pero STL (estereolitografía [1]) es el más común en la impresión 3D. STL es el estándar para este tipo de prototipado.

Este proyecto final de carrera trata sobre el proceso de creación de una herramienta software centrada en la conversión del formato BRep al estándar STL.

BREP proviene de la abreviación del inglés Boundary Representation. Es otro formato de modelado 3D. Como su propio nombre indica, se caracteriza por representar los objetos en base a las superficies que las definen, esto es, las caras, aristas y vértices que las representan. Debido a la gran cantidad de estudios y trabajos realizados sobre el modelado de objetos en BREP, existen bastantes y muy diferentes variantes de diseños de estructuras de datos. Posteriormente, se hablará en esta memoria de algunas de esas variantes.

STL, abreviación de Stereo Lithography (estereolitografía), se caracteriza por definir la geometría de los objetos tridimensionales exclusivamente con caras triangulares. El formato STL estándar no utiliza información de color, textura o propiedades físicas. También es conocido como Standard Tesselation Language.

Para el desarrollo de la herramienta de conversión, se tomará como punto de referencia inicial el proyecto final de carrera realizado por Vicente Matoses, "Visor BREP. Un programa para visualizar modelos 3D" [2], también en la Escuela Técnica Superior de Informática de la Universidad Politécnica de Valencia, en el que se desarrolla un programa para visualizar objetos 3D en formato BRep. En este trabajo, basándose en la técnica de la arista alada (Winged Edge), se redefinió específicamente para este visualizador un formato de archivo BRep, con extensión `.brp` que ya se estaba usando anteriormente, pero que además de los elementos base de BRep, esto es vértices, aristas y caras, incluiría también información sobre los colores de cada cara del objeto que se definiera en el archivo.

El programa analizaba línea por línea el archivo `.brp` para detectar posibles errores de sintaxis, así como evitar incorrecciones en la topología que definiera a un objeto. Una vez pasado con éxito el análisis y dependiendo del modo de programa escogido, se mostraba en pantalla las tablas de vértices, aristas y caras correspondientes o bien la representación gráfica del objeto, pudiendo mover la cámara virtual para inspeccionarlo o hacer uso de un zoom, entre otras funciones.

A partir de las características del formato de archivo `.brp` se estudiará cómo hacer la conversión y qué requisitos son necesarios. No existen más antecedentes, por tanto, en los que apoyarse con la finalidad de lograr la conversión de este formato concreto de archivo BREP a archivo STL.

Centrándonos ahora en dicho objetivo, dado un archivo de formato BREP, de extensión `.brp`, el programa resultante ha de ser capaz de traducir sus datos a otros equivalentes para así obtener su conversión en un archivo de formato STL, de extensión `.stl`, que podrá leer cualquier visor que soporte STL. Técnicamente se pretende transformar las caras poligonales del modelo BREP en su equivalente teselado en caras triangulares, es decir, lo que se llama triangulación, de lo cual existen varias técnicas como la del triangulador de Delaunay (Delaunay Triangulator) o el algoritmo de recorte de orejas (Ear Clipping Triangulation).

La consecución de la conversión, pretende servir también de apoyo al alumnado de las asignaturas relacionadas con el diseño y fabricación asistidas por computador. La razón principal que ha inducido al desarrollo de una herramienta de conversión está relacionada con el hecho de que el programa Visor BREP está implementado bajo la tecnología Java. Con cada nueva actualización de Java, aparecen nuevos problemas de compatibilidad con el visor. Se pretende asegurar que el alumno pueda ver en pantalla de forma fiable el objeto que define en el archivo `.brp`, con el fin de que detecte con mayor facilidad dónde puede haberse equivocado y entender mejor el formato. Por ello, interesa lograr que el programa resultante sea además multiplataforma, y evitar de este modo posibles errores de ejecución.

Por otro lado es también deseable que sea la programación base sobre la que construir una aplicación más completa en el futuro.

Siguiendo estas normas, la tecnología escogida para la implementación será el potente motor gráfico OpenSceneGraph 3 (OSG) [3]. Este presenta las ventajas de ser de código abierto, ser multiplataforma porque sólo requiere C++ estándar y OPENGL, soportar el formato STL y ser altamente escalable, lo que ofrece una gran facilidad de extensión.

Con las ventajas que ofrece OSG, el proyecto puede quedar preparado para ser fácilmente ampliable con otras con nuevas funcionalidades, por otros alumnos o personas interesadas, como por ejemplo el siguiente paso lógico: la conversión desde el formato STL al formato BREP.

La memoria del presente proyecto presenta un estudio de los formatos BREP y STL, hablar del proceso de diseño e implementación de la herramienta de conversión, planteando el problema inicial, analizando los pros y los contras de diferentes soluciones, resultados finales y, por fin, proponer posibles ampliaciones.

Así mismo al final de la memoria, se puede consultar en los anexos instrucciones más concretas sobre cómo reproducir el entorno de desarrollo que se usó para implementar el conversor y las instrucciones de uso para instalar y manejar el programa bajo el sistema operativo Windows.

2. Estudio sobre BRep y STL

Este estudio pretende conocer y comparar las características de diferentes técnicas de representación de sólidos del tipo BRep y la técnica de representación STL.

En primer lugar se revisarán ambos formatos de forma más exhaustiva y, seguidamente, en una tabla resumen en la que se pueden ver de un vistazo todas las características de STL y las de cada técnica de representación BRep. Tras esta, se analizarán los costes de memoria teóricos aproximados que conllevaría representar objetos sólidos básicos en cada técnica de BRep. Por último se hablará de las conclusiones sobre los resultados de cara al proyecto de crear un conversor de formato BRep a formato STL.

2.1. El formato BRep

La expresión BRep proviene de la abreviatura del término inglés, Boundary Representation, esto es, Representación Mediante Fronteras [4] [5]. Es un formato de modelado de objetos sólidos que se centra en las caras que forman la superficie de un objeto.

Un objeto en BRep se asume como un poliedro. De esta manera, se conservan sus propiedades topológicas. Por tanto, los elementos básicos en BRep son los de un poliedro. Es decir, caras, aristas y vértices.

Antes de representar el objeto, es necesario notar cada uno de sus elementos para construir el árbol que lo representa, conocido como árbol BRep. A partir de ahí se puede definir la estructura de datos que se decida más adecuada.

El árbol BRep tiene como raíz el nombre del objeto que representa, el cual está formado por caras (nivel 1). Cada cara se define con un bucle (nivel 2), el cual es un conjunto de aristas ordenadas en sentido horario o anti horario, según conveniencia (nivel 3). En las hojas se sitúan los vértices del objeto, con los cuales se definen las aristas.

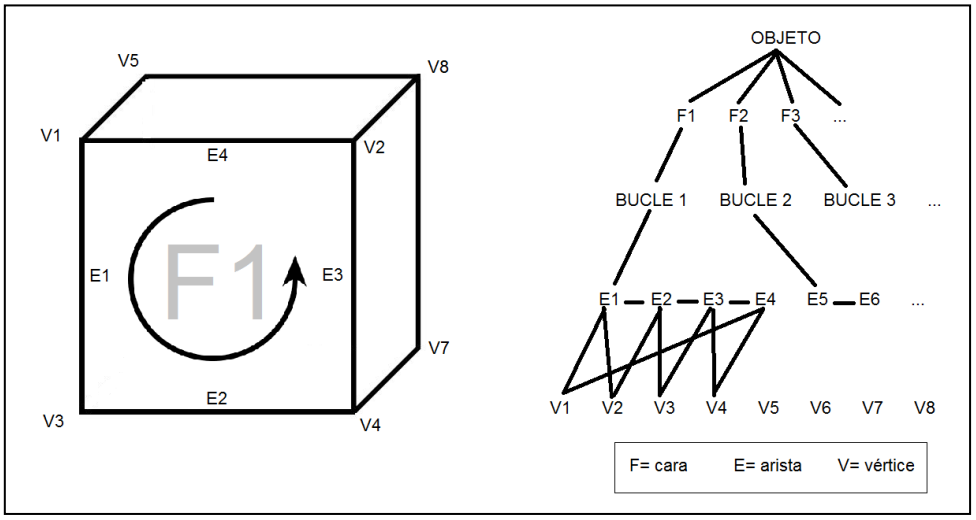


Figura 1. Notación y estructura de un objeto sólido en un árbol BRep

2.2. Representaciones BRep regularizadas

La principal debilidad de BRep por ser un modelo de representación por fronteras es la validez, ya que cualquier conjunto de caras, aristas y vértices no define un sólido válido.

En cada representación influye el dominio de figuras representables: con agujeros o no, grado de multiplicidad del sólido y si permite figuras no regulares. Algunas representaciones engloban a otras y otras son equivalentes entre sí. Si una representación incluye a otra, es fácil hacer traducciones exactas, ya sea directamente o basada en una relación transitiva.

Mantÿla [6] identificó tres tipos de modelos por fronteras:

- Basado en la arista (Arista Alada, Media Arista)
- Basado en el vértice (Triángulo Alado)
- Basado en la cara o polígono (Elemento Máximo)

2.2.1. Arista Alada (Winged Edge)

Introducida por Bruce Baumgart en 1972 [7], la *Arista Alada (Winged Edge)* es la representación más conocida para los objetos BRep.

Para poder representar un objeto sólido, este ha de cumplir ciertas propiedades que le den consistencia topológica:

- Cada arista está delimitada por dos vértices.
- Un objeto sólido debe ser múltiple. Es decir, cada arista sólo debe ser compartida por dos caras (sólidos múltiples). Por eso, la solución es buscar topologías equivalentes. Ver Figura 4.
- Las caras que coinciden en una arista tienen orientación conforme.
- Las aristas sólo hacen intersección en los vértices.
- Las caras sólo hacen intersección en los vértices y aristas.

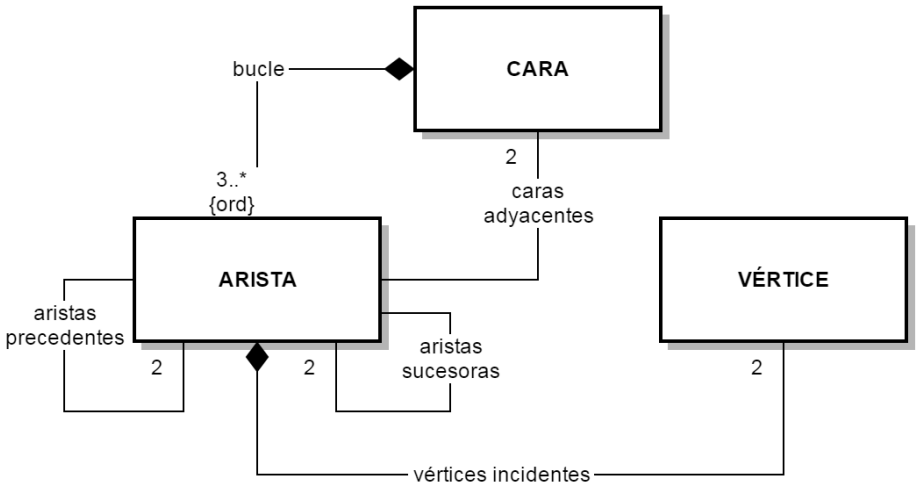


Figura 2. Diagrama de la representación de la Arista Alada

Para guardar la información topológica, necesitamos tomar como elemento central la arista, ya que proporciona las relaciones de vecindad con otras aristas y con las caras. Se construyen tres tablas:

- **Tabla de vértices.** Cada entrada contiene una de las aristas incidentes en él.
- **Tabla de aristas.** Cada arista se define con:
  - Vértices incidentes en ella.
  - Caras adyacentes a ella.
  - Aristas precedentes y sucesoras

- **Tabla de caras.** Cada entrada almacena una de las aristas incidentes en ella.

La tabla de aristas es la más importante y vamos a fijarnos en un ejemplo para ver cómo rellenarla.

Sea el poliedro en forma de cubo de la Figura 3. Tomamos la arista *a* como referencia. Para la entrada correspondiente a esta arista en la tabla de aristas, debemos primero **considerar el cubo visto desde fuera**. A continuación, establecemos que el sentido en el que se mueve la arista es desde el vértice 2 al vértice 1. Así vemos que sus caras adyacentes son A, a la izquierda, y B, a la derecha. Habiendo convenido que los bucles de A y B son en sentido horario, podemos determinar cuáles son las aristas que preceden y que suceden a la arista *a* en cada uno de ellos. En este caso, en la definición del bucle de la cara A, la arista precedente a *a* es la arista *b* y la sucesora es la arista *d*. Mientras que en la definición del bucle de la cara B, la arista precedente a *a* es la arista *e* y la sucesora es la arista *c*.

ARISTA	VÉRTICES INCIDENTES		CARAS ADYACENTES		ARISTAS INCIDENTES			
	desde	hacia	Izq.	Der.	Preced. cara izq.	Suces. cara izq.	Preced. cara der.	Suces. cara der.
A	2	1	A	B	b	d	e	c
…	…	…	…	…	…	…	…	…

VÉRTICE	ARISTA
1	b
2	d
…	…

CARA	ARISTA
A	b
B	c
…	…

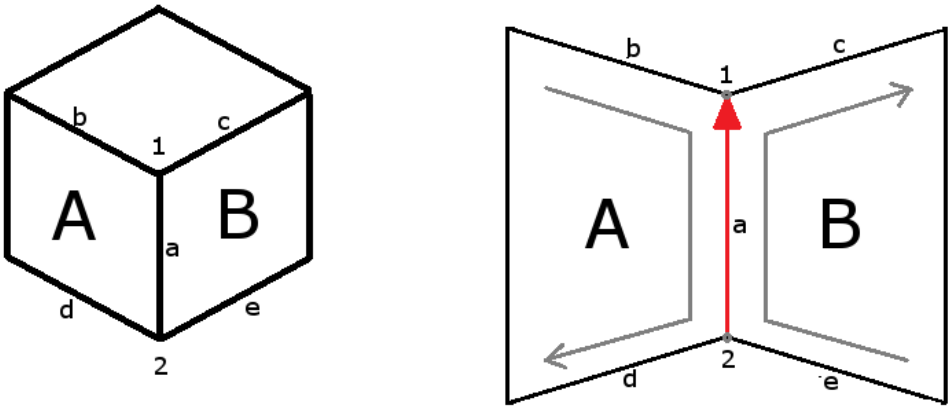


Figura 3. Ejemplo de representación con arista alada

En el caso de que una cara presente más de un bucle, por ejemplo, para representar una hendidura o un agujero, puede hacerse de dos maneras (Ver Figura 5.)

a) Teniendo en cuenta el sentido en el que se definen los bucles. Esto sería, por ejemplo, definiendo el bucle principal en sentido horario y el bucle secundario en sentido anti horario. (Arista Alada modificada de Braid, 1980, [8])
b) Usando una o varias aristas auxiliares para definir con un solo bucle la cara. (Arista Alada con puente de Yamaguchi, 1985, [8])

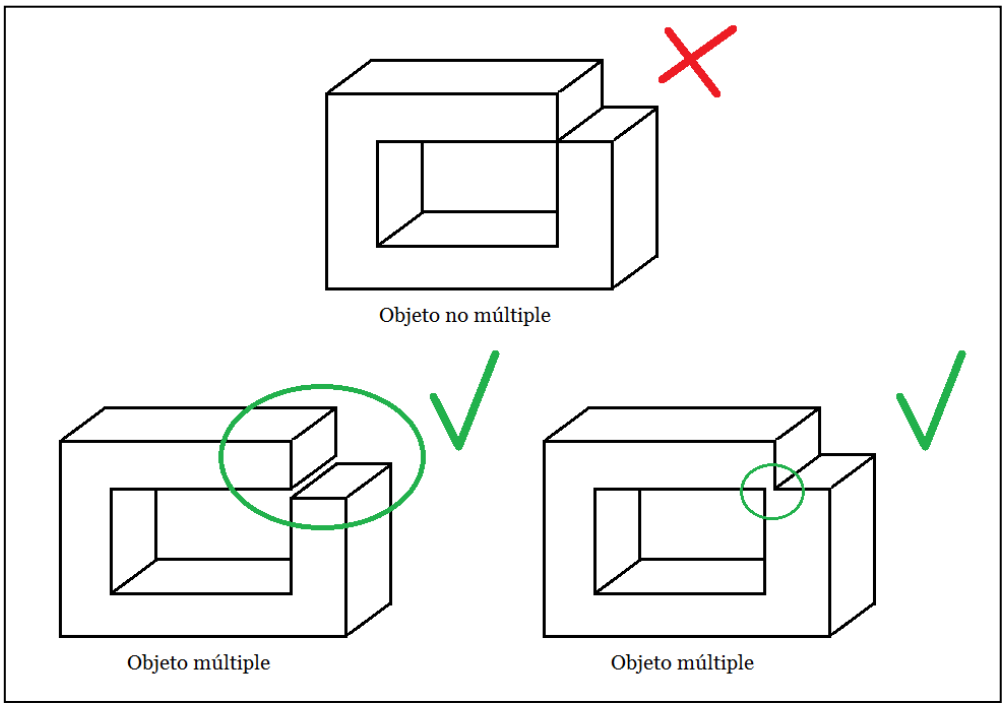


Figura 4. Objeto no múltiple y sus topologías equivalentes válidas para BRep

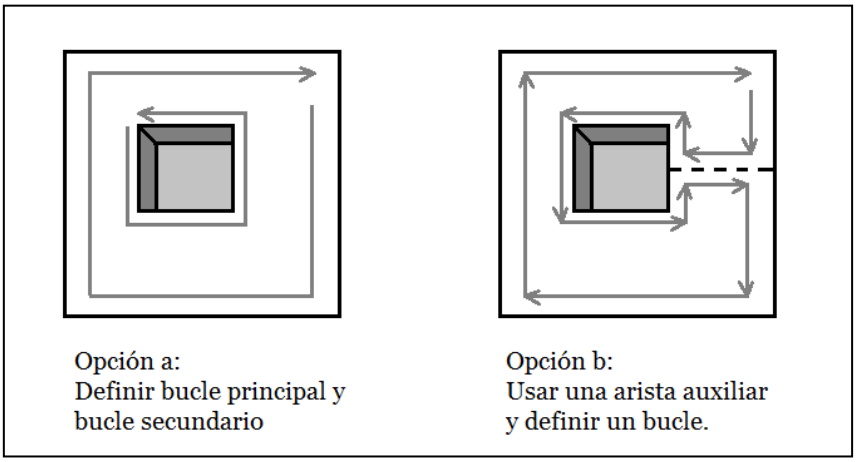


Figura 5. Representación de caras con más de un bucle en BRep

2.2.1.1. Un formato de archivo basado en la Arista Alada: .brp

En el proyecto del Visor BREP [2], se define un archivo `.brp` con la estructura que se muestra a continuación, basada en la representación de la Arista Alada, siguiendo la regla de la mano izquierda. Cada archivo debe contener uno o más objetos sólidos.

Hay dos tipos de caras: normales e internas. Las normales son visibles y las internas son invisibles. Las internas se usan para definir agujeros en la superficie de la cara a la que suceden inmediatamente después en la lista de caras.

Tanto las coordenadas XYZ de los vértices como los valores RGB de los colores han de estar en coma flotante.

```
#Los comentarios se distinguen así.

Objeto nombreobjeto

#Vértices.
V nombrevertice v_x v_y v_z

#Aristas
A a_n v_start v_end c_left c_right a_left_prev a_left_next a_right_prev a_right_next

#Caras
    #Cara normal
    c nombrecara <lista de aristas ordenadas en sentido CW>
    #Cara interna (invisible)
    i nombrecara <lista de aristas ordenadas en sentido
    contrario a las de una cara normal, CCW en este caso>

#Colores
color nombrecolor R G B <lista de caras pintadas de este color>
```

2.2.2. Media Arista (Half Edge)

Esta representación fue introducida por Marti Mantÿla, en 1988 [6]. En ella se asume que cada arista está formada por dos medias aristas, por lo que la media arista es el elemento principal, aunque realmente se basa en la arista, al igual que la representación de la Arista Alada.

Características de la Media Arista:

- Cada media arista está contenida en una única cara.
- Cada cara tiene su propio conjunto único de medias aristas
- El orden en sentido horario de las medias aristas de una cara determinan la orientación de la cara.
- Cada media arista está orientada en sentido opuesto a su otra mitad.
- Dado un par de medias aristas, cada una de ellas está limitada por el mismo par de vértices.

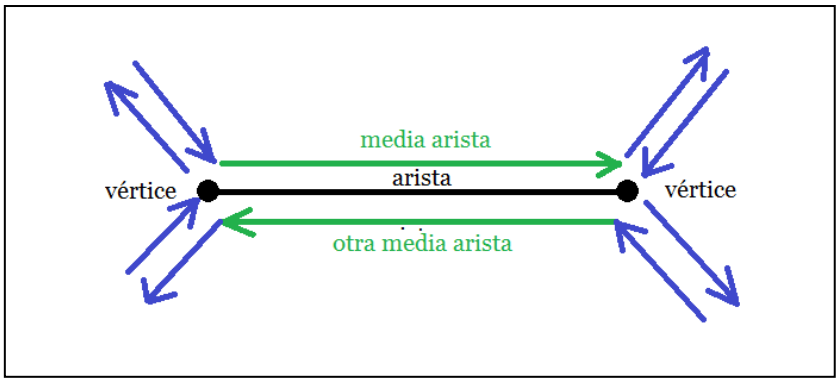


Figura 6. Media arista

Hace falta 6 tablas con la siguiente información básica:

- **Tabla de sólidos:** Cada sólido hace referencia al primer elemento de sus correspondientes listas de caras, aristas y vértices.
- **Tabla de caras:** Cada cara tiene un bucle externo y una referencia a una lista de bucles internos.
- **Tabla de bucles:** Cada bucle hace referencia a la media arista que lo inicia.
- **Tabla de aristas:** Cada arista hace referencia a dos medias aristas.
- **Tabla de medias aristas:** Cada media arista hace referencia al vértice de inicio, el cual indica la dirección del bucle de la que forma.
- **Tabla de vértices**: Cada vértice hace referencia al vértice anterior y posterior en la lista de vértices.

En esta representación todos los elementos del objeto están contenidos en listas doblemente enlazadas, de manera que por ejemplo, un objeto sólido puede hacer referencia a un sólido predecesor y sucesor en la lista de sólidos.

Además, cada cara, bucle, media arista y vértice apunta a su elemento padre: un sólido, una cara, un bucle y una media arista, respectivamente. La representación de la Media Arista admite caras con múltiples bucles, pero no admite piezas anidadas (shells). En la Figura 6 se muestra un diagrama UML con el fin de hacerse una mejor idead de cómo es la estructura de la Media Arista de Mantÿla.

Es posible hacer derivaciones que triangulen cada una de sus caras, como ya se ha implementado en muchos algoritmos de visualización gráfica (Foley, 1992). En consecuencia, pueden derivar en la representación del Triángulo Alado de Paoluzzi, aunque no puede englobarla. De ella se hablará en el siguiente apartado.

Sin embargo, la Media Arista sí engloba a la representación de la Arista Alada de Baumgart, ya que una arista alada es en realidad la suma de dos medias aristas.

La Media Arista puede definir caras hechas de múltiples bucles de aristas, que la estructura de la Arista Alada de Baumgart no puede representar. En consecuencia, la relación de orden de subsunción es de un solo sentido y las dos representaciones no son equivalentes.

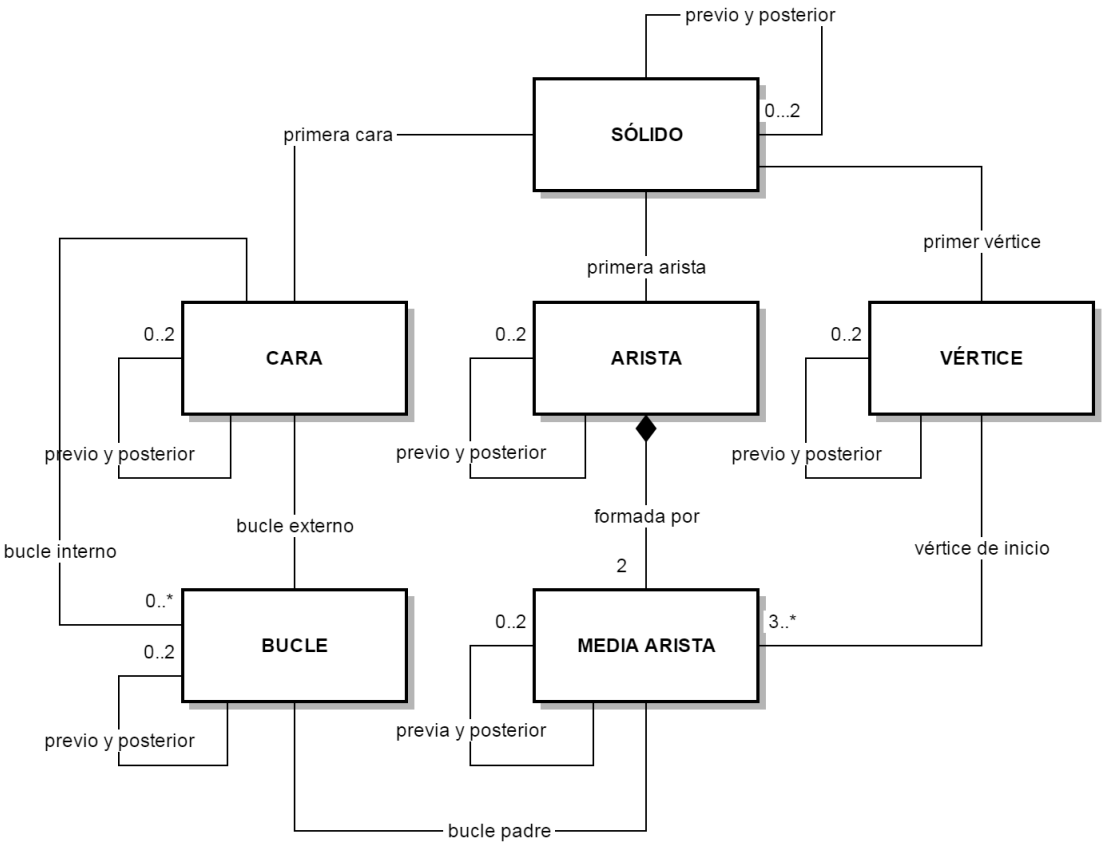


Figura 7. Diagrama de la representación de la Media Arista

Ahora veamos un ejemplo de representación en Media Arista y como se rellenarían sus tablas de un tetraedro.

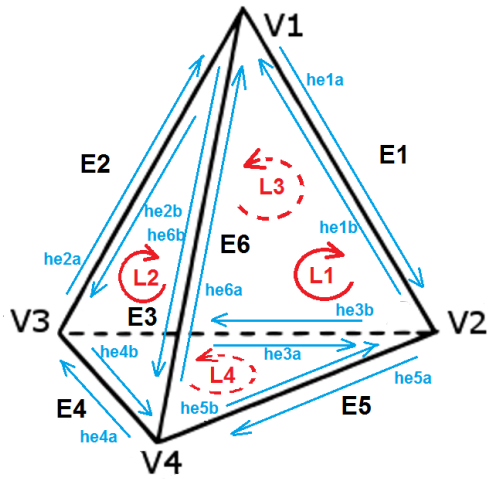


Figura 8. Ejemplo de representación con Media Arista (I) - tetraedro

SÓLIDO	1ª CARA	1ª ARISTA	1er VÉRTICE	SÓLIDO PREVIO	SÓLIDO POSTERIOR
S	F1	E1	V1	-	-

CARA	BUCLE EXTERNO	BUCLES INTERNOS	CARA PREV.	CARA POST.	SÓLIDO PADRE
F1	L1	-	F4	F2	S
F2	L2	.-	F1	F3	S
F3	L3	-	F2	F4	S
F4	L4	-	F3	F1	S

BUCLE	MEDIA ARISTA	BUCLE PREV.	BUCLE POST.	CARA MADRE
L1	he1a	L4	L2	F1
L2	he6b	L1	L3	F2
L3	he2b	L2	L4	F3
L4	he3b	L3	L1	F4

ARISTA	MED.ARISTAS	A.PREV.	A.POST.
E1	he1a, he1b	E6	E2
E2	he2a, he2b	E1	E3
E3	he3a, he3b	E2	E4
E4	he4a, he4b	E3	E5
E5	he5a, he5b	E4	E6
E6	he6a, he6b	E5	E1

VÉRTICE	X	Y	Z	V.PREV.	V.POST
V1	4	4	2	V4	V2
V2	0	0	0	V1	V3
V3	7	0	0	V2	V4
V4	4	0	7	V3	V1

MEDIA ARISTA	VÉRTICE INICIO	MEDIA ARISTA PREVIA	MEDIA ARISTA POSTERIOR	BUCLE PADRE
he1a	V1	he6a	he5a	L1
he1b	V2	he3a	he2b	L3
he2a	V3	he4a	he6b	L2
he2b	V1	he1b	he3a	L3
he3a	V3	he2b	he1b	L3
he3b	V2	he5b	he4b	L4
he4a	V4	he6b	he2a	L2
he4b	V3	he3b	he5b	L4
he5a	V2	he1a	he6a	L1
he5b	V4	he4b	he3b	L4
he6a	V4	he5a	he1a	L1
he6b	V1	he2a	he4a	L2

Figura 9. Ejemplo de representación con Media Arista (II) – tablas

2.2.3. Triángulo Alado (Winged Triangle)

Introducida por Paoluzzi en 1989 [10]. Es una representación por fronteras centrada en el vértice y cuya característica principal es la de usar única y explícitamente caras triangulares y vértices para definir un objeto. Las aristas se representan de forma implícita entre pares de vértices adyacentes.

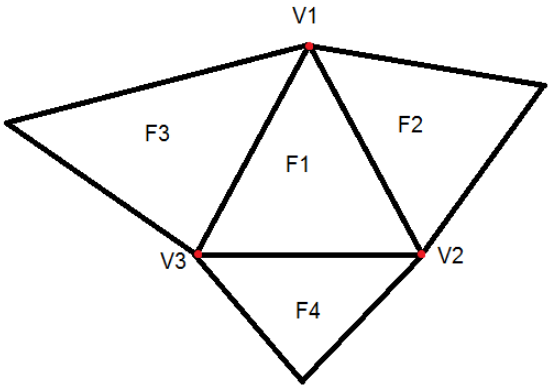


Figura 10. Estructura de un triángulo alado

Como se puede deducir, cada cara se ha de registrar dos triples tuplas: una almacenará los tres vértices que la delimitan y la otra las tres caras adyacentes a cada uno de sus lados. El Triángulo Alado admite objetos pseudo-múltiples, es decir, la representación múltiple de objetos no múltiples. La manera de hacerlo es mediante el uso de *shells*. Una carcasa o shell es una superficie 2-múltiple que encierra algo, que puede ser el propio objeto sólido o puede ser una "cámara vacía" su interior. Nótese que el sólido en sí mismo se cuenta como un shell.

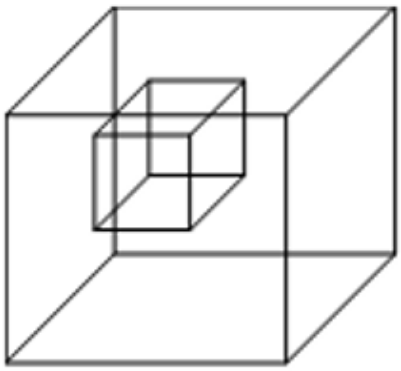


Figura 11. Objeto sólido formado por dos shells

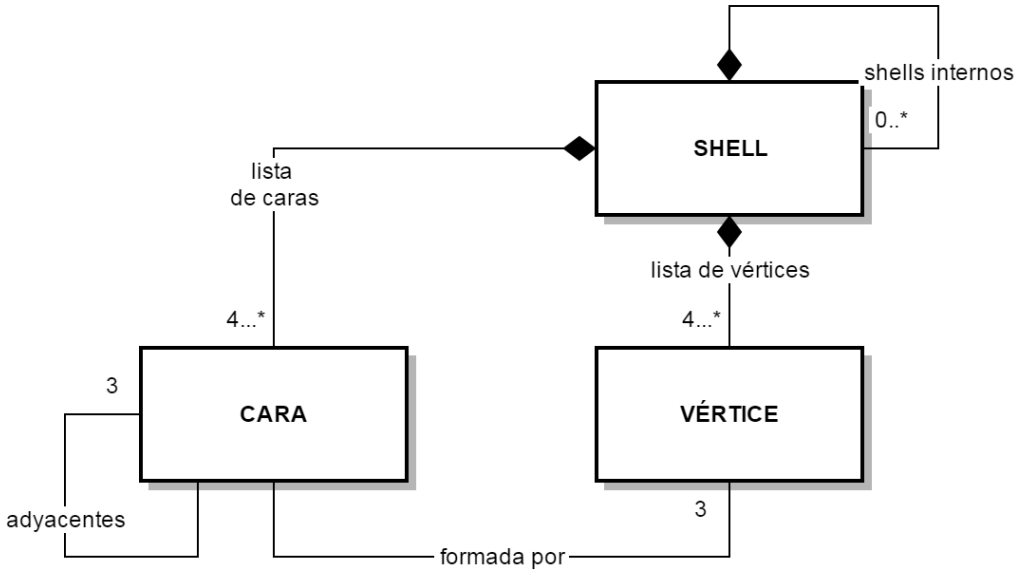


Figura 12. Diagrama de la representación del triángulo alado

Se necesitan, pues, tres tablas:

- **Tabla de shells:** Es una lista ordenada de shells en la que al menos debe existir una entrada correspondiente al propio objeto sólido. Contiene por cada shell una lista de caras y una lista de vértices.
- **Tabla de caras:** Cada entrada contiene los tres vértices que limitan la cara y sus tres caras adyacentes.
- **Tabla de vértices:** Cada entrada contiene las coordenadas 3D de un vértice.

Las caras contienen información interior y exterior, lo cual permite la representación de sólidos sin fronteras (volumen infinito), con superficies finitas.

Esta representación sólo permite poliedros lineales o aproximaciones lineales de poliedros con superficies curvas.

En el siguiente ejemplo se muestra cómo se rellenarían las tablas para representar un tetraedro.

SHELL	CARAS	VÉRTICES
SH1	F_SH1	V_SH1

F_SH1		
CARA	VÉRTICES	CARAS ADYACENTES
F1	V1, V3, V2	F2, F3, F4
F2	V1, V4, V3	F1, F3, F4
F3	V1, v2, v4	F1, F2, F4
F4	V2, v3, v4	F1, F2, F3

V_SH1			
VÉRTICE	X	Y	Z
V1	4	4	2
V2	0	0	0
V3	7	0	0
V4	4	0	7

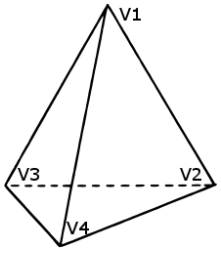


Figura 13. Ejemplo de representación con Triángulo Alado

2.2.4. Elemento Máximo (Maximal Element)

Los autores del Elemento Máximo son Krishnamurti (1992) [11] y Stouffs (1994) [12]. Centrada en el polígono o cara, esta representación es canónica y su topología se representa en su mayoría de forma implícita.

Su filosofía es: cualquier parte de una forma es también una forma. Los sólidos, caras, aristas, y vértices son definidos recursivamente por elementos de un orden de dimensión inferior. Por ello, el elemento básico en la representación:

- Es la cara (segmento de plano) para representar un sólido (segmento de volumen).
- Es la arista (segmento de línea) para representar un segmento de plano.
- Es el vértice (punto) para representar un segmento de línea.

En el Elemento Máximo, un sólido general consiste en uno o más segmentos de volumen disjuntos (máximos) que no comparten caras, pero pueden compartir aristas y/o vértices. Por tanto, admite sólidos no múltiples.

Las fronteras de un segmento de volumen se componen de un shell externo y de cero, uno o más shells internos, constituyendo agujeros en el sólido.

Cada Shell se compone de un conjunto de segmentos de plano (máximos), canónicamente ordenados de acuerdo con la ecuación de la cara. De manera similar, cada segmento de plano tiene una simple frontera externa (bucle externo), cero o más fronteras internas de segmentos de línea (máximos). Un segmento de línea está definido por dos puntos.

La geometría se representa en los vértices y parcialmente duplicado en las ecuaciones de caras y aristas.

Se necesitan 5 tablas:

- **Tabla de shells:** Es una lista ordenada de shells en la que cada entrada contiene una lista ordenada de caras.
- **Tabla de caras:** Cada cara tiene una lista ordenada de fronteras o bucles.
- **Tabla de bucles:** Cada bucle corresponde a una lista ordenada de aristas.
- **Tabla de aristas:** Cada arista hace referencia a dos vértices que la delimitan.
- **Tabla de vértices:** Cada vértice son unas coordenadas 3D.

Antes hemos dicho que se definen de manera implícita las aristas y los vértices. Dicha información se obtiene de las intersecciones entre sí de los planos que definen las caras del objeto sólido.

La representación del Elemento Máximo es fundamental para gramáticas de formas. Existen diferentes formalismos de gramáticas que incluyen información no geométrica tal como color y grosor de línea. Información no gráfica como etiquetas u otros tipos de pesos podría incluirse también.

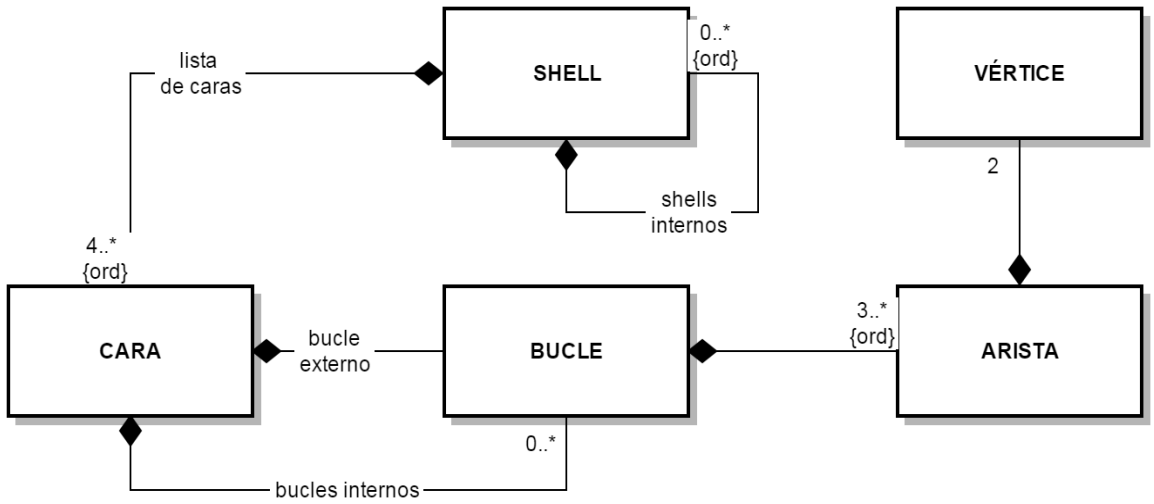


Figura 14. Diagrama de la representación del Elemento Máximo

Veamos cómo sería un ejemplo de tablas con la representación de un tetraedro con la técnica del Elemento Máximo.

SHELL	CARAS
SH1	F1, F2, F3, F4

CARA	BUCLES
F1	B1
F2	B2
F3	B3
F4	B4

ARISTA	VERTICES
a	V1, V2
b	V2, V3
c	V3, V1
d	V3, V4
e	V4, V1
f	V2, V4

BUCLE	ARISTAS
B1	a, c, b
B2	c, e, d
B3	b, d, f
B4	a, f, e

VERTICE	X	Y	Z
V1	4	4	2
V2	0	0	0
V3	7	0	0
V4	4	0	7

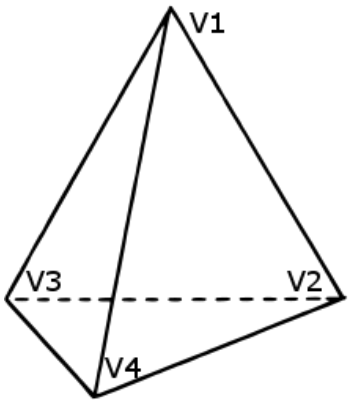


Figura 15. Ejemplo de representación con Elemento Máximo

2.3. El formato STL

Del inglés, STereo Lithography [6] [7], define la geometría de objetos 3D sin información de color, textura o propiedades físicas. Creado por la empresa 3D Systems para prototipado rápido, actualmente es muy apreciado para su uso con impresoras 3D. También es conocido como Standard Tesselation Language. Los archivos tienen típicamente la extensión `.stl`, aunque también puede ser `.sta`. Suelen escribirse en representación ASCII y también en binario, para ficheros más compactos.

STL describe una superficie geométrica tridimensional. La superficie es teselada o dividida de manera lógica en una serie de facetas triangulares. Cada faceta se describe con la normal unitaria, que indica su orientación, y los vértices convenientemente ordenados siguiendo la regla de la mano derecha. Usa un sistema cartesiano de coordenadas positivas. Sin información de escala, las unidades son arbitrarias.

2.3.1. Especificaciones de STL. Forma correcta de uso.

Todo sólido STL debe cumplir la regla vértice-a-vértice. Esto consiste en que cada triángulo debe compartir 2 vértices con cada uno de sus triángulos adyacentes. Es decir, el vértice de un triángulo no puede descansar en el lado de otro.

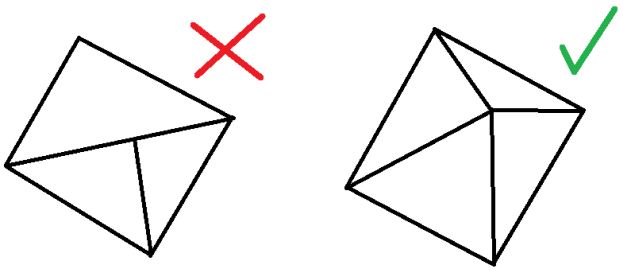


Figura 16. Regla vértice-a-vértice

Además, para formar correctamente un volumen sólido 3D, la superficie representada por cualquier archivo STL debe ser cerrada y conectada, donde cada arista forma parte exactamente de 2 triángulos que no hacen intersección entre sí.

Como la sintaxis STL no fuerza esta propiedad, las aplicaciones en las que el cierre no importa pueden ignorarlo. Esta sólo importa en la medida que el software que corta los triángulos requiera para asegurarse de que los polígonos 2D resultantes estén cerrados.

A veces, software así puede escribirse para limpiar pequeñas discrepancias moviendo vértices que están muy cerca uno del otro para que coincidan. Los resultados no son predecibles, pero a menudo eso es suficiente.

La normal de cada cara es un vector unitario que apunta hacia el exterior del objeto sólido. El formato de archivo STL puede ser tanto en ASCII como en binario. Hay que prestar atención al hecho de que en la mayoría del software se pone a (0,0,0) y luego es el propio software el que calcula automáticamente una normal basada en el orden de los vértices del triángulo, los cuales, siguiendo la regla de la mano derecha, deben estar listados en orden contrario al sentido de las agujas del reloj (CCW).

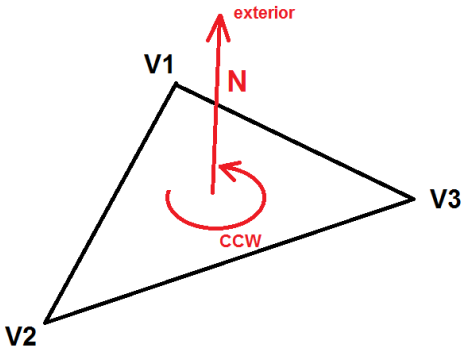


Figura 17. Faceta triangular de STL

Así pues, un consejo para que un archivo sea completamente portable es proporcionar obligatoriamente la normal de la cara y ordenar apropiadamente los vértices. Es recomendable hacerlo ascendentemente, según el valor de Z, para optimizar un programa de laminado (*slicing*). Hay una excepción a esto. SolidWorks, que usa la normal para efectos de sombreado (*shading*).

2.3.2. Archivo ASCII STL

En ASCII el archivo comienza siempre por `solid` y el nombre del sólido a representar. A continuación se describe cada una de sus facetas. El archivo debe terminar con `endsolid` y el nombre del sólido. NOTA: `n` y `v` son en coma flotante y positivos.

```
solid nombre

facet normal ni nj nk
  outer loop
    vertex v1x v1y v1z
    vertex v2x v2y v2z
    vertex v3x v3y v3z
  endloop
endfacet

endsolid nombre
```

2.3.3. Archivo binario STL

La cabecera es de 80 caracteres. Le sigue un número entero sin signo de 4 bytes que indica el número de facetas triangulares en el archivo. A continuación se describe los triángulos del sólido. El archivo finaliza tras la descripción del último triángulo. Para cada triángulo se emplean 12 valores en coma flotante (4 bytes, cada valor): 3 para la normal y 3 para las coordenadas XYZ de cada vértice.

Tras esto, sigue un entero sin signo de 2 bytes (short) que es el número de bytes que se utiliza para el color. En el formato estándar este debería ser cero, ya que la mayoría del software no entiende nada más.

Los números en coma flotante están representados como números en coma flotante IEEE y se asume que son Little-endian, aunque no se especifica en la documentación.

```
UINT8[80]  -Cabecera
UINT32     -Número de triángulos

Por cada triángulo:
REAL32[3]  -Vector normal
REAL32[3]  -Vértice 1
REAL32[3]  -Vértice 2
REAL32[3]  -Vértice 3
UINT16     -Número de bytes utilizados para el color
```

2.3.4. Color en STL binario

Aunque no es estándar, hay dos variaciones de STL para añadir información de color:

Los paquetes de software de VisCAM y SolidView usan atributos de número de bytes para el color al final de cada triángulo para almacenar un color RGB de 15 bits:

- Bits de 0 a 4 son el nivel de intensidad de azul (0 a 31)
- Bits de 5 a 9 son nivel de intensidad de verde (0 a 31)
- Bits de 10 a 14 son el nivel de intensidad de rojo (0 a 31)
- El bit 15 es 1 si el color es válido, o 0 si el color no es válido (como en los archivos STL normales)

El software Materialise Magics usa una cabecera de 80 bytes al principio del archivo para representar el color de todo el objeto. Para un color diferente en cada cara habría que sobrescribir cada vez la cabecera. La cabecera debe contener la cadena `COLOR=` seguida de 4 bytes que representan los canales rojo, verde, azul y alfa (transparencia) con un rango de 0 a 255. Magic también “reconoce” el material, de manera que si a continuación de `COLOR=RGBA` se pone `MATERIAL=` seguido de otros tres colores de 4 bytes, estaremos indicando con el primer color la reflexión difusa, con el segundo la luz especular y con el tercero, la luz ambiental.

2.4. Tabla resumen comparativa de BRep y STL

Una vez presentados BRep y STL, a continuación se muestra desplegada en dos páginas una tabla que resume y compara las características de todas las técnicas vistas.

	WINGED EDGE	HALF EDGE
AUTOR	Baumgart (1972)	Mäntylä (1988)
ELEMENTO PRINCIPAL	Arista	Arista
ELEMENTOS DEFINIDOS EXPLÍCITAMENTE	Cara, arista y vértice	Sólido, cara, bucle, arista, media arista y vértice
ELEMENTOS DEFINIDOS IMPLÍCITAMENTE	ninguno	ninguno
ESTRUCTURA TOPOLÓGICA	Representación completa	Representación completa
ESTRUCTURA DEL ELEMENTO PRINCIPAL	Cada arista se define con: + Una orientación + 2 vértices + 2 caras de las que forma parte	Cada arista se define con: + 2 medias aristas + aristas anterior y posterior
TOPOLOGÍA	**Sólido:** Lista de caras, aristas y vértices **Cara:** Una de las aristas que la forman Lista ordenada de aristas (bucle CCW) **Arista:** * + Orientación + 2 vértices incidentes + 2 caras adyacentes + 4 aristas vecinas: 2 precedentes y 2 sucesoras **Vértice:** Una de sus aristas incidentes	**Sólido:** Primer elemento en la lista de caras, aristas y vértices y sólido anterior y posterior en la lista de sólidos. **Cara:** bucle externo, lista de bucles internos, cara anterior y posterior en la lista de caras y sólido padre. **Bucle:** Una media arista de la frontera, bucle anterior y posterior en la lista de bucles y cara madre. **Arista:** Dos medias aristas y aristas previa y posterior en la lista de aristas. **Media arista:** Vértice de inicio de la arista en la dirección del bucle, media arista anterior y posterior en la lista y bucle padre. **Vértice:** Vértice anterior y posterior en la lista de vértices, media arista madre.
GEOMETRÍA	**Cara:** Normal que apunta al exterior **Vértice:** Coordenadas 3D	**Cara:** Ecuación de un plano **Vértice:** Coordenadas 3D
TIPOS DE SÓLIDO REPRESENTABLES (DOMINIO)	+ **Regulares:** Cuerpos 3D que no tienen vértices, aristas o caras sueltas o aisladas. + **Múltiples**: Cada arista es adyacente a exactamente 2 caras + **Superficies conectadas:** cuerpo con sólo un *shell* + **Caras sin agujeros**: cada cara permite un simple bucle de aristas	+ **Regulares** + **Pseudo-múltiples:** Representación múltiple de poliedros no múltiples. Cada arista tiene exactamente 2 medias aristas y forma parte de 2 caras. Sin embargo, una arista puede coincidir con otro vértice, una arista o una cara. + **Superficies conectadas**
OBSERVACIONES	Existen **dos extensiones** que permiten sólidos con agujeros: + **Braid (1980):** En la cara del agujero hay un bucle principal en CW y uno secundario o interno en CCW. + **Yamaguchi (1985):** Usa una arista auxiliar. 1 bucle CCW *La arista se analiza vista desde fuera del objeto.	+ Puede derivar en el Winged Triangle de Paoluzzi, pero no subsumirlo. + Extiende la representación Winged Edge de Baumgart para poder representar sólidos no múltiples.

WINGED TRIANGLE	MAXIMAL ELEMENT	STL
Paoluzzi (1989)	Krishnamurti(1992) & Stouffs(1994)	3DSystems (1986)
Vértice	Cara o polígono	Vértice
Cara y vértice	Segmento de volumen(sólido) Shell Segmento de plano (cara) Frontera (bucle)	Cara y vértice
Arista (pares de vértices adyacentes)	Segmento de línea (arista) Punto fronterizo (vértice)	Arista y normal (si no se especifica)
Representación parcial	Representación parcial	Representación parcial
Cada cara se define con: + 3 vértices + 3 caras adyacentes	Cara → para representar un sólido Arista→ para representar una cara Vértice→ para representar una arista (Ver "Observaciones" *)	Cada cara se define por: + 3 vértices ordenados + Vector normal
Sólido: Lista ordenada de shells. Mínimo 1 (un poliedro es un Shell en sí) **Shell:** Lista de caras y vértices **Cara:** Dos triples tuplas: + 3 vértices + 3 caras adyacentes	**Sólido:** Lista ordenada de shells **Shell:** Lista ordenada de caras **Cara:** Lista ordenada de bucles **Bucle:** Lista ordenada de aristas **Arista:** 2 vértices	**Sólido:** Lista de caras únicas **Cara:** + 3 vértices ordenados en sentido antihorario (CCW) visto desde fuera. Esto es, usando la regla de la mano derecha. + 1 vector normal. (Ver sección "Observaciones")
Cara: Vector normal apuntando al exterior **Vértice**: Coordenadas 3D	**Cara** Ecuación de un plano **Arista:** Ecuación de una recta **Vértice:** Coordenadas 3D	**Cara:** Vector normal apuntando al exterior. **Normal y vértice:** Coord. 3D
+ Regulares **+ Pseudo-múltiples:** (multishell) Representación múltiple de poliedros no múltiples **+ Volúmees infinitos con superficie finita** **+ Poliedros lineales** o representaciones linealizadas de superficies curvas **+ Caras sin agujeros**	+ **No regulares:** Formados por más de un objeto con dimensionalidades diferentes (sólido, cara, arista o vértice). Las operaciones son regulares dentro de su dimensionalidad + **Intrínsicamente no múltiple** + **Agujeros:** Shells múltiples(o fronterizos) para un simple sólido (o cara), de los cuales, uno es externo y otro interno.	+ **Regulares** + **Múltiples** + **Superficies conectadas** + **Caras sin agujeros**
Posiblemente canónica: + Un poliedro tiene un número máximo de shells. + Triangulación canónica de caras basada en vértices ordenados + Lista ordenada de Shells, caras y vértices.	Representación canónica: + Máximos segmentos de volumen, de plano y de línea + Listas ordenadas de elementos. *Sólidos, aristas y vértices son definidos recursivamente por sus fronteras de una dimensionalidad inferior. (ver estructura del elemento principal)	+Cada arista es parte de dos caras que no se intersectan. + En la mayoría de software, la normal se pone a (0,0,0) y se calcula automáticamente la normal basada en el orden de los vértices usando la regla de "la mano derecha".

2.5. Análisis de costes de las técnicas BRep

Para llevar a cabo este análisis aproximado, se tomará un volumen sólido general, con cualquier número de caras, aristas y vértices. El objetivo es conocer cuál es la técnica de representación BRep más adecuada para el proyecto teniendo en cuenta sus características y costes de memoria.

Se considerará que:

- Es despreciable el coste de almacenar el nombre de cada elemento .
- Las coordenadas tridimensionales de los vértices son datos tipo float.
- Las referencias a otras tablas son de tipo short int.
- El coste del tipo short int es de 2 bytes.
- El coste del tipo float es de 4 bytes.

Nombre	Descripción
S	sólido
SH	número de shells
F	número de caras
L	bucle de aristas
E	número de aristas
HE	media arista
V	número de vértices
nLint	número de bucles o agujeros

Figura 18. Nomenclatura de variables en el cálculo de costes de BRep

2.5.1. Costes con la Arista Alada (Winged Edge)

Un sólido necesita referenciar a tres tablas: caras, aristas y vértices. Cada cara está formada por bucles externos e internos de aristas (Arista Alada de Braid).

$$FaceList =< \left(L_{ext}^{F}, L_{int}^{nLint}\right) \geq \begin{cases} \sum_{f=1}^{f=F} 2 * E_f & , \ nLint = 0 \\ \sum_{f=1}^{f=F} \left(2 * E_f + \sum_{b=1}^{b=nLint_i} 2 * E_b\right), & nLint > 0 \end{cases}$$

$$EdgeList =< (v1, v2, f1, f2, e1, e2, e3, e4)^{E} \geq 2 * 8 * E = 16 * E \ bytes$$

$$VertexList =< (x, y, z)^{V} \geq 3 * 4 * V = 12 * V \ bytes$$

$$\boldsymbol{Total\ coste\ WE} = S =$$

$$< (FaceList), (EdgeList), (VertexList) >= \boldsymbol{FaceList + EdgeList + VertexList\ bytes}$$

2.5.2. Costes con la Media Arista (Half Edge)

En esta representación un sólido forma parte de una lista de sólidos y debe hacer referencia a sus sólidos precedente y sucesor en dicha lista. Debido a que el objetivo de este análisis se va a centrar en un solo sólido, se despreciará el coste de referencia antes citado. Por otro lado, para definirlo, se necesita las listas de caras, bucles, aristas, medias aristas y vértices.

$$FaceList =< \left(L_{ext}, L_{int}^{nLint}, f_{prev}, f_{next}\right)^{F} > = \begin{cases} 6 * F \ bytes & , \quad nLint = 0 \\ \sum_{f=1}^{f=F} \left(2 * nLint_f + 6\right) bytes, & nLint > 0 \end{cases}$$

$$LoopList =< \left(he_1, L_{prev}, L_{next}, f_{parent}\right)^{(F+nLint)} >= 8 * (F + nLint) \ bytes$$

$$EdgeList =< \left(he_1, he_2, e_{prev}, e_{next}\right)^{E} >= \ 2 * 4 * E = 8 * E \ bytes$$

$$HalfEdgeList =< \left(v_1, he_{prev}, he_{next}, L_{parent}\right)^{(2*E)} >= 2 * 4 * 2 * E = 16 * E \ bytes$$

$$VertexList =< \left(x, y, z, v_{prev}, v_{next}, he_{parent}\right)^{V} >= (4 * 3 + 2 * 3) * V = 18 * V \ bytes$$

$$S =< (f_1, e_1, v_1) >= 2 * 3 = 6 \ bytes$$

$$\boldsymbol{Total\ coste\ H = S + FaceList + LoopList + EdgeList + HalfEdgeList + VertexList\ bytes}$$

2.5.3. Costes con el Triángulo Alado (Winged Triangle)

Para esta representación se necesita:

- Lista de shells.
- Por cada shell, una lista de caras y una lista de vértices.

$$FaceList =< \left(v_1, v_2, v_3, f_{ady1}, f_{ady2}, f_{ady3}\right)^{F} >= 2 * 6 * F = 12 * F \ bytes$$

$$VertexList =< (x, y, z)^{V} >= 4 * 3 * V = 12 * V \ bytes$$

$$ShellList =< \left((FaceList), (VertexList)\right)^{SH} >= \sum_{sh=1}^{sh=SH} 2 * 2 = 4 * SH \ bytes$$

$$\boldsymbol{Total\ coste\ WT} = S =< (ShellList) >= \sum_{\boldsymbol{sh=1}}^{\boldsymbol{sh=SH}} \boldsymbol{4 + 12 * (F_{sh} + V_{sh})\ bytes}$$

2.5.4. Costes con el Elemento Máximo (Maximal Element)

Para esta representación se necesita:

- Lista de shells.
- Por cada shell, lista de caras.

- Por cada cara, lista de bucles (externo e internos).
- Por cada bucle, lista de aristas. Nótese que cada arista forma forzosamente parte de 2 bucles. El número total de bucles es la suma de bucles externos mas internos, es decir, F+nLint.
- Por cada arista, una tupla de dos vértices
- Por cada vértice, sus coordenadas 3D.

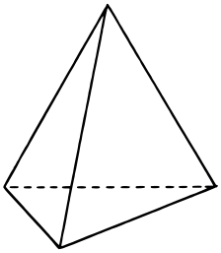
$$VertexList =< (x,y,z)^{V} >= 4*3*V = 12*V\ bytes$$
$$EdgeList =< (v_1,v_2)^{E} >= 2*2*E = 4*E\ bytes$$
$$LoopList =< (e)^{(2*(F+nLint))} >= 2*1*\big(2*(F+nLint)\big) = 4*(F+nLint)\ bytes$$
$$FaceList =< (l)^{(F+nLint)} >= 2*1*(F+nLint) = 2*(F+nLint)\ bytes$$
$$ShellList =< (f)^{F_{sh}} >= \sum_{sh=1}^{sh=SH} 2*1*F_{sh}\ bytes$$
$$\boldsymbol{Total\ coste\ ME} = S = < (ShellList), ((FaceList), (LoopList), (EdgeList), (VertexList))^{SH} >$$
$$= \sum_{sh=1}^{sh=SH} (2*F_{sh} + 2*(F_{sh}+nLint_{sh}) + 4*(F_{sh}+nLint_{sh}) + 4*E_{sh} + 12*V_{sh})$$
$$= \sum_{\boldsymbol{sh=1}}^{\boldsymbol{sh=SH}} (\boldsymbol{2*F_{sh} + 6*(F_{sh}+nLint_{sh}) + 4*E_{sh} + 12*V_{sh})\ bytes}$$

2.6. Comparativa BRep con sólidos de prueba

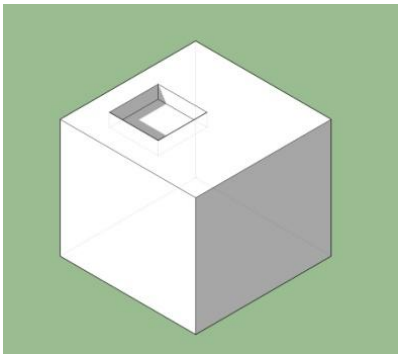
Una vez analizados los costes sobre un volumen sólido general, para compararlos, se han aplicado los cálculos a diferentes tipos de sólido, intentando cubrir el mayor número de casos posible: mínimo representable (tetraedro), sólido con caras de más de un bucle, sólido con un agujero, sólido con un agujero y más de un shell y sólido con más de un agujero. [8]

En las siguientes tablas se presentan los datos que caracterizan a cada sólido. Estos pueden variar dependiendo de la representación:

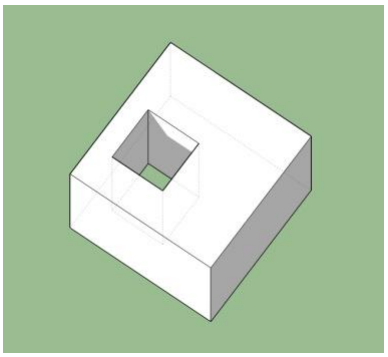
- El número de caras es mayor cuando el sólido se representa con el Triángulo Alado (Winged Triangle), ya que hay que cumplir la restricción de que las caras sean triangulares.
- El sólido básico 3 tiene dos shells. En la tabla, se muestra en formato de suma el número de caras, aristas y vértices de cada uno de los shells. En las representaciones de la Arista Alada (Winged Edge) y Media Arista (Half Edge), cada Shell se toma como un sólido independiente. El coste en estos dos casos será la suma de costes de cada sólido.
- Los datos innecesarios para el cálculo de costes de cada representación se omiten en las tablas y se muestra el símbolo "-" en su lugar.



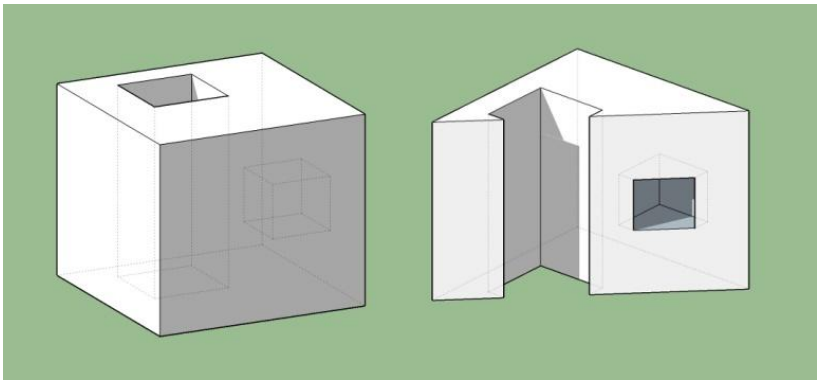
Sólido mínimo o tetraedro



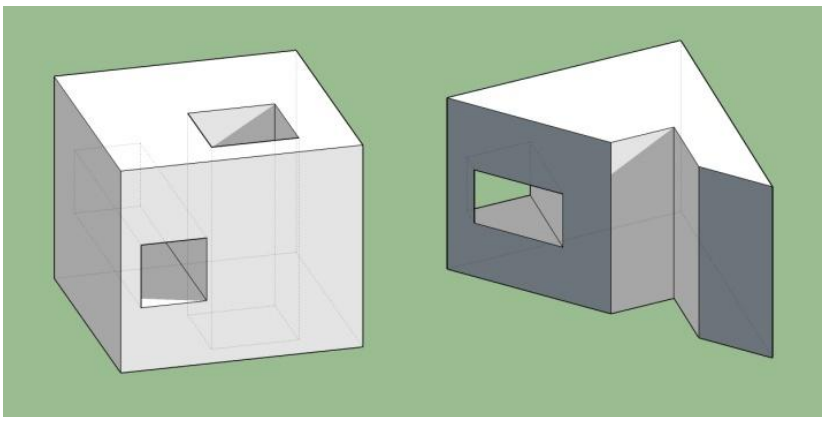
Básico 1: sólido con caras de más de un bucle



Básico 2: sólido con un agujero



Básico 3 sin seccionar y seccionado: sólido con un agujero y más de un shell



Básico 4 sin seccionar y seccionado: sólido con más de un agujero

Figura 19. Sólidos de prueba usados en el análisis de costes de BRep

TÉCNICA	Sólidos S	Shells SH	Caras F	Aristas E	Vértices V	Bucles internos nLint
Winged Edge	1	-	4	6	4	0
Half Edge	1	-	4	6	4	0
Winged Triangle	1	1	4	-	4	-
Maximal Element	1	1	4	6	-	-

Figura 20. Características sólido mínimo

TÉCNICA	Sólidos S	Shells SH	Caras F	Aristas E	Vértices V	Bucles internos nLint
Winged Edge	1	-	11	24	16	1
Half Edge	1	-	11	24	16	1
Winged Triangle	1	1	28	-	16	-
Maximal Element	1	1	11	24	-	-

Figura 21. Características sólido básico 1

TÉCNICA	Sólidos S	Shells SH	Caras F	Aristas E	Vértices V	Bucles internos nLint
Winged Edge	1	-	10	24	16	2
Half Edge	1	-	10	24	16	2
Winged Triangle	1	1	32	-	16	-
Maximal Element	1	1	10	24	-	-

Figura 22. Características sólido básico 2

TÉCNICA	Sólidos S	Shells SH	Caras F	Aristas E	Vértices V	Bucles internos nLint
Winged Edge	2	-	10+6	24+12	16+8	2+0
Half Edge	2	-	10+6	24+12	16+8	2+0
Winged Triangle	1	2	32+12	-	16+8	-
Maximal Element	1	2	10+6	24+12	-	-

Figura 23. Características sólido básico 3

TÉCNICA	Sólidos S	Shells SH	Caras F	Aristas E	Vértices V	Bucles internos nLint
Winged Edge	1	-	14	36	24	4
Half Edge	1	-	14	36	24	4
Winged Triangle	1	1	50	-	24	-
Maximal Element	1	1	14	36	-	-

Figura 24. Características sólido básico 4

SÓLIDO DE PRUEBA	Coste Winged Edge	Coste Half Edge	Coste Winged Triangle	Coste Maximal Element
Mínimo	236 bytes	278 bytes	**100 bytes**	104 bytes
Básico 1	672 bytes	1034 bytes	532 bytes	**382 bytes**
Básico 2	672 bytes	1030 bytes	580 bytes	**380 bytes**
Básico 3	1008 bytes	1552 bytes	824 bytes	**572 bytes**
Básico 4	1008 bytes	1538 bytes	892 bytes	**568 bytes**

Figura 25. Resultados de costes de cada técnica sobre los sólidos de prueba

2.7. Conclusiones de cara al proyecto

Como se puede observar en la tabla de la Figura 25, en casi todos los casos la representación que necesita menos memoria es la del Elemento Máximo, dejando en segundo lugar al Triángulo Alado, en tercer lugar a la Arista Alada y en el último puesto a la Media Arista.

Para el objetivo del conversor, aunque la técnica del Triángulo Alado tiene gran similitud a STL en el empleo de caras triangulares, además de tener un coste de memoria menos elevado que otras técnicas BRep, es, sin embargo, desde el punto de vista del usuario, y sobretodo del alumno, realmente tedioso tener que definir cada una de las facetas de un objeto. Por ello, la Arista Alada sigue siendo la más amigable para comprender la representación por fronteras.

3. Desarrollo del conversor

En este capítulo se hablará de los requisitos del conversor y de la tecnología OpenSceneGraph con la que se desarrolla el proyecto. También se analizará el problema de la triangulación de un objeto 3D y se abordará con diferentes soluciones hasta llegar a la solución más acertada. Se describirá la estructura de elementos del proyecto a nivel de implementación y se finalizará el capítulo hablando de cómo ha ido el desarrollo según lo previsto y de los resultados obtenidos.

3.1. Especificación de requisitos

En las asignaturas relacionadas con el diseño y fabricación asistidos por computador, interesa entender el uso diferentes formatos de modelado 3D. Por la parte del diseño, la representación por fronteras (BRep) y por el lado de la fabricación, especialmente en lo referente a la impresión en 3D, el formato STL.

Como ya se dijo anteriormente, el programa desarrollado en el proyecto final de carrera "Visor BREP. Un programa para visualizar modelos 3D" [2], presenta algunos problemas de compatibilidad con cada nueva versión de Java que se publica, de manera que no garantiza el correcto funcionamiento, especialmente del visor que lleva incorporado, impidiendo al alumnado aprender más intuitivamente como modelar objetos con BRep.

Para solventar este problema, se requiere crear un nuevo programa que permita visualizar el objeto que se defina en Brep conviertiendo los datos del archivo Brep en datos de archivo STL. Al ser STL un lenguaje de modelado estándar, es fácil visualizar su contenido en cualquier visor que admita STL.

Por otro lado, para minimizar la posibilidad de incompatibilidades, el conversor ha de funcionar en el mayor número de plataformas diferentes posible.

Por tanto, los requisitos mínimos que interesa cumplir este conversor son:

- Lectura de archivo de extensión `.brp`, conversión de sus facetas a sólo facetas triangulares y escritura en archivo de extensión `.stl`
- Multiplataforma
- Fácilmente ampliable para futuras funcionalidades

Otros requisitos más secundarios pero deseables:

- Visor incorporado para comprobar directamente el resultado de la lectura.
- Visualizar también los colores descritos en el `.brp` si se diera el caso.

En el capítulo siguiente se hablará de la tecnología de desarrollo escogida: OpenSceneGraph y en capítulos posteriores de diferentes opciones para triangular las facetas y cuál ha sido la más apropiada para este proyecto.

3.2. Conociendo OpenSceneGraph

OpenSceneGraph (OSG) [9], es uno de los mejores conjuntos de herramientas para la representación de gráficos 3D. Es de código abierto y está diseñado para ejecutarse en diferentes sistemas operativos y en la mayoría de computadores, incluso en plataformas móviles con OpenGL ES instalado, el cual es un subconjunto de APIs de gráficos 3D de OpenGL. OSG ha sido diseñado para aprovechar las ventajas de los multiprocesadores y de las arquitecturas multinúcleo y, por tanto, funciona perfectamente con procesadores de 32 y 64 bits.

El núcleo de funcionalidad de OSG tiene 4 librerías:

OpenThreads: Interfaz de hilos orientado a objetos. Se usa para la implementación del hilo principal.
Osg: Contiene lo necesario para construir grafos de escenas, en 2D y en 3D.
osgDB: Necesaria para la lectura y escritura de archivos 2D y 3D.
osgUtil: Necesaria para construir *backends*.

OSG tiene una gran lista de *nodekits* y utilidades. Algunas de las cuales se usan en este proyecto y se citan a continuación:

osgGa: Soporta la abstracción de OSG GUI. Ayuda a manejar eventos interactivos desde dispositivos periféricos.
osgViewer: Usado para visualización de gráficos.

3.2.1. En resumen: ¿Por qué escoger OSG?

Las razones que llevan a usar OpenSceneGraph son varias:

Estructura rigurosa: Hace uso de la Standard Template Library [10] para C++ y de múltiples patrones de diseño.

Desarrollo superior: Técnicas de grafos de escena ya bien implementados, como nivel de detalle (LOD), sombras, partículas y un completo encapsulado de extensiones de OpenGL, entre otras.

Alta escalabilidad: Las funcionalidades del núcleo de OSG son limpias y altamente extensibles. Para el usuario es fácil escribir sus propios *Nodekits* y plugins de lectura y escritura e integrarlos en grafos de escena y aplicaciones.

Portabilidad Hardware y Software: El núcleo de OSG está diseñado para tener una mínima dependencia de cualquier plataforma. Sólo requiere Standard C++ y OpenGL. De esta manera, es posible portar rápidamente aplicaciones basadas en OSG a Windows, Linux, Mac OSX, FreeBSD, Solaris e incluso plataformas integradas.

A la última: OSG siempre está actualizado porque tiene una comunidad haciéndolo crecer y evolucionar a una velocidad increíble.

Open source: En la industria moderna, el código abierto significa co-inteligencia, calidad y flexibilidad, más que meramente económico. Los usuarios y compañías no deben preocuparse por las violaciones de patente cuando usen OSG en sus aplicaciones.

Una de las cosas más interesantes para este proyecto, es la de poder crear plugins de lectura y escritura personalizados. En este caso, como mínimo necesitamos que el plugin fuese ser capaz de reconocer y leer archivos de extensión `.brp` para poder manipular sus datos. Un plugin de BRep completo, además, podría escribir archivos de formato BRep , lo cual es altamente adecuado como ampliación de las funciones del conversor en un futuro próximo.

3.2.2. Estructura de un plugin de OSG

Un plugin en OSG es un componente de funcionalidad separada que personaliza los formatos de archivo soportados por una aplicación basada en OSG. Se reconoce como un archivo de librería compartida que implementa la interfaz de lectura, la de escritura, o ambas. Siempre se requieren diferentes plugins en las aplicaciones de usuario para cargar o construir grandes y complejos grafos de escena sin tener que hacer mucho trabajo de programación.

Todo plugin sigue la misma convención de nomenclatura; de otro modo, no serían reconocibles y no podrían usarse para leer archivos. Sin embargo, un mismo plugin puede soportar múltiples extensiones. Por ejemplo, el formato de imagen JPEG usa `.jpeg` y `.jpg` como extensiones de archivo más comunes.

En el conversor, las extensiones de archivo implicadas son `.brp`, para la cual es necesario escribir un plugin que lo reconozca, y la extensión `.stl`, para el que, OSG tiene un plugin de lectura y escritura ya preparado.

Para implementar un plugin personalizado hay que seguir estos pasos:

1- En la declaración de la clase del nuevo plugin, por ejemplo `class ReaderWriterBRP`, Implementar los métodos de lectura para el nuevo formato, sobrescribiendo dos métodos virtuales de `ReadNode()`: uno para leer datos desde un archivo y el otro para leer datos desde *streams*.

2- En el constructor, declarar que la nueva extensión es soportada por este plugin. Por ejemplo, para el caso de BREP sería así:

```
ReaderWriterBRP::ReaderWriterBRP(){
 supportsExtension("brp", "Boundary Representation");}
```

3- Implementar el método `ReadNode()` para leer archivos desde el disco. Comprobará si la extensión y el nombre de archivos están disponibles y tratará de redirigir el contenido del archivo a `std::fstream` para futuras operaciones.

4- Implementar el método `ReadNode()` para leer archivos desde *streams*. Aquí se sitúa el núcleo de implementación del nuevo formato. Con los datos leídos del *stream*, se crea un nuevo objeto de tipo osg::Geometry que contendrá los objetos primitivos. Finalmente, genera las normales de la geometría y devuelve un nodo osg::Geode, que contiene el resultado de la lectura.

5- Registrar la clase del plugin de lectura/escritura con el macro siguiente. Hay que hacerlo al final del código fuente del plugin. El primer parámetro indica el nombre de la librería del plugin (sin prefijo `osgdb_`) y el segundo parámetro proporciona el nombre de la clase. Este paso es necesario para que OSG sea capaz de leer el nuevo formato.

Ejemplo: `REGISTER_OSGPLUGIN( brp, ReaderWriterBRP)`

Nótese que el nombre objetivo de salida debería ser `osgdb_brp`, y que debería ser una librería compartida en lugar de un fichero ejecutable. Así que, para añadir la nueva librería al proyecto de compilación de OSG, el script de CMake tendría que usar la macro `add_library( osgdb_brp SHARED readerwriter.cpp)`.

No obstante, el lector puede encontrar más información y un ejemplo completo de implementación en la referencia [9], capítulo 10, "Writing your own plugins".

3.3. El problema de la triangulación

Una vez superada la primera parte del proyecto, que es la de lograr que el conversor lea datos de un archivo de extensión .brp, el núcleo de implementación del proyecto, reside en "traducir" las caras poligonales de 3 o más vértices que puede tener un objeto sólido BRep a facetas triangulares que describan un objeto sólido 3D equivalente en STL. Es decir, es necesario triangular las caras.

¿Pero, qué es la triangulación? Es el proceso de crear una malla de triángulos a partir de un conjunto de puntos. Hay diferentes algoritmos de triangulación de polígonos que se usan para hacer teselados de geometrías y para otros muchos propósitos.

A la hora de triangular hay que en cuenta si los polígonos son cóncavos o convexos.

Un polígono convexo es aquel donde no hay ángulos internos de más de 180º. En este caso, es muy sencillo descomponer el polígono en triángulos.

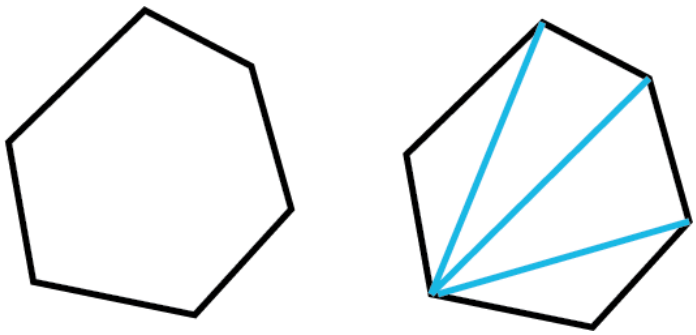


Figura 26. Polígono convexo y su triangulación

Un polígono cóncavo es aquel que tiene al menos un ángulo de más de 180º. Una característica de los polígonos cóncavos es que se pueda dibujar al menos una línea que intersecte al polígono 4 veces como mínimo. En este caso, una correcta triangulación depende del algoritmo que se emplee.

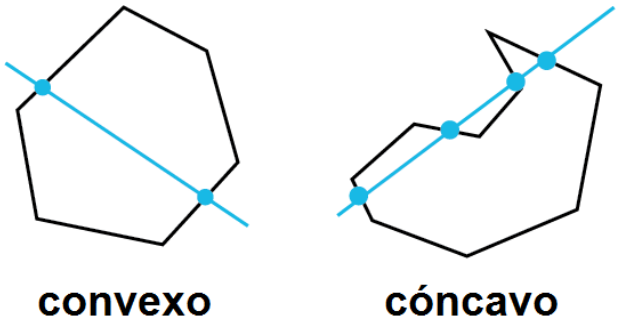


Figura 27. Polígono cóncavo y polígono convexo

No hay, por tanto, una sola la solución y vamos a analizar las que se han contemplado para resolver el problema en el caso del conversor y cuál ha sido la solución definitiva.

3.3.1. Triangulación con el algoritmo *Ear clipping*

Una forma de triangular se basa en el simple hecho de que un polígono con al menos 4 vértices sin agujeros tiene como mínimo dos "orejas" (*ears*), las cuales son triángulos, separados por la arista interna que atravesaría al polígono uniendo dos vértices que no pertenecieran a una misma arista del perímetro. El algoritmo consiste en encontrar una oreja y quitarla del polígono, lo cual daría como resultado otro polígono al que volver a aplicar el algoritmo hasta que sólo quedara un triángulo.

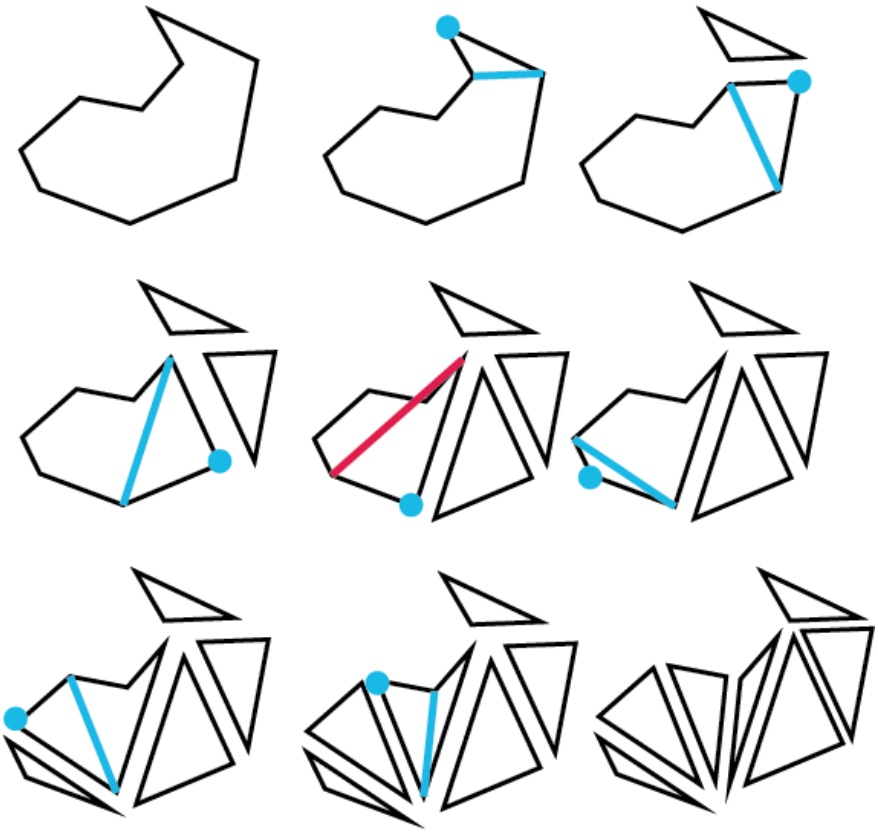


Figura 28. Ejemplo de triangulación con el algoritmo *Ear clipping*

Este algoritmo es fácil de implementar, pero más lento que otros algoritmos, y sólo funciona con polígonos que no tengan agujeros. Una implementación que tenga listas separadas de vértices cóncavos y convexos se ejecutará en tiempo $O(n^2)$ en el mejor de los casos y $O(n^2 \cdot \log(n))$ en el peor de los casos. Este método se conoce como *Ear clipping* [11] [12] y en ocasiones también como *Ear trimming*. Veamos cómo es el pseudocódigo:

Dado un polígono con una lista de puntos `P`, la función de triangulación que implementa el algoritmo *Ear Clipping* devolverá un array `listaTriángulos` donde almacenará los puntos de los `n` triángulos (orejas) en que descomponga a `P`.

Una oreja es un triángulo con los vértices ordenados en un sentido determinado, según convenio, preferiblemente CCW, y que no contiene ningún otro punto del polígono.
Así, si un triángulo se define por los puntos `A`, `B` y `C`, la lista de `n` triángulos se compondrá en este orden: $A_1, B_1, C_1, A_2, B_2, C_2, \ldots, A_n, B_n, C_n$.

```
FUNCIÓN Triangular(P):

    listaTriángulos=vacío;
    iniP=P;                        //Puntos iniciales

    SI (número de puntos < 3):
     devolver listaTriángulos;  //No es triangulable
    SI NO:
     triánguloEncontrado=false;

     MIENTRAS queden puntos en P:

          SI (no triánguloEncontrado):
            devolver listTriangulos;  //P no es triangulable.
            //Tras una iteración, no se encontró ninguna oreja.
          FIN_SI

          triánguloEncontrado =false; //comenzamos la búsqueda

          POR CADA 3 puntos consecutivos de P, siendo Pi el
          punto actual, probamos el triángulo candidato a oreja:
            pPrev= Pi;
            pCurr= Pi+1;
            pNext= Pi+2;

            SI (no triánguloEncontrado):
              res=false;

              SI este triángulo está ordenado correctamente:
                res=true;
                POR CADA punto inipj de iniP:
                  SI inipj está en este triángulo:
                   res=false;  //Este triángulo no es una oreja
                  FIN_SI
                FIN_POR_CADA
              FIN_SI
```

```
                SI (res):  //Tenemos una oreja
                   trianguloEncontrado =true;
                   Añadir pPrev, PCurr y PNext a listaTriángulos;
                   Borrar de P los puntos de este triángulo, porque
                   sabemos que es una oreja y ya no lo necesitamos;
                FIN_SI
             FIN_SI
          FIN_POR_CADA
       FIN_MIENTRAS
    FIN_SI

    devolver listaTriángulos;

FIN_FUNCIÓN
```

Para ver si un triángulo tiene sus puntos ordenados correctamente, basta con calcular el determinante de los vectores `(PCurr-pPrev)` y `(pNext-PCurr)`. Si el orden de definición de un polígono es convenido como CCW, el resultado debe ser menor o igual que `0`. Si el resultado fuese mayor que `0`, significaría que el triángulo que se ha comprobado en realidad es una zona cóncava del polígono.

El método de cálculo de un punto dentro de un triángulo se conoce como colisión de un punto dentro de un polígono. Para más información de estos los algoritmos de estos dos métodos, se puede consultar la web de la fuente [13], un trabajo final de carrera realizado por G. Pérez y dirigido por M. Abellanas.

Para el conversor se pensó como primera opción usar la triangulación de Delaunay, de la cual se hablará en el siguiente apartado pero, como se verá, da problemas con los polígonos cóncavos. Por eso, se tomó como segunda opción el algoritmo *Ear clipping*. Funcionó correctamente con los objetos que definidos con polígonos convexos y cóncavos.

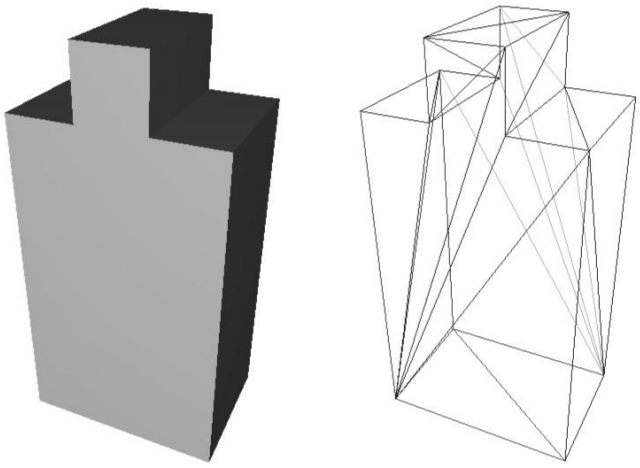


Figura 29. *Ear clipping* con un sólido de polígonos cóncavos y convexos

Sin embargo, al probar objetos con agujeros el algoritmo falló, dando lugar a resultados extraños como caras incompletas, como sucedió con el caso de un prisma con agujero

pasante, en el que las caras que contenían el agujero, faltaban dos triángulos, como se puede ver en la Figura 30.

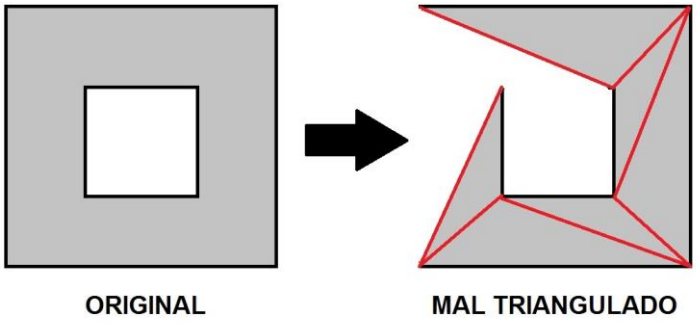


Figura 30. *Ear clipping* con un polígono que contiene un agujero

3.3.2. Triangulación de Delaunay

Una condición deseable, para la que se podría calificar de buena triangulación es que sus triángulos fuesen regulares, o al menos lo más regulares posible.

Imaginemos un conjunto de clavos clavados, pero no por completo, en una tablilla de madera. Si colocáramos una goma elástica alrededor del conjunto veríamos como resultado un polígono convexo. Esto nos da a entender mejor que, dado un conjunto de puntos o vértices, llamamos **envolvente convexa o cierre convexo** al polígono que dibujan al unirlos con líneas y sin dejar ningún ángulo cóncavo, dejando o no encerrados otros puntos que estén en el interior del área polígono.

Una triangulación de Delaunay cumple la **condición de Delaunay**. Esta condición dice que la circunferencia circunscrita de cada triángulo de una malla de triángulos no debe contener ningún vértice un triángulo que no sea del la define. También se dice que estas circunferencias son vacías.

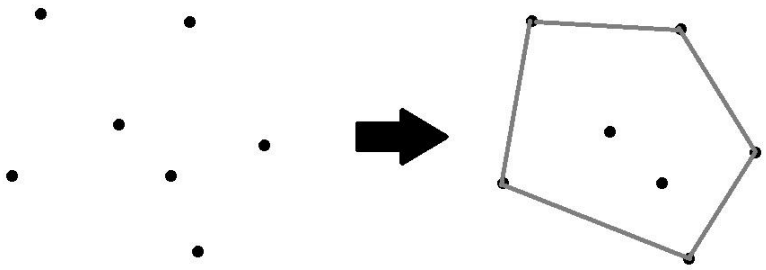


Figura 31. Envolvente convexa de un conjunto de puntos

Otra cosa que se cumple en la **condición de Delaunay** es que no existen aristas ilegales en la malla de triángulos. Para definir lo que es una **arista ilegal**, pensemos

primero en una arista de la triangulación. Si no pertenece a la envolvente convexa, separa dos triángulos de la triangulación. Sabiendo que esos dos triángulos forman un cuadrilátero convexo, la arista será una de sus diagonales. Si esa diagonal es ilegal, la circunferencia circunscrita a uno de los dos triángulos que separa la arista contendrá en su interior por completo al otro triángulo. Para evitar esta situación hay que **legalizar** la arista. Ello consiste en sustituir la arista ilegal por la otra diagonal del cuadrilátero descrito por los triángulos. Nótese que salvo que los cuatro vértices sean concíclicos, en cuyo casi ambas aristas serán legales siempre habrá una arista legal y otra legal.

Como se puede ver en la Figura 32, en el caso de la izquierda ambas circunferencias no son vacías porque las dos contienen a ambos triángulos, luego su arista es ilegal. En el caso de la derecha, tomando la otra posible arista del cuadrilátero, se cumple la condición de Delaunay porque cada circunferencia contiene vértices de un sólo triángulo.

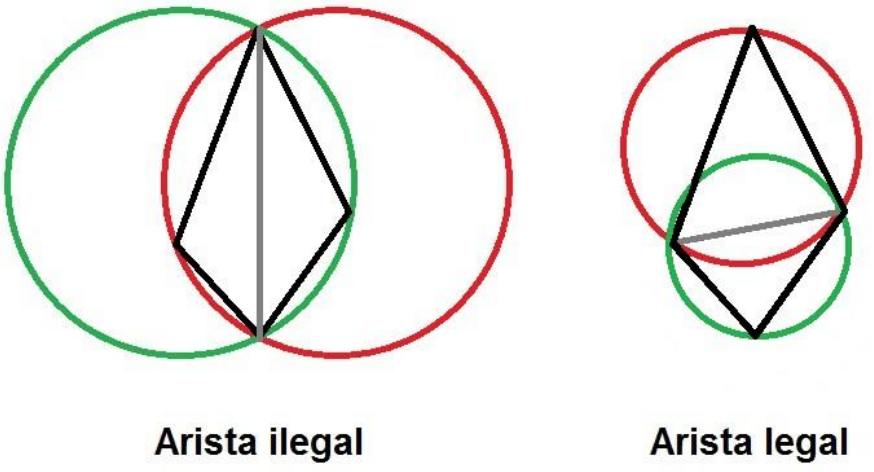


Figura 32. Diferencia entre arista ilegal y arista legal

En conclusión, dado un conjunto de puntos triangulados y legalizando una a una las aristas ilegales se consigue una triangulación mejorada, con una complejidad temporal de O(n log n), con triángulos lo más regulares posible y que es conocida como **triangulación de Delaunay**. Se puede consultar un interesante y ameno artículo que establece la relación entre la envolvente convexa, la triangulación de Delaunay y el diagrama de Voronoi escrito por Manuel Abellanas [14], así como más información y detalles sobre la construcción del algoritmo de triangulación de Delaunay en el artículo de Wikipedia [15].

OSG dispone de una librería de funciones dentro de `osgUtil` llamada `delaunayTriangulator`, que implementa la triangulación de Delaunay y que su vez se apoya en la librería `Tesselator`, también de la librería `osgUtil`.

`Tesselator` asume que el polígono que se le pasa está sobre el plano XY, mirando hacia Z positiva. Internamente, el triangulador coloca 3 puntos alrededor de los vertices que se van a teselar, formando un super-triángulo que los va a contener completamente. Los puntos extra se quitan tras la triangulación. Este super-triángulo

es el primer triángulo. Obviamente, este contiene muchos otros puntos y no formará parte de la malla de triángulos final. El algoritmo, entonces, busca un punto contenido en el super-triángulo y divide el super-triángulo en al menos 2 triángulos, aunque generalmente son 3. El super-triángulo se elimina y entonces se busca en cada uno de los nuevos triángulos un punto contenido en ellos. Si encuentra ese punto, divide y borra el triángulo en el que lo ha encontrado, igual que antes con el super-triángulo. Esto continúa hasta que no quedan más puntos por encontrar en los triángulos. Esos triángulos luego se corrigen legalizando sus aristas, como se ha descrito antes.

En la implementación de OSG el procedimiento es crear un objeto `DelaunayTriangulator` al que se le pasa un array con los vértices de la cara a triangular, preparar el array de normales que se generarán tras la triangulación y llamar a la función `triangulate()` para que se encargue de todo.

Volviendo a hablar del conversor, la primera dificultad que encontramos es que falla la triangulación de Delaunay porque las caras que componen a un objeto tridimensional no están todas en el plano XY, como asume `Tesselator`. Al tomar sólo en cuenta los valores x e y, se dan errores como por ejemplo si la cara estuviera sobre el plano YZ, en cuyo caso vértices con mismos valores de x e y pero diferentes valores de z los interpretaría como puntos repetidos y no triangularía bien. Esto se resuelve simplemente rotando los vértices de cada cara al plano XY, aplicar Delaunay y luego devolverlos a su ubicación original.

Respecto a la rotación de los puntos, hay una cuestión a tener en cuenta de la que se hablará en el apartado 3.4.3, Estructura de clases del proyecto.

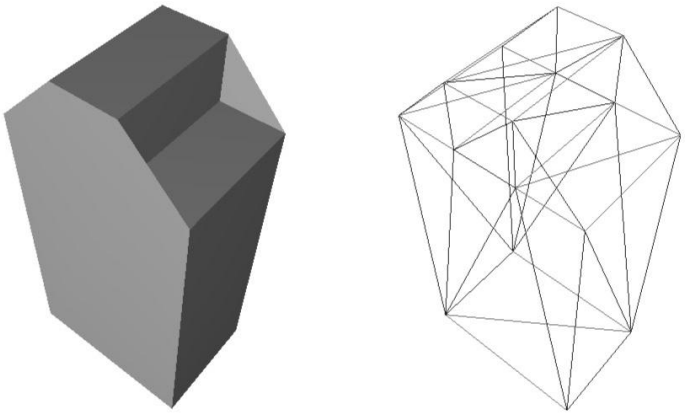


Figura 33. Triangulación de Delaunay con caras cóncavas

Como se puede observar en la Figura 33, la segunda dificultad viene dada por la propia naturaleza del algoritmo de triangulación: sólo funciona con polígonos convexos, así que la triangulación de objetos 3D con alguna cara cóncava, aunque con triángulos bastante más regulares, no es satisfactoria y aparecen "orejas" que es necesario recortar, como se ha visto con el algoritmo del *Ear clipping*.

Es obvio, por tanto, que tampoco soporta la triangulación de caras con agujeros, por lo que hay que dar un paso más y buscar una oportunidad en una variante de la triangulación de Delaunay: Delaunay Restringido o **Delaunay Constrained.**

3.3.3. Triangulación de Delaunay restringida

La **triangulación de Delaunay restringida** es una generalización de la triangulación de Delaunay que fuerza ciertos segmentos a la triangulación. Debido a que una triangulación de Delaunay es casi siempre única, a menudo una triangulación de Delaunay restringida contiene aristas que no satisfacen la condición de Delaunay. Por tanto, la triangulación de Delaunay restringida muchas veces no es realmente una triangulación de Delaunay en sí misma.

Una restricción de Delaunay o *Delaunay Constraint*, es un mecanismo para definir qué aristas debe retenerse en una triangulación de Delaunay. En el caso del conversor, necesitamos desechar los triángulos que forman parte de agujeros o concavidades y quedarnos con el resto del polígono.

OSG dispone en la librería `osgUtil` de la clase `DelaunayConstraint`. Esta deriva de la clase `Geometry`; así que, la restricción puede consistir en una o más secuencias de líneas (`LINE_STRIP`) o en uno o más bucles de líneas (`LINE_LOOP`). La triangulación final incluirá estas líneas como aristas en la malla triangulada.

Algunos usos que se puede hacer, por ejemplo, es tomar los triángulos definidos en la restricción de Delaunay y crear con ellos una nueva geometría (un objeto de tipo `Geometry`) que reemplace, por ejemplo, el área de un terreno, quizás con una textura diferente, como agua para situar un lago. Incluso se puede derivar una restricción de Delaunay personalizada para crear un método personal de reemplazo.

Las funciones de utilidad que tiene OSG permiten que los triángulos definidos en la restricción de Delaunay se puedan borrar. Esto es lo que ayuda a crear agujeros en una superficie.

Este es el orden correcto implementado en OSG para aplicar una triangulación de Delaunay, restricciones incluídas:

1- Crear un objeto de tipo `DelaunayTriangulator dt`.

```
osg::ref_ptr <osg::Util::DelaunayTriangulator> dt=
        new osgUtil::DelaunayTriangulator;
```

2- Crear las restricciones de Delaunay necesarias y añadirlas a `dt`.

```
osg::ref_ptr <osgUtil::DelaunayConstraint> dc=
   new osgUtil::DelaunayConstraint;
osg::Vec3Array *agujero= new osg::Vec3Array;
dc->setVertexArray(agujero);
//n= número de vertices del agujero.
dc->addPrimitiveSet(
      new osg::Drawarrays(
        osg::PrimitiveSet::LINE_LOOP,0,n));
dt->addInputConstraint(dc.get());
```

3- Añadir a `dt` los puntos que hay que triangular.
```
dt->setInputArray(points.get());
```

4- Preparar a `dt` el array de normales que se generarán tras la triangulación.
```
dt->setOutputNormalArray(new osg::Vec3Array);
```

5- Llamar a la función de triangular de `dt`.
```
dt->triangulate();
```

6- Eliminar los triángulos descritos en las restricciones.
```
dt->removeInternalTriangles(dc.get());
```

7- Obtener las primitivas resultantes de la triangulación.
```
dt->getTriangles();
```

Pero, ¿cómo aplicar esto en el proyecto de conversor? En el formato de archivo Brep que tratamos, recordemos que los agujeros siempre se describen justo a continuación de la cara "normal" a la que pertenece (etiquetada con "c"). Por tanto, cuando el programa lee del archivo y detecta una cara interna (etiquetada con la letra "i"), debe acceder a la última cara "normal" almacenada para añadir esta cara interna a su lista de agujeros. Ejemplo:

```
CaraF es la última cara "normal" leída desde un archivo Brep

c caraF a19 a18 a17 a20
i intF  a23 a24 a21 a22
i intG  a25 a26 a27 a28 a29 a30

Esto significa que ambas caras internas intF e intG son un par de
agujeros descritos en caraF
```

Suponiendo que el objeto a convertir en objeto STL en el conversor contiene caras con agujeros, y que la lectura del archivo Brep ha sido correcta, el algoritmo seguido es:

```
POR CADA cara leída del objeto Brep:

  POR CADA agujero de su lista de agujeros (si los hay):
  Invertir el orden de los puntos para una correcta rotación.
  Rotar los puntos del agujero al plano XY
  Crear una restricción de Delaunay
  Añadir la restricción a la lista de restricciones
  FIN_POR_CADA

FIN_POR_CADA
```

Las restricciones que se incluyan en esta lista se añadirán al objeto `DelaunayTriangulator`, antes de llamar a la función `triangulate()`.

Una vez eliminados los triángulos con `removeInternalTriangles()` y habiendo tratado debidamente los triángulos resultantes de la triangulación para obtener el archivo STL, al ejecutar el conversor observaremos que triangula perfectamente objetos de caras convexas con o sin agujero, incluso si el agujero tiene forma de polígono cóncavo, como se puede ver en la Figura 34.

Pero sigue fallando cuando una cara "normal" es cóncava y aparecen, de nuevo, las molestas orejas de concavidades que es necesario recortar. En la Figura 35, se puede observar en el objeto de ejemplo una casa en la que la que el agujero de la chimenea está bien definido, mientras que el hueco de la puerta y parte del porche están "tapiados" con orejas, porque ambos están definidos con alguna cara cóncava. De nuevo hay que dar un paso más hacia la solución perfecta.

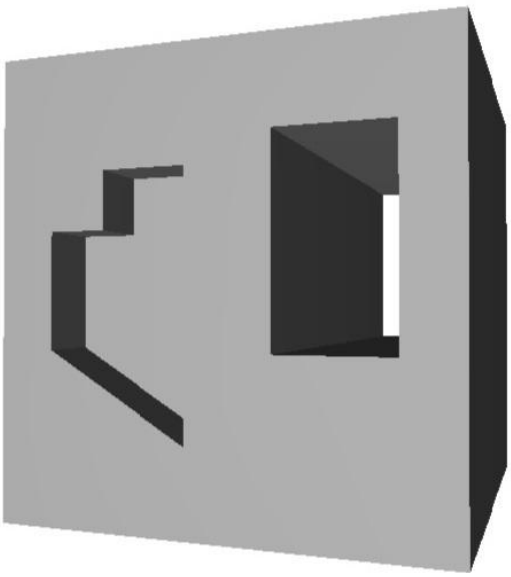


Figura 34. Delaunay restringido. Objeto con agujeros y sólo caras convexas.

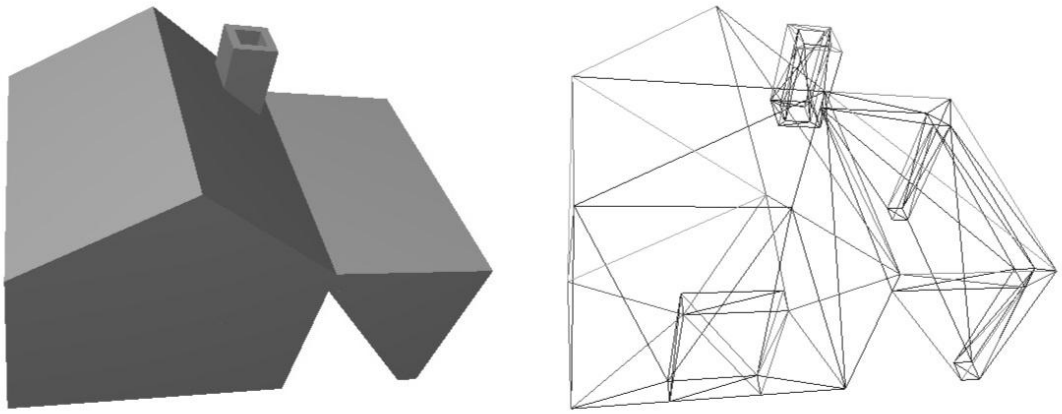


Figura 35. Delaunay restringido. Objeto con agujeros y caras convexas y cóncavas.

3.3.4. Solución definitiva: Delaunay restringido + *Ear clipping*

Ya hemos visto que el algoritmo *Ear clipping* no obtiene la triangulación de polígonos óptima y funciona con polígonos convexos y cóncavos, pero no admite polígonos con agujeros.

Por otro lado, Delaunay logra la triangulación óptima de un polígono y, jugando con las restricciones de Delaunay, se puede conseguir además triangular polígonos con uno o más agujeros. Sin embargo falla con polígonos cóncavos.

Llegamos a la conclusión de que lo que necesitamos es añadir un mecanismo que detecte los triángulos sobrantes en un polígono cóncavo y los elimine.

Lo más lógico es utilizar el algoritmo de *Ear clipping* y modificarlo ligeramente para que cuando encuentre un triángulo perteneciente a una concavidad, en lugar de desecharlo, lo guarde en el array de triángulos que devuelva. En la implementación del conversor, esto se ha encapsulado en una función llamada `getConcaveTriangles()`, a la que se le pasa el conjunto de puntos del polígono a comprobar. Si el array de concavidades no está vacío, se procede a "fundir" los triángulos de esas concavidades para obtener los polígonos que definirán las restricciones de Delaunay referidas a las concavidades que hay que recortar. La función encargada de hacerlo es `getPolygons()`. Por cada polígono, entonces, se creará una restricción de Delaunay que se añadirá a la lista de restricciones.

El resultado de esta combinación de técnicas de triangulación da lugar a lo que se esperaba conseguir fundamentalmente del conversor: la transformación de un objeto definido en formato Brep, con caras poligonales de cualquier número de vértices, cóncavas, convexas, con o sin agujeros, en un objeto equivalente con facetas triangulares.

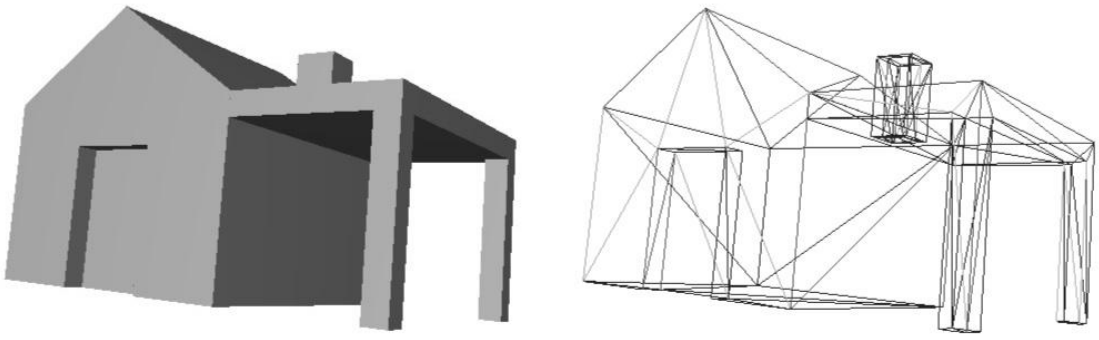


Figura 36. Triangulación correcta con polígonos convexos, cóncavos y agujeros.

3.4. Diseño de la aplicación

3.4.1. Algoritmo general del conversor

Esto es, a grandes rasgos lo que ocurre al ejecutar programa conversor:

1- Acceder a los datos de un archivo Brep, de extensión .brp.
2- Si el acceso es correcto, escribir un nodo osg con los datos, siguiendo los pasos:

a. Leer línea por línea el archivo .brp e interpretar el tipo de dato según etiqueta al inicio de la línea:

i. Si la etiqueta es `objeto` u `Objeto`: Si no hay ningún objeto BRep aún registrado, crear nuevo objeto BRep, con nombre y listas vacías de vértices, aristas, caras y colores.
ii. Si la etiqueta es `v` o `V`: Crear un vértice con nombre y coordenadas XYZ y añadirlo a la lista de vértices del objeto BRep.
iii. Si la etiqueta es `a` o `A`: Crear una arista con nombre, vértice inicial, vértice final, cara adyacente izquierda, cara adyacente derecha, arista entrante izquierda, arista saliente derecha, arista entrante derecha y arista saliente derecha. Añadir la arista a la lista de aristas del objeto BRep.
iv. Si la etiqueta es `c` o `C`: Crear una cara "normal" con nombre, lista de aristas ordenadas y lista vacía de agujeros. Añadir la cara a la lista de caras del objeto BRep.
v. Si la etiqueta es `i` o `I`: Crear una cara "interna" (agujero) con nombre y lista de aristas ordenadas. Añadir la cara interna a la lista de agujeros de la última cara "normal" leída del archivo..
vi. Si la etiqueta es `color`: Crear un nuevo color con nombre, valores RGB y lista de caras que llevarán este color. Añadir este color a la lista de colores del objeto BRep.

b. Crear un nuevo objeto teselado en triángulos con el nombre del objeto BRep y una lista de facetas vacía:

Por cada cara del objeto Brep:
Obtener su lista de vértices.
Si el número de vértices es 3:
Crear un triángulo con esos vértices .
Añadir el triángulo a la lista de facetas.
Si no, si el número es mayor que 3:
Calcular el vector normal para saber su orientación.
Rotar los puntos al plano XY, si es necesario, para triangular correctamente.
Preparar restricciones de Delaunay:
para agujeros, si los hay.
para las concavidades, si la cara es cóncava.

Aplicar triangulación de Delaunay y quitar los triángulos que indiquen las restricciones.
Obtener la secuencia de vértices de los nuevos triángulos.
Rotar los vértices de la lista a su posición original.
Cada 3 vértices de la lista de vértices:
Crear una nueva faceta.
Añadirla a la lista de facetas del objeto teselado.

c. Crear un nodo osg con los datos del objeto teselado con los datos:
   - Nombre
   - Array de puntos
   - Array de normales
   - Array de primitivas
   - Array de colores

d. Registrar el plugin para que osg sea capaz de reconocer archivos BRep.

3- Enviar los datos del nodo al visor de osg y al escritor de archivos del plugin STL de osg.

3.4.2. Control de errores

El sistema cubre los siguientes errores:

Errores de lectura o entrada:

- Si en la llamada al programa falta el nombre del archivo de entrada.
- Si la extensión del archivo de entrada no tiene extensión `.brp`.
- Si la extensión del archivo de salida no tiene extensión `.stl`.
- Si se provoca un fallo de lectura del archivo por ser inexistente en la ruta especificada o por algún otro motivo interno desconocido.

Errores de declaración en el archivo Brep:

- Si durante el proceso de conversión se intenta manipular algún elemento que o ha sido declarado anteriormente en el archivo: caras, aristas y vértices.

Errores de escritura o salida:

- Si sucede algún problema durante la escritura del archivo de salida `.stl`.

3.4.3.Estructura de clases del proyecto

El estilo de programación del proyecto es orientada a objetos. Como se ha visto en el algoritmo general del conversor, los datos que se manipulan son almacenados en instancias de objetos definidas en clases. De esta manera, podemos dividir la estructura de archivos del código en varios bloques:

Archivos de control de errores

Se encargan de definir las excepciones que se lanzan durante la ejecución del programa y de mostrar los mensajes de error correspondientes.

- **ConversorExceptions.h, ConversorExceptions.cpp**

Archivos relacionados con BRep

Son las clases que definen la estructura y atributos de un objeto 3D sólido BRep:

- **BRepObject.h, BRepObject.cpp:** Nombre, y listas de vértices, aristas, caras y colores.
- **Color.h, Color.cpp:** Nombre, valores RGB y lista de caras de este color.
- **Edge.h, Edge.cpp:** Nombre, vértices de incisión, caras adyacentes y lista de aristas de entrada y salida.
- **Face.h, Face.cpp:** Nombre, lista de aristas ordenadas, lista de agujeros y atributo booleano que dice si es cara interna o no.
- **Vert.h, Vert.cpp:** Nombre, coordenadas XYZ.

Archivos relacionados con los resultados de la triangulación

Son las clases que definen a un objeto 3D sólido formado por caras triangulares. Muy similar a STL:

- **TesselatedObject.h, TesselatedObject.cpp:** Nombre y lista de facetas triangulares.
- **TriangularFace.h, TriangularFace.cpp:** Bucle de 3 puntos 3D, vector normal y color vector de color RGBA.

Archivos relacionados con el proceso de triangulación

Aquí hay que distinguir entre la clase auxiliar Polygon, el archivo que contiene el código del plugin de BRep, ReaderWriterBRP, y la colección de funciones auxiliares MyUtil.

- **MyPolygon:** Esta clase define un polígono con un bucle de vértices. Se usa para definir los polígonos resultantes de recortar las zonas cóncavas de la cara de objeto BRep. Cada uno de estos *MyPolygon* será una restricción de Delaunay.
- **ReaderWriterBRP.h, ReaderWriterBRP.cpp:** Es el plugin de osg que se encarga del proceso de lectura de archivos BRep y de transformación en un

objeto 3D teselado en triángulos. Por el momento, soporta la lectura de archivos .brp, pero no su escritura.

- **MyUtil:** Define una serie de funciones que no pertenecen a ninguna clase en concreto, pero que encapsulan instrucciones útiles para la triangulación de un objeto. Estas funciones son:

    - **fToS:** Convierte un valor `float` a una cadena `String`

    - **calcNormalNewell:** Método de Newell para calcular la normal de un polígono 3D arbitrario.

    - **calcMinAngle:** Calcula en grados el ángulo mínimo entre dos vectores.

    - **getAngles:** Revisa todas las condiciones necesarias para saber qué ángulo de rotación alrededor de cada eje es más apropiado para dejar el vector normal que se le pasa perpendicular al plano XY, mirando al eje Z negativo.

    - **getPolygons:** Funde los puntos de los triángulos adyacentes de la secuencia de triángulos que se le pasa para obtener uno o más polígonos.

    - **determinant:** Calcula el determinante de dos vectores.

    - **getConcaveTriangles:** Dado un conjunto de puntos y usando el algoritmo de triangulación *Ear clipping* modificado, determina si el polígono que forman tiene triángulos cóncavos y los devuelve en un array.

    - **rotateFaceToXY:** Rota los puntos de una cara al plano XY para poder triangular correctamente con el método de Delaunay. Nótese, en primer lugar, que la misma función se encarga de rotar al plano XY o devolver los puntos a su posición original con cambiar el valor de un parámetro booleano. Téngase en cuenta, también, que en el formato BRep que estamos usando para este proyecto, las caras visibles de un objeto están definidas siguiendo el sentido CW y las internas siguen el orden CCW. El método de Delaunay asume que el conjunto de puntos que se le pase para triangular sigue el convenio de la regla de la mano derecha, es decir, justo al revés que este formato BRep. Si se rotara una cara visible dejando su normal apuntando hacia el eje Z positivo, Delaunay la triangularía correctamente, pero la interpretaría como una cara interna porque los puntos seguirían el orden CW; por tanto, generaría facetas triangulares internas, de tal manera que el objeto teselado resultante podría visualizarse erróneamente. Por ejemplo, en el visor de osg, al ser todas las facetas internas, el efecto sería que el objeto teselado estaría iluminado desde el interior y desde el exterior se vería como si estuviera en una estancia a oscuras, como se ilustra en la Figura

37, en la que una casa sin tejado muestra iluminado el interior, pero no el exterior.

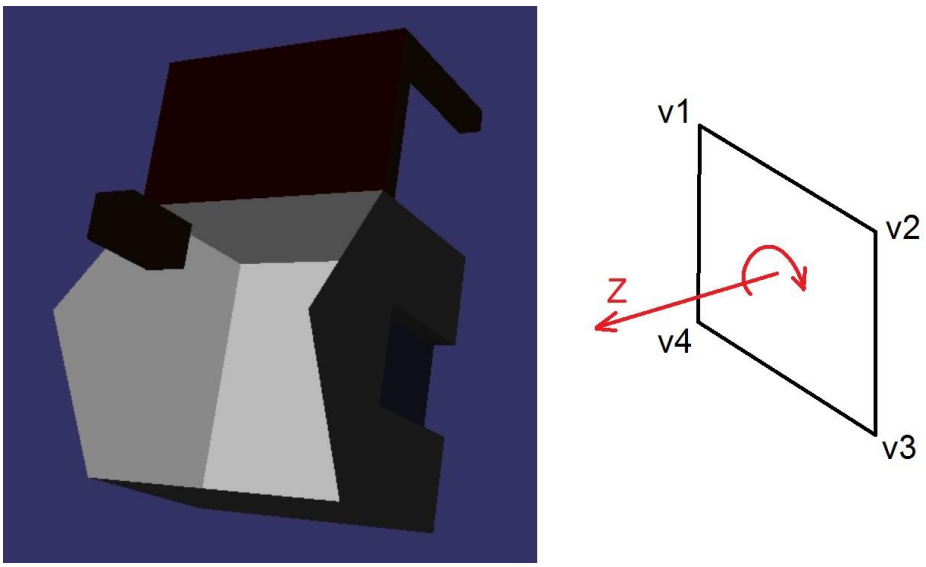


Figura 37. Rotación de caras al plano XY con normal mirando a eje Z positivo.

3.5. Resultados

3.5.1. Planificación de tareas

A continuación se muestra una tabla en la que se puede ver el proceso de desarrollo de este proyecto final de carrera, subdividido en tareas, junto al tiempo estimado y el tiempo real empleado para llevar a cabo cada una de ellas.

TAREA	**TIEMPO ESTIMADO**	**TIEMPO REAL**
Investigación sobre BRep y STL	4 semanas	8 semanas
Investigación sobre la tecnología de desarrollo	2 semanas	1 semana y 3 días
Puesta a punto del entorno de desarrollo	1 semana	5 días
Implementación y pruebas	12 semanas	10 semanas
Escritura de memoria	4 semanas	4 semanas
TOTAL	**20 semanas (5 meses)**	**24 semanas (6 meses)**

3.5.2. Programa resultante y conclusiones

El programa resultante ha sido una sola versión compatible con Windows de 32 y de 64 bits que cumple con el objetivo principal de obtener un archivo de formato STL a partir de uno de formato BRep y muestra por su visor propio el objeto resultante de la conversión. El programa sólo acepta la lectura de ficheros BRep bien definidos y su ejecución se efectúa desde una ventana de línea de comandos.

La investigación inicial sobre todo tipo de información sobre BRep y STL se ha dilatado demasiado con relación al tiempo total de desarrollo disponible. A pesar de que se ha compensado ligeramente con el tiempo real usado en otras tareas, el tiempo real total supera en un mes al tiempo total estimado. Ello implica que en el periodo de implementación no se hayan hecho todas las pruebas convenientes ni se haya podido hacer una versión más completa del programa.

La conclusión es que el programa es poco robusto y excesivamente básico y necesita mejoras y ampliaciones.

4. Mejoras y ampliaciones

Este capítulo es muy específico presentando una lista de propuestas de mejoras y ampliaciones del proyecto con breves descripciones de las mismas.

4.1. Mejoras

- **Escritura de archivos STL binarios:** El plugin STL que tiene osg no soporta escritura de archivos STL binarios. Los binarios admiten información de color, por lo que esta mejora sería interesante.

- **Interfaz de usuario nueva:** La idea es que se convierta en una aplicación completamente de ventanas, por ser mucho más amigable para el usuario.

4.2. Ampliaciones

- **Control de errores perfeccionado:** El programa podría sustituir al Visor BRep si hiciera un análisis del archivo BRep que lee y dar mensajes de error concretos y en qué línea los encuentra para que el usuario sepa cómo corregir los fallos de definición o de sintaxis.

- **Implementar la conversión STL-BRep:** Lo que implicaría ampliar el plugin ReaderWriterBRP para que escriba archivos de extensión `.brp`.

- **Versión compilada para otras plataformas:** Como MAC, Linux o Android.

Referencias

[1] Wikipedia, «Estereolitografía,» Marzo 2015. [En línea]. Available: http://es.wikipedia.org/wiki/Estereolitograf%C3%ADa.

[2] V. Matoses Correcher, «Proyecto Final de carrera - Visor BRep,» Universitat Politècnica de València. Escola Tècnica Superior d'Enginyeria Informàtica, 21 05 2012. [En línea]. Available: https://riunet.upv.es/handle/10251/15804. [Último acceso: 2015].

[3] OpenSceneGraph, «http://www.openscenegraph.org/,» [En línea].

[4] E. Vendrell Vidal y U. P. d. Valencia, «Representacion B-Rep.© UPV,» 21 Sept 2011. [En línea]. Available: www.youtube.com/watch?v=vg_TZsTM9hE. [Último acceso: 2013].

[5] R. Stouffs, R. Krishnamurti y C. Eastman, «A formal Structure for Nonequivalent Solid Representations,» 1996. [En línea]. Available: http://cumincad.architexturez.net/system/files/pdf/0b2e.content.pdf. [Último acceso: 2015].

[6] M. Mantÿla, An Introduction to Solid Modeling, Rockville , Maryland: Computer Science Press, 1988.

[7] B. Baumgart, «A Polyhedron Representation for Computer Vision,» de *National Computer Conference*, Montvale, New Jersey, 1975.

[8] I. Braid, R. Hillyard y I. Stroud, «Construction of Polyhedra in Geometric Modeling,» de *Mathematical Methods in Computer Graphics and Design*, K. Brodlie, Ed., London, Academic Press, 1980.

[9] F. Yamaguchi y T. Tokieda, «Bridge Edge and Triangular Approach in Solid Modeling,» de *Frontiers in Computer Graphics*, D. L. Kunii, Ed., Tokio, Springer-Verlag, 1985, pp. 44-65.

[10] A. Paoluzzi, M. Ramella y A. Santarelli, «Boolean Algebra over Linear Polyhedra,» *Coputer Aided Design,* vol. 21, pp. 474 - 484, 1989.

[11] Wikipedia, «STL (File format),» 18 abril 2015. [En línea]. Available: http://en.wikipedia.org/wiki/STL_(file_format). [Último acceso: 4 noviembre 2013].

[12] ENNEX, «The StL Format - Standard Data Format for Fabbers,» 1993,1999. [En línea]. Available: http://www.ennex.com/~fabbers/stl.asp. [Último acceso: 4 Nov 2013].

[13] Michigan Tech, «The Euler-Poincaré Formula,» [En línea]. Available: http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/model/euler.html. [Último acceso: 2014].

[14] R. Wang y X. Qian, OpenSceneGraph 3.0: Beginner's Guide, Packt Publishing Ltd., 2010.

[15] Wikipedia, «Standard Template Library,» [En línea]. Available: https://es.wikipedia.org/wiki/Standard_Template_Library.

[16] Wikipedia, «Polygon triangulation - Ear Clipping Method,» [En línea]. Available: https://en.wikipedia.org/wiki/Polygon_triangulation#Ear_clipping_method.

[17] D. Taylor, «gamedev.net - Polygon Triangulation,» Junio 2014. [En línea]. Available: http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/polygon-triangulation-r3334.

[18] G. Pérez, «ALGORITMOS - Punto interior a un triángulo,» [En línea]. Available: http://www.dma.fi.upm.es/mabellanas/tfcs/kirkpatrick/Aplicacion/algoritmos.htm#kirkpatrickIntroduccion.

[19] M. Abellanas, «Envolvente convexa, triangulación de Delaunay y diagrama de Voronoi: tres estructuras geométricas en una, con muchas aplicaciones,» 2007. [En línea]. Available: http://divulgamat2.ehu.es/divulgamat15/index.php?option=com_docman&task=doc_download&gid=552&Itemid=75.

[20] Wikipedia, «Triangulación de Delaunay,» Abril 2014. [En línea]. Available: https://es.wikipedia.org/wiki/Triangulaci%C3%B3n_de_Delaunay.

[21] CMake, «Web oficial de CMake,» [En línea]. Available: http://www.cmake.org.

[22] OpenScenGraph, «Compiling with Visual Studio,» [En línea]. Available: http://www.openscenegraph.org/index.php/documentation/platform-specifics/windows/37-visual-studio.

Anexo I: Instalando OSG en Windows

La instalación de OpenSceneGraph se puede hacer de dos maneras: descargando de la web oficial los binarios de la versión deseada o bien descargar los ficheros fuente y compilarlos para la máquina en la que se va a utilizar.

Este anexo va a describir los pasos necesarios para dejar osg preparado para poder trabajar con el proyecto del conversor, en caso de querer hacerle alguna modificación.

Los requisitos mínimos, pues, para poder seguir desarrollando este proyecto en una máquina con sistema operativo Windows son:

- OpenSceneGraph 3.2.1 o superior
- Microsoft Visual Studio 2010

Por tanto, se van a describir los pasos para dejar preparado el entorno de programación para OSG 3.2.1 y Visual Studio 2010 para una máquina de 32 bits.

1. **Descargar los archivos fuente osg**

   Tanto los binarios como los fuentes se pueden desgargar de la web oficial de OpenSceneGraph [3]. Basta con ir a `Downloads -> Stable Releases` buscar la versión 3.2.1 y descargar el archivo comprimido `OpenSceneGraph-3.2.1.zip`

2. **Obtener las dependencias de osg y herramientas necesarias**

   - **Herramienta CMake** 2.8.12.2 o superior de su web oficial [16].

   - Entorno de programación Microsoft **Visual Studio 2010**.

   - **Dependencias de osg**, para el entorno de programación escogido (**VS2010**), las cuales se pueden descargar de la web de OpenSceneGraph en `Downloads->Dependencies`. En este caso, descargar todas las de pendencias de 32 bits.

     Muchas de las dependencias son opcionales y sólo son necesarios si se quiere cargar un tipo específico de datos. El sistema de compilación de osg basado en CMake puede detectar automáticamente qué dependencias están ya instaladas y habilitar o deshabilitar la compilación de varios módulos de acuerdo con lo que esté disponible.

   - ***Sample datasets***, son programas e imágenes de ejemplo de osg, no imprescindibles para el desarrollo, pero útiles en cualquier caso. Se descargan en un archivo zip de la web de OpenSceneGraph en `Downloads-> Data Resources`.

3. **Compilación de osg** [17]

   a. **Preparación**

   Hay que tener en cuenta es que se va a trabajar con 3 directorios:

   - **FUENTE:** Donde están los ficheros fuente de OpenSceneGraph.
   - **COMPILACIÓN:** Donde CMake generará un proyecto de compilación de OpenSceneGraph.
   - **BINARIOS:** Donde se dejarán los ficheros binarios de OpenSceneGraph tras compilar el proyecto de la carpeta de COMPILACIÓN.

   En la siguiente tabla se sugieren unas rutas de ejemplo para cada uno de esos directorios., aunque estos pueden ser a elección propia. A partir de aquí nos referiremos a ellos por su código:

CODIGO	RUTA EJEMPLO
FUENTE	`C:/OpenSceneGraph-3.2.1`
COMPILACIÓN	`%FUENTE%/osg321-mybuild32`
DEPENDENCIAS	`%FUENTE%/3rdParty`
BINARIOS	`C:/ myosg32-3.2.1`

   Entonces, lo primero que hay que hacer es descomprimir los ficheros fuente de osg en el directorio FUENTE.

   Lo siguiente es crear o seleccionar una carpeta `COMPILACIÓN` donde colocar los ficheros que prepare CMake. Puede estar o no dentro de la carpeta `FUENTE`.

   Ahora hay que extraer las dependencias. En el momento de la redacción de este tutorial, se pueden descargar dos archivos zip. Uno de ellos contiene una carpeta `release` y otra `debug`. El otro contiene en carpetas separadas dependencias para 32 y para 64 bits y cada una de ella las carpetas `include`, `bin` y `lib`. Conviene combinar todas las carpetas `include`, `bin` y `lib` y así tenerlo todo unificado en una sola carpeta llamada, por ejemplo, `3rdParty` que se puede incluir en el mismo directorio de los archivos fuente de osg. Téngase cuidado de mantener las versiones más modernas de algunas dependencias al combinar las carpetas. De no ser así, podría darse algún problema en la configuración.

b. **Generar proyecto y solución con CMake para VS2010**

Tras instalar CMake, ejecutarlo y configurarlo tal y como se muestra en la Figura 38. Hay que decir cuál es la ruta a la carpeta `FUENTE` y la ruta a la carpeta de `COMPILACIÓN` donde CMake dejará preparado un proyecto que se compilará con el generador que se seleccione. A continuación pulsar `Configure`. Al ser la primera vez se tendrá que escoger el generador. En este caso, Visual Studio 2010.

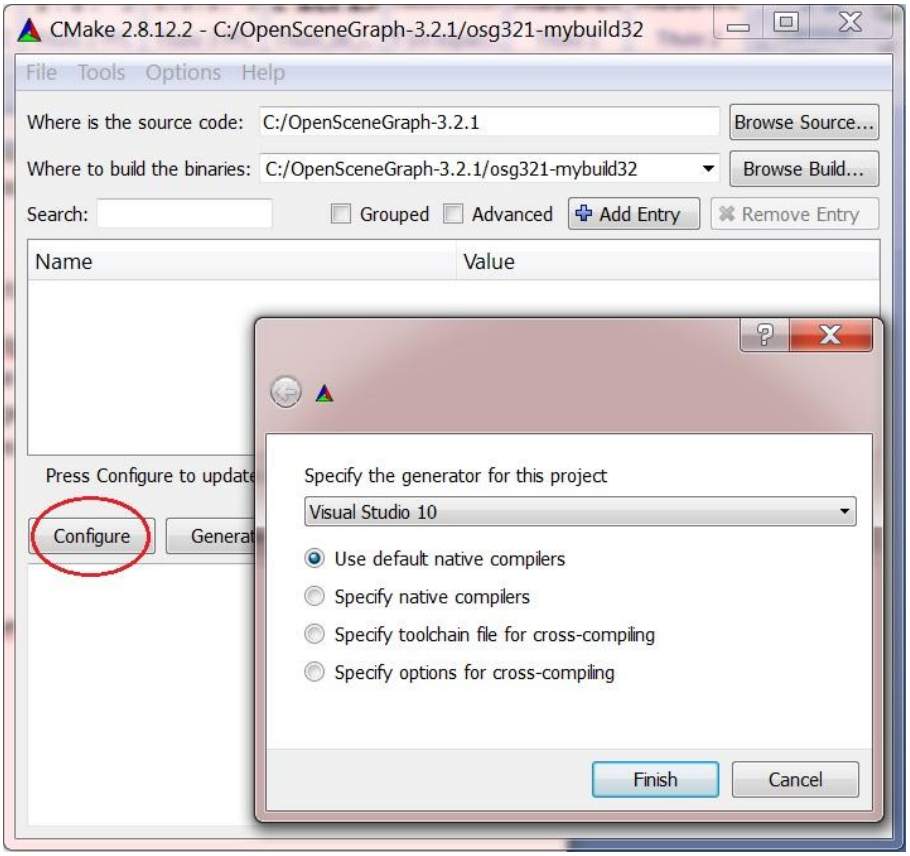


Figura 38. Configuración de CMake (I)

Pulsar `Finish`. Tras unos segundos de espera, lo más probable es que aparezcan valores en rojo que será necesario corregir.

Valores a corregir:

- **`CMAKE_INSTALL_PREFIX`**. Aquí hay que poner la ruta a la carpeta donde se quiere instalar los archivos de osg tras la compilación con VS2010, es decir, la de BINARIOS. Por ejemplo, como se ve en la Figura 39.
- **`ACTUAL_3RDPARTY_DIR`**. Aquí se pone la ruta a la carpeta de `DEPENDENCIAS`.

- **`BUILD_OSG_EXAMPLES`**. Asegurar que está seleccionada su casilla para tener a mano algunos ejemplos de osg que pueden ser útiles.
- Ocasionalmente CMake puede no encontrar algunas librerías de debug, lo que muestra algunas líneas rojas adicionales que corregir. Basta con seleccionar manualmente la ruta correcta donde se encuentre la librería especificada, por ejemplo libjpegd.lib.

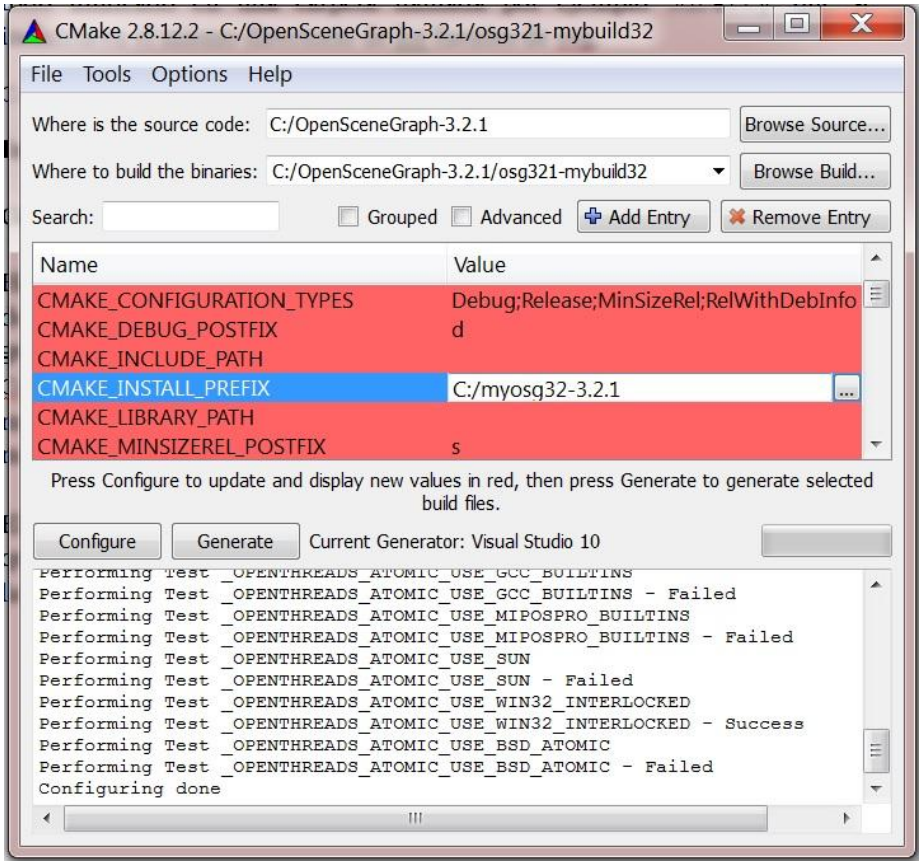


Figura 39. Configuración de CMake (II)

Pulsar el botón `Configure` de nuevo. En principio, no debería ser necesario hacer ninguna otra configuración y las líneas rojas desaparecerán. En ese caso, pulsar `Generate` para generar el proyecto de VS2010.

Cuando haya finalizado el proceso de generación, limpiar la caché seleccionando `File->Delete Cache`. A continuación, cerrar CMake.

Se puede comprobar ahora que se ha generado un proyecto de Visual Studio en `COMPILACIÓN`.

El siguiente paso es compilar osg.

c. **Compilando osg con VS2010**

Este es el paso más largo. La compilación durará un tiempo variable dependiendo de la máquina en la que se ejecute, pero puede durar alrededor de 45 minutos.

Situarse en la carpeta de `COMPILACIÓN`. Abrir la solución del proyecto haciendo doble clic sobre OpenScenGraph.sln.

Una vez abierto el proyecto, seleccionar el tipo de compilación (`Debug`, `Release`, `RelWithDebInfo` o `MinSizeRel`). NOTA: Como queremos compilar osg en 32 bits, asegurar que está seleccionado el modo de compilación en `Release` y que la plataforma seleccionada es `Win32`.

A continuación, ir al menú `Generar->Generar ALL_BUILD`. Comenzará la compilación.

Cuando termine, buscar en el explorador de soluciones el archivo `INSTALL`, hacer clic derecho sobre él y seleccionar en el menú contextual `Generar`.

Se puede seguir el proceso de compilación en la pantalla de *Resultados* seleccionando `Ver->Otras ventanas->Resultados` o pulsando `Ctrl+Alt+O`.

Una vez finalizada la compilación, se puede comprobar que en la carpeta de `BINARIOS`, la cual es la que indicamos a CMake en el parámetro `CMAKE_INSTALL_PREFIX`, se hallan ahora los archivos binarios de OpenSceneGraph 3.2.1 compilados para 32 bits.

Ahora se puede añadir también la carpeta de *datasets* a la carpeta de `BINARIOS` de osg.

La estructura final de carpetas de la instalación de osg, pues, debería quedar así:

```
myosg32-3.2.1\
    |_ bin\
    |_ datasets\
    |_ include\
    |_ lib\
    |_ share\
```

En caso de querer compilar para 64 bits, hay que hacer lo siguiente:

Ir al menú *Generar -> Administración de configuración*.
Abrir el menú desplegable *Plataforma de soluciones activas*. Si no aparece x64, seleccionar <*Nueva...*>. Ver Figura 40.

En el cuadro de diálogo que se abre, donde dice *Escriba o seleccione la nueva plataforma*, poner x64.
A continuación, seleccionar que copie la configuración de Win32. Esto literalmente copia las casillas que están seleccionadas en la columna *Generar* a la configuración para x64. Ver Figura 41.

Tras aceptar, en *Plataforma* cambiará todo de Win32 a x64. Ver Figura 42.

Cerrar y generar ALL_BUILD, como se ha descrito anteriormente.

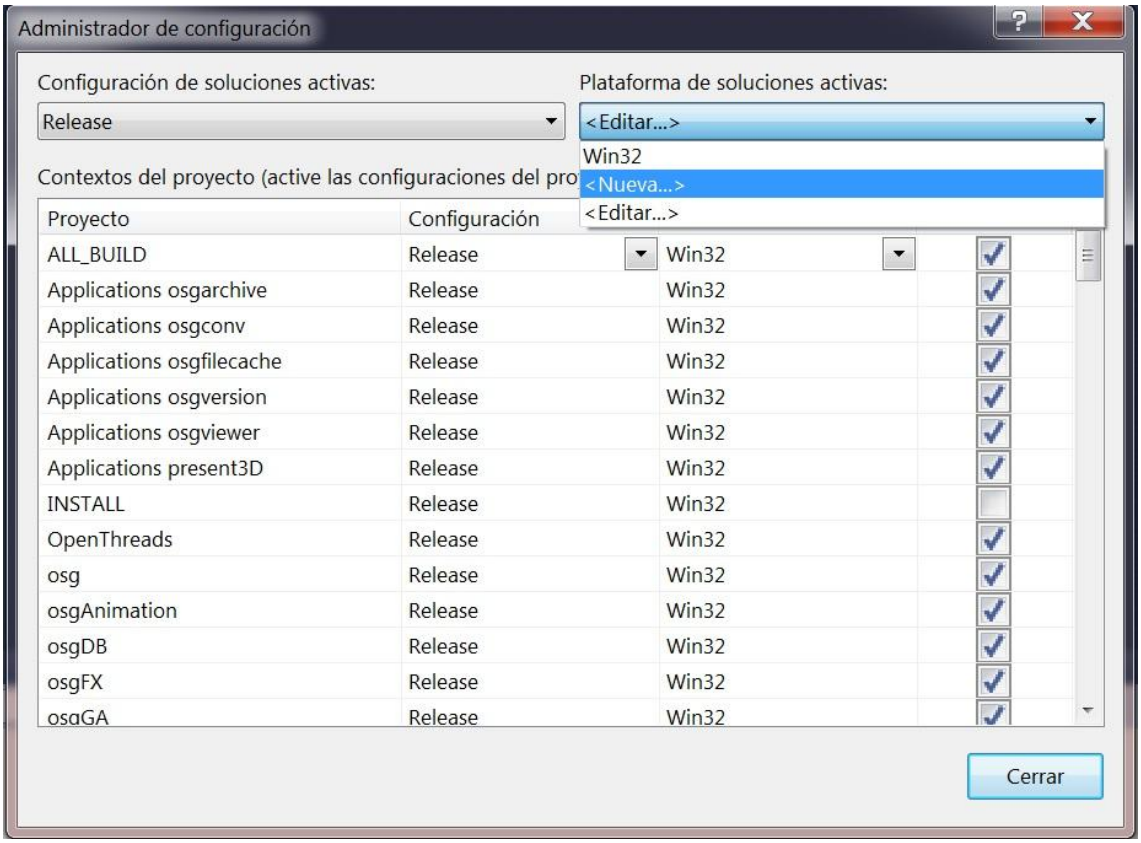


Figura 40. Configuración de plataformas de compilación (I)

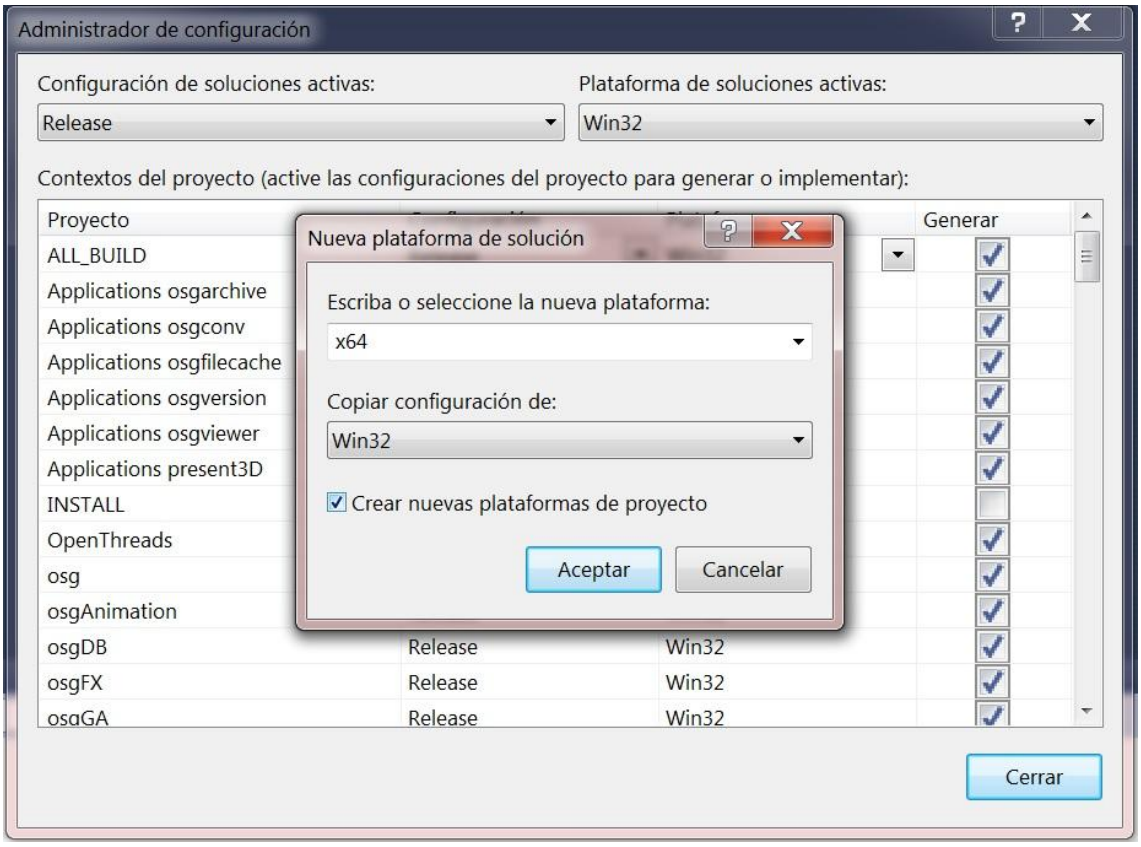


Figura 41. Configuración de plataformas de compilación (II)

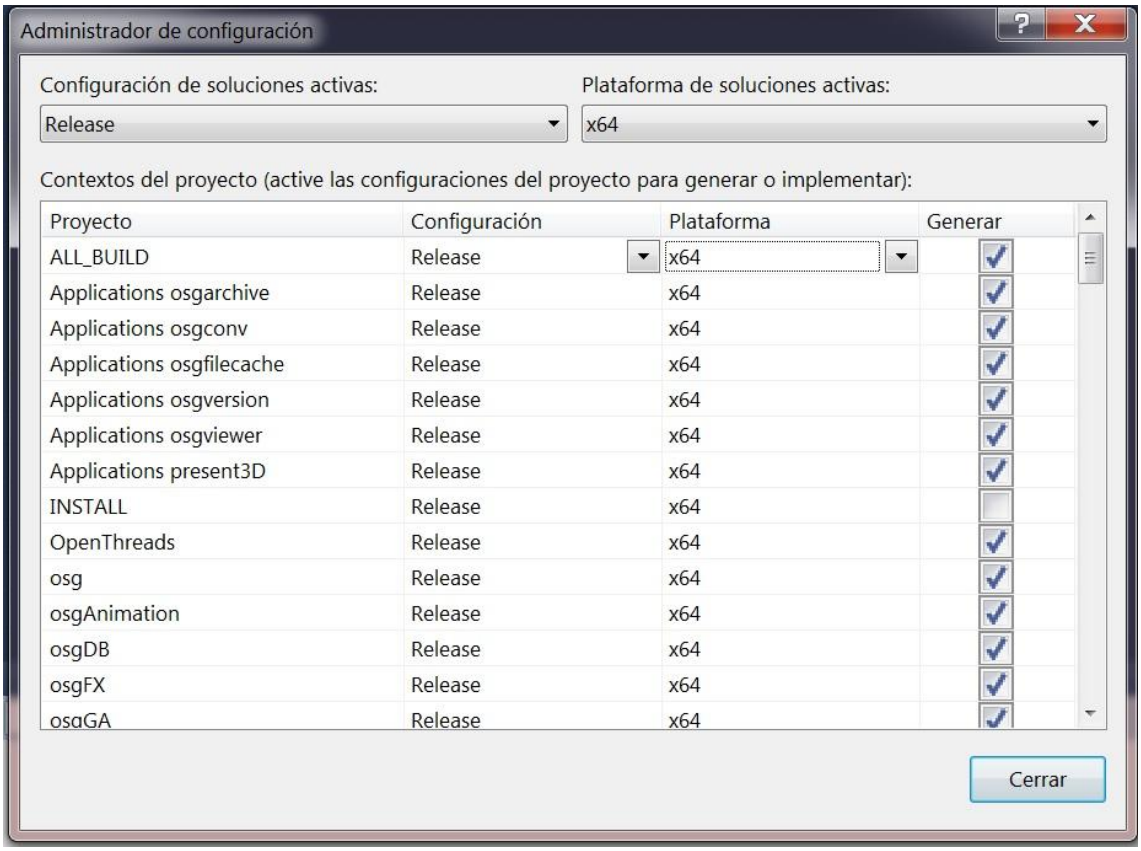


Figura 42. Configuración de plataformas de compilación (III)

4. Variables de entorno

Abrir la ventana de configuración de variables de entorno. En Windows 7 se abre así:

`Inicio -> Panel de control -> Sistema y seguridad -> Sistema -> Configuración avanzada del sistema -> Variables de entorno`

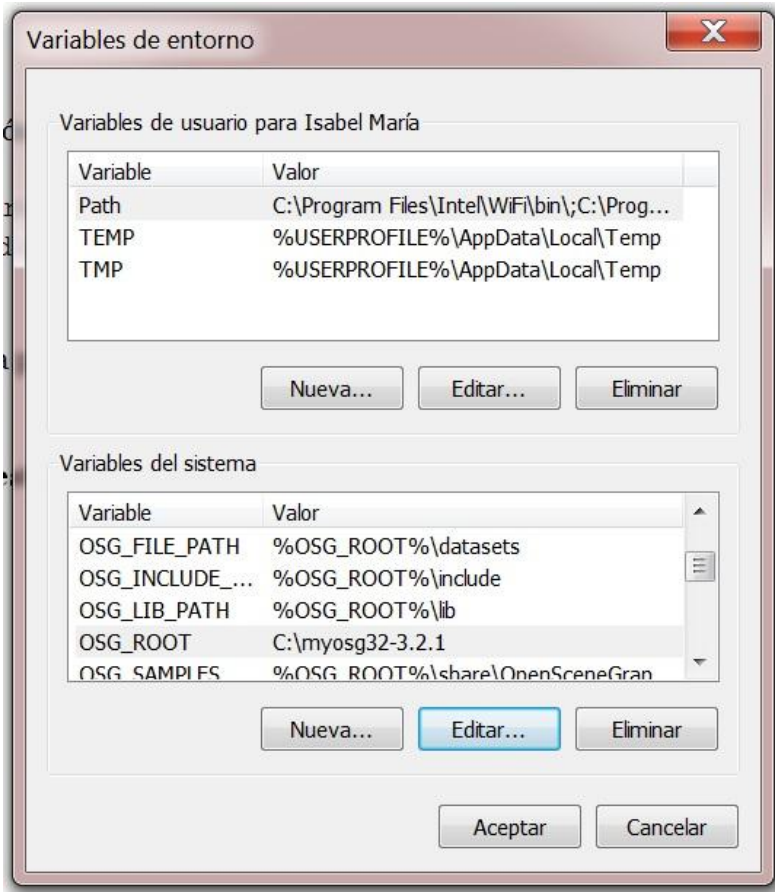


Figura 43. Variables de entorno

VARIABLE DEL SISTEMA	RUTA
OSG_ROOT	BINARIOS (C:\myosg32-3.2.1)
OSG_BIN_PATH	%OSG_ROOT%\bin
OSG_INCLUDE_PATH	%OSG_ROOT%\include
OSG_LIB_PATH	%OSG_ROOT%\lib
OSG_SAMPLES_PATH	%OSG_ROOT%\share\OpenSceneGraph\bin
OSG_FILE_PATH	%OSG_ROOT%\datasets

VARIABLE DE USUARIO	RUTA	OBSERVACIONES
Path	%OSG_BIN_PATH%	Para acceder a aplicaciones
Path	%OSG_SAMPLES_PATH%	Para acceder a ejemplos osg

NOTA: Tras configurar las variables de entorno, reiniciar Visual Studio, si estaba abierto.

5. Comprobación de que osg está instalado

El paso anterior deja el sistema listo para poder ejecutar comandos de osg. La forma de comprobarlo es por ejemplo ejecutar el visor de osg, `osgviewer` en un terminal *Símbolo de Sistema* y visualizar algún archivo osg de ejemplo:

```
C:\>osgviewer cow.osg
```

Si aparece la imagen siguiente, OpenSceneGraph estará instalado en el sistema con éxito.

Figura 44. Visor de OpenSceneGraph, osgviewer.

Anexo II: Preparación del entorno de desarrollo en Visual Studio 2010

En este anexo se explicará paso a paso cómo crear un proyecto para trabajar con OpenScenGraph en el entorno de desarrollo Visual Studio 2010.

1. Abrir Visual Studio 2010

2. Proyecto Nuevo-> Aplicación de consola de 32 bits. Proyecto vacío. Escoger nombre del proyecto y ruta donde guardarlo.

3. Configuración del proyecto:

   Para este paso es necesario haber configurado previamente las variables de entorno de OpenSceneGraph. Ver Anexo I.

   Puesto que se va a trabajar con osg, hay que indicar dónde están los archivos binarios de osg. Para ello hacer clic derecho sobre el nombre del proyecto en el explorador de soluciones y seleccionar `Propiedades`. En el cuadro de diálogo que aparece:

   a. `C/C++ -> Directorios de inclusión adicionales` y escribir `$(OSG_INCLUDE_PATH)`.
   b. `Vinculador (Linker) -> General ->Directorios de biblioteca adicionales` y escribir `$(OSG_LIB_PATH)`.
   c. `Vinculador (Linker) -> Entrada -> Dependencias adicionales` y añadir los nombres de las librerías que precise el proyecto, las cuales están en la carpeta `lib`.

   Esta configuración es idéntica tanto en modo `Release` como en modo `Debug`, pero las librerías del vinculador son distintas según el modo. Así, mientras en modo `Release` los nombres de las librerías son del tipo *nombre.lib*, en el modo `Debug` las librerías son del tipo *nombre**d**.lib*

4. Escribir los ficheros fuente del proyecto.

5. Este paso es **muy importante**. Una vez configurado el proyecto, y antes de compilarlo, se debe escoger el tipo de compilación: `Release` o `Debug`, así como la plataforma para la que se va a compilar: `Win32` para un computador de 32 bits o `x64` para un computador de 64 bits.

Anexo III: Instrucciones de uso del conversor BRep - STL

1. Instalación

El conversor se presenta en dos versiones: una para 32 bits y la otra para 64 bits. Para instalar el programa basta con descomprimir el archivo zip correspondiente a la versión que se necesite en el directorio que se desee.

La estructura de archivos es la siguiente:

```
conversorBREP-STL_inst32/
     |_ testBRP/
     |_ testSTL/
     |_ BRP2STL.exe
     |_ glut32.dll
     |_ Leeme.txt
     |_ osg100-osg.dll
     |_ osg100-osgDB.dll
     |_ osg100-osgGA.dll
     |_ osg100-osgText.dll
     |_ osg100-osgUtil.dll
     |_ osg100-osgViewer.dll
     |_ osgdb_stl.dll
     |_ ot20-OpenThreads.dll
     |_ zlib1.dll
```

`BRP2STL.exe` es el ejecutable del programa.
`testBRP` y `testSTL` son la carpeta donde se encuentran ficheros `.brp` de ejemplo y la carpeta donde dejar los resultados de la conversión en `.stl`.
`Leeme.txt` describe detalles del programa y forma de uso.
El resto de los archivos, con extensión `.dll`, son librerías que necesita el programa para poder ejecutarse. Con ellas, no es necesario tener instalado OpenSceneGraph.

2. Llamando al conversor

El programa está hecho como una aplicación de consola, así que para ejecutarlo hay que abrir una ventana de Simbolo de Sistema de Windows y situarse en el directorio donde estén instalados los ficheros del conversor. Por ejemplo, si la carpeta está en C:

```
C:\>cd conversorBREP-STL_inst32
```

Una vez dentro, la forma de llamar al programa es:

```
conversorBREP-STL.exe -src nombre_brp [-dst nombre_stl]
```

`-src` indica que a continuación se escribe la ruta del fichero BRep.
`-dst` es opcional e indica que a continuación se escribe la ruta del fichero STL resultante. Si la carpeta de destino indicada no existe, el fichero STL no se generará.

Si no se indica el fichero stl de salida, se creará uno del mismo nombre y ruta que el fichero `.brp` de entrada, pero con extensión `.stl`.

Se generará el fichero `.stl` en el directorio destino indicado con `-dst` y se abrirá un visor con el objeto definido, ya teselado en triángulos.

Ejemplo de uso:

```
BRP2STL.exe -src testBRP/casa.brp -dst testSTL/casa.stl
```

Se verá algo como esto:

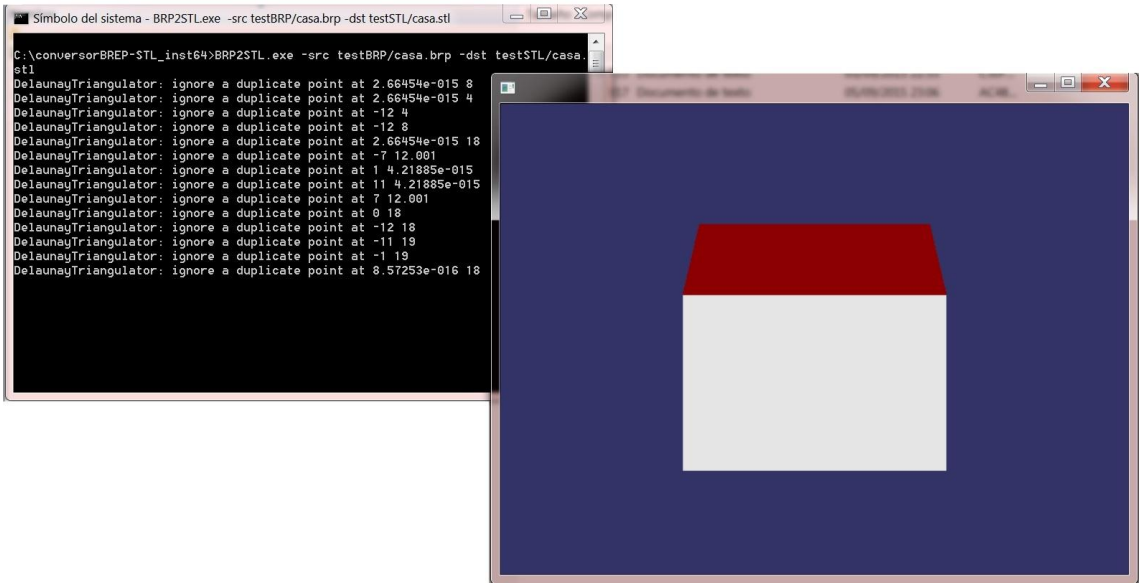


Figura 45. Ejecución del conversor BRep-STL

3. Manejo del visor de OpenSceneGraph

El programa muestra el objeto teselado en una ventana que contiene el visor de OpenSceneGraph. Es un visor básico en el que se pueden hacer las siguientes acciones:

TECLA	ACCIÓN
espacio	Retornar el objeto a su posición inicial.
l	Encemder/apagar la luz
w	Alternar entre tres modos de visión del objeto: sólido, alámbrico y puntos.
rueda del ratón hacia delante	Zoom de acercamiento
rueda del ratón hacia atrás	Hace un zoom de alejamiento
escape	Salir del programa