

Informática Gráfica.

Sesión 4: Modelos de Objetos. Mallas indexadas..

Carlos Ureña, Sept 2025.

Dept. Lenguajes y Sistemas Informáticos.
Universidad de Granada.

Índice

Modelos geométricos	3
Modelos de fronteras: mallas de polígonos.	16
Representación de modelos de fronteras.	44
Problemas	80

Sección 1.

Modelos geométricos.

-
1. Modelos geométricos. Introducción.

Subsección 1.1.

Modelos geométricos. Introducción.

Modelos geométricos formales

Un **modelo geométrico** es un modelo matemático abstracto que sirve para representar un objeto geométrico que existe en un espacio afín E (2D o 3D).

- Los modelos deben permitir la visualización computacional de los objetos que representan.
- Los más usados hoy en día son los **modelos de fronteras**: son estructuras de datos que representan la frontera del objeto de forma exacta o aproximada, normalmente mediante **mallas de triángulos**, pero hay otras posibilidades.
- Un modelo alternativo son los **modelos de volúmenes**.
- Últimamente se usan bastante modelos basados en las **funciones de distancia con signo** (*signed distance functions*) o SDFs. Cada objeto se representa con un algoritmo que calcula la distancia (o una cota) desde cualquier punto al objeto.

En la asignatura nos centramos en las mallas de triángulos.

Representaciones computacionales

El modelo basado en subconjuntos de puntos del espacio:

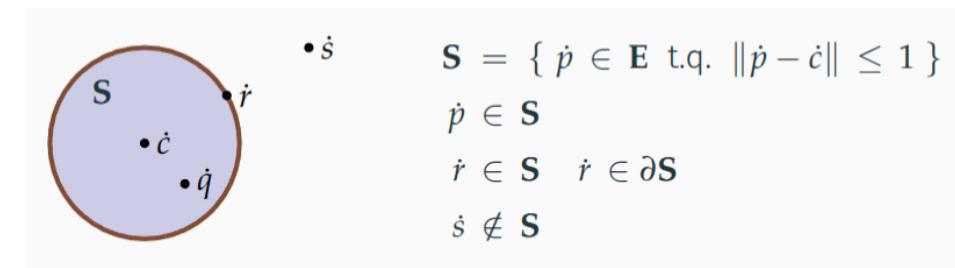
- Es un modelo válido para cualquier geometría (es el modelo **más general** posible).
- En muchos casos, **no se puede representar en la memoria** (finita, discreta) de un ordenador.

Hay representaciones aproximadas que usan una cantidad finita de memoria (**modelos geométricos computacionales**):

- **Enumeración espacial**: se partitiona el espacio en celdas o **voxels**, cada una se clasifica como interior o exterior al objeto.
- **Modelos de fronteras**: se representa la frontera (la superficie) en lugar de todo el interior, para ello se usan conjuntos finitos de polígonos planos o **caras**
- Otros tipos de modelos: algoritmos, SDFs, redes neuronales, etc..

Los conjuntos de puntos

Los modelos geométricos matemáticos abstractos más generales posibles son los **subconjuntos de puntos** de un espacio afín (típicamente 2D o 3D), por ejemplo, una esfera:

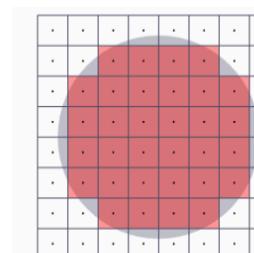


Cada subconjunto o **región** S es **cerrado** (incluye a su propia **superficie** o **frontera**, ∂S), además:

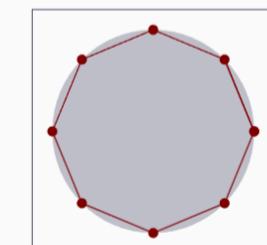
- Es **acotado** (no tiene extensión infinita),
- Su superficie es diferenciable (**plana** al menos a escala muy pequeña)

Ejemplo 2D de los modelos aproximados

Las dos formas computacionales de representar objetos son aproximadamente iguales al modelo ideal (un subconjunto de puntos), con un error que disminuye al aumentar la cantidad de memoria usada (la precisión o resolución).



Enumeración espacial

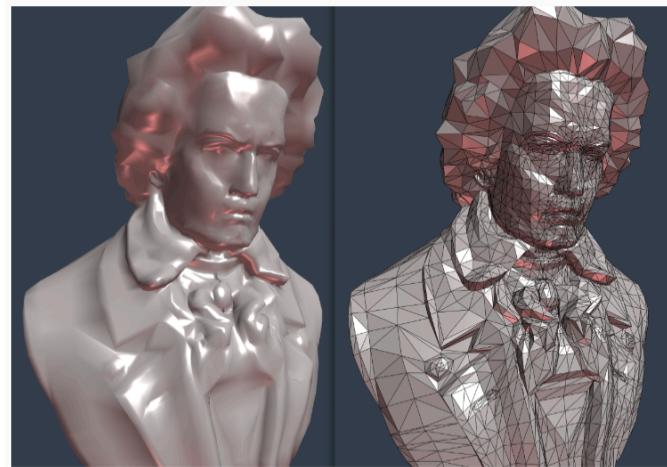


Modelos de fronteras

- **Modelos de fronteras**: usados en la mayoría de las aplicaciones.
- **Enumeración espacial**: muy útiles en aplicaciones específicas

Ejemplos de modelos de fronteras 3D (1/2)

Ejemplo de una **mallas de polígonos** (de las prácticas). A la izquierda el modelo con iluminación, a la derecha vemos las caras que forman el modelo:



Sesión 4: Modelos de Objetos. Mallas indexadas.

Created 2025-12-10

Page 9 / 91.

1. Modelos geométricos..
1.1. Modelos geométricos. Introducción..

Otros modelos de fronteras: nubes de puntos

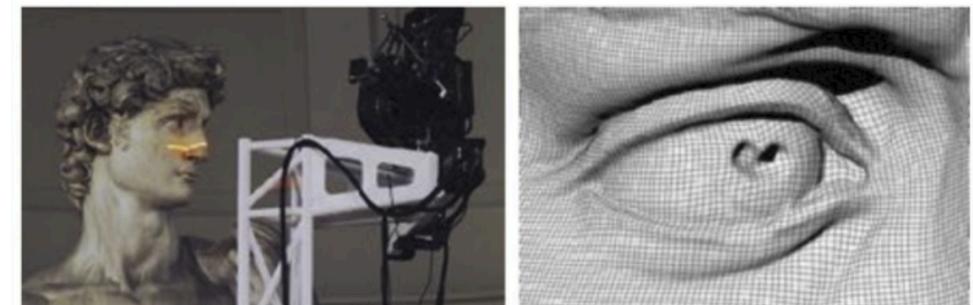
Una **nube de puntos** es un conjunto finito de puntos en el espacio 3D, cada uno con una posición y un color, que aproximan la superficie del objeto. No hay conectividad entre los puntos. Se suelen obtener a partir de escaneado 3D (LIDAR, o fotogrametría)



Imagen de la Web de Autodesk: www.autodesk.com/eu/solutions/point-clouds

Ejemplos de modelos de fronteras 3D (2/2)

Los modelos de fronteras (a muy alta resolución) permiten representar fielmente casi cualquier objeto real:



Escaneo 3D del David de Miguel Angel en Florencia en 1999.
Marc Levoy et al. The Digital Michelangelo Project: accademia.stanford.edu/mich

Sesión 4: Modelos de Objetos. Mallas indexadas.

Created 2025-12-10

Page 10 / 91.

Sesión 4: Modelos de Objetos. Mallas indexadas.

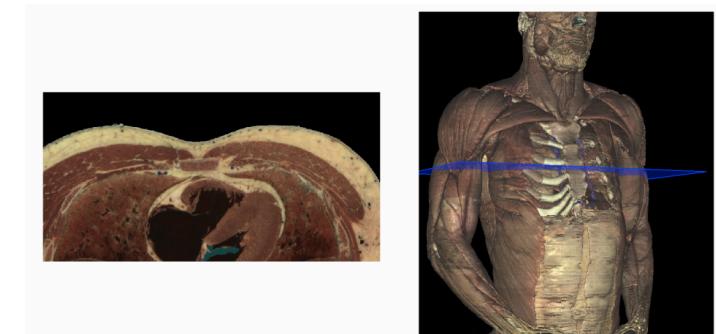
Created 2025-12-10

Page 10 / 91.

1. Modelos geométricos..
1.1. Modelos geométricos. Introducción..

Enumeración espacial 3D

Las **modelos volumétricos** se usan en aplicaciones donde interesa todo el volumen del objeto (p.ej. en la **Tomografía Axial Computerizada**, TAC, para Medicina y Arqueología, o en Geología y Climatología)



Captura de pantalla del software VH Dissector de Toltech:
www.toltech.net/anatomy-software/solutions/vh-dissector-for-medical-education

Modelos algorítmicos o procedurales

Se basan en modelar un objeto O mediante la implementación en el código de la aplicación de una función asociada al mismo (que se evalúa en cualquier punto p del espacio), sin usar una estructura de datos. Hay varias opciones:

- **Función de pertenencia de O :** la función produce un valor lógico, `true` si p está dentro de O o `false` si está fuera. La visualización usando estos modelos puede ser muy costosa en tiempo e inexacta.
- **Función de distancia con signo de O (Signed Distancia Function, o SDF):** la función devuelve la distancia más corta desde p a la frontera de O (negativa si está dentro de O , positiva si está fuera).

Se pueden usar para visualizar el objeto:

- Directamente, usando técnicas basadas en **ray-tracing**, o bien
- Indirectamente: mediante conversión a modelo de fronteras o volúmenes seguida de **rasterización**.

Modelos complejos basados en SDFs

En la actualidad las SDFs se usan para representar escenas y objetos complejos:

- Las SDFs constituyen la única forma de representar y visualizar (con un grado alto de exactitud visual) algunos tipos de objetos matemáticos, como los **fractales** (frontera no diferenciable en ningún punto). Se usan métodos numéricos iterativos (usualmente lentos).
- Se usan técnicas basadas en IA para entrenar **redes neuronales** usando imágenes o videos de escenarios reales, de forma que
 - ▶ La red neuronal es capaz de aproximar una SDF que codifica el modelo geométrico (y opcionalmente el de aspecto) del escenario real. La SDF se obtiene como la combinación de las SDFs de objetos simples (como por ejemplo elipsoides).
 - ▶ La visualización de la escena se puede hacer en tiempo real usando Ray-Tracing, incluso en escenarios muy complejos.

Modelos algorítmicos: ejemplo sencillo (esfera)

Por ejemplo, para una esfera O con centro en c y radio r , usando C++

Función de pertenencia: (F_O) se compara $\|\mathbf{p} - \mathbf{c}\|^2$ con r^2 :

$$F_O(\mathbf{p}) = \begin{cases} \text{true} & : \text{si } (\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) \leq r^2 \\ \text{false} & : \text{en otro caso} \end{cases}$$

```
bool pertenece_esfera( vec3 & p, vec3 & c, float r )
{
    return dot( p-c, p-c ) <= r*r ;
}
```

Función de distancia con signo: (SDF_O) se restan distancia y radio:

$$SDF_O(\mathbf{p}) = \|\mathbf{p} - \mathbf{c}\| - r$$

```
float sdf_esfera( vec3 & p, vec3 & c, float r )
{
    return (p-c).length() - r ;
}
```

Sección 2. Modelos de fronteras: mallas de polígonos.

1. Introducción
2. Elementos y adyacencia.
3. Atributos de vértices.

Mallas de polígonos.

Una **malla de polígonos** (*polygon Mesh*) es un conjunto de puntos de un espacio afín que forman **caras** (*faces*) planas, usualmente adyacentes entre ellas, y que aproxima la frontera de un objeto en el espacio 3D

- El término **objeto** designa un conjunto de puntos como los descritos antes (de extensión finita, continuo), todos ellos en un mismo espacio afín.
- Una **cara** es un conjunto de puntos en un plano de dicho espacio afín, delimitados por un polígono.
- Las mallas aproxima una superficie, la cual
 - ▶ encierra completamente una región del espacio (el objeto tiene volumen), o bien
 - ▶ constituye en si misma el objeto, que tiene volumen nulo.

Las primeras son mallas ***cerradas*** y las segundas ***abiertas***.

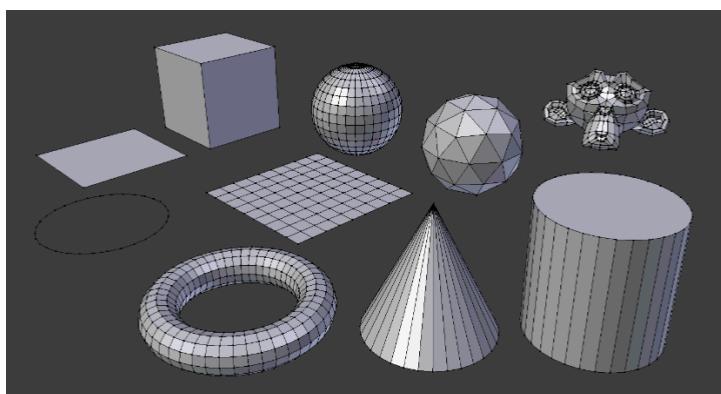
Subsección 2.1.

Introducción

2. Modelos de fronteras: mallas de polígonos..
2.1. Introducción.

Ejemplos de mallas

Aquí vemos varios ejemplos de mallas de polígonos (excepto la circunferencia que no lo es). Algunas son cerradas y otras abiertas:



Catálogo de objetos predefinidos de la aplicación Blender para modelado 3D:
docs.blender.org/manual/en/latest/modeling/meshes/primitives.html

Sesión 4: Modelos de Objetos. Mallas indexadas.

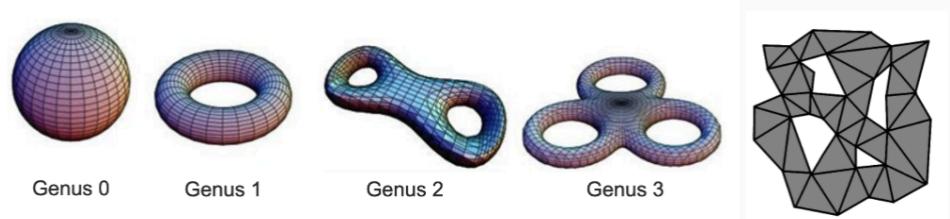
Created 2025-12-10

Page 18 / 91.

2. Modelos de fronteras: mallas de polígonos..
2.1. Introducción.

Características de las mallas

- Las mallas cerradas pueden tener cualquier **género topológico** (*genus*), aquí vemos mallas de género 0, 1, 2 y 3.
- Las mallas abiertas pueden tener huecos entre los polígonos:



Izquierda: Rudiger Westermann (Univ. Munich) - Computer Graphics Course slides
slideplayer.com/slide/4642205/

Derecha: [Stackoverflow](#)

Elementos de las mallas: vértices

Un **vértice (vertex)** es un par formado por un **punto** del espacio afín (en el extremo de alguna una arista), y un **valor entero único** (entre 0 y $n - 1$, donde n es el número de vértices de la malla).

Subsección 2.2.

Elementos y adyacencia.

- Al punto lo llamamos la **posición** del vértice.
- Al entero lo llamamos **índice** del vértice.
- Dos vértices distintos (con distinto índice) pueden tener la misma posición.
- Usar estos índices tiene estas ventajas:
 - ▶ Permiten expresar la **topología** de una malla independientemente de su **geometría**.
 - ▶ Facilita construir representaciones computacionales de las mallas con ciertas propiedades.

Elementos de las mallas: caras

Una **cara (face)** contiene un conjunto de puntos coplanares que están delimitados por un único polígono plano. Se determina por una secuencia ordenada de índices de vértices que forman dicho polígono.

- En la secuencia de índices, cada vértice comparte una arista con el siguiente (y el último con el primero). En esta secuencia
 - ▶ es indiferente cual índice es el primero, y
 - ▶ en principio, es indiferente en que sentido se recorren los vértices (solo hay dos posibilidades).
- Dos caras distintas no pueden tener asociado el mismo conjunto de índices, ni siquiera con distinto orden o empezando en distintos vértices.

Elementos de las mallas: aristas. Representación de mallas.

Una **arista (edge)** contiene el conjunto de puntos en un lado del polígono que delimita una cara, puntos que forman un segmento de recta. Se determina por un par único de índices de vértices.

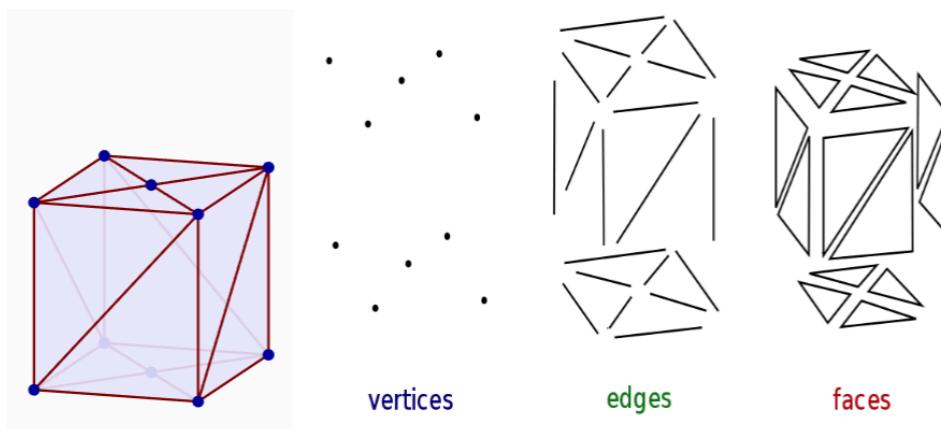
- Los dos índices de vértice de una arista no pueden coincidir.
- El orden en el que aparecen los índices en el par es irrelevante (las aristas no están orientadas).
- Dos aristas distintas no pueden tener el mismo par de índices de vértice, ni siquiera en distinto orden.

Esto implica que **una malla viene determinada por**:

- la secuencia $\{p_0, p_1, \dots, p_{n-1}\}$ de **posiciones** de sus n vértices.
- la secuencia de **caras**, cada una de ellas representada como una secuencia de k índices de vértice: $\{i_0, \dots, i_{k-1}\}$ (k puede ser distinto en cada cara).

Vértices, caras y aristas

Elementos de una malla que forman la frontera de un paralelepípedo:



Imágenes de la derecha tomadas de: [Wikipedia: Polygon Mesh](#).

Geometría y topología de las mallas

Una malla tiene una geometría y una topología:

Geometría: conjunto de puntos que están en alguna cara (eso incluye los puntos que están en alguna arista y las posiciones de los vértices).

Topología: conjunto de relaciones de adyacencia entre vértices, aristas y caras (sin tener en cuenta la geometría, es decir, considerando únicamente los índices de los vértices).

Esta definición permite que dos mallas distintas:

- Tengan la misma topología pero distinta geometría (p.ej., partimos de una malla y cambiamos las posiciones de sus vértices, pero mantenemos las adyacencias).
- Tengan la misma geometría pero distinta topología (p.ej., partimos de una malla con caras de cuatro aristas y dividimos cada cara en dos caras triángulares coplanares).

Adyacencia entre elementos de una malla

En una malla existen relaciones binarias de adyacencia entre estos elementos:

- Un vértice en el extremo de una arista es adyacente a la arista (por tanto, toda arista es adyacente a exactamente dos vértices).
- Una arista y una cara son adyacentes si la arista forma parte del polígono que delimita la cara.
- Dos vértices son adyacentes si hay una arista adyacente a ambos.
- Dos caras son adyacentes si hay una arista adyacente a ambas.
- Un vértice y una cara son adyacentes si hay una arista adyacente a ambos.

Puesto que los vértices están numerados, las relaciones de adyacencia se pueden expresar en términos de los índices de los vértices.

Características de las 2-variedades

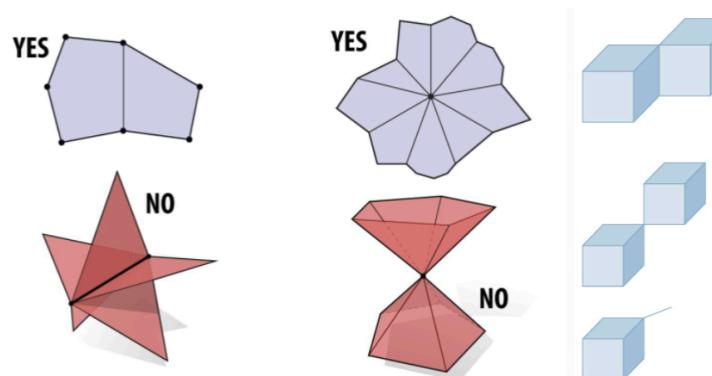
Vamos a usar exclusivamente mallas que son una **2-variedad (2-manifold)**, esto implica que:

- Un vértice siempre **es adyacente a dos aristas como mínimo** (no hay vértices aislados).
- Una arista siempre **es adyacente a una o a dos caras** (no hay aristas aisladas, ni aristas adyacentes a 3 o más caras).
- Todas las caras adyacentes a un vértice **se pueden ordenar en una secuencia en la cual cada cara es adyacente a la siguiente**.

Estas propiedades aseguran, entre otras cosas, que se pueden asignar ciertos atributos a cada vértice de forma única (por ejemplo, normales y coordenadas de textura), ya que el entorno de un punto de la superficie siempre es *equivalente a un plano*.

Ejemplos de mallas que no son 2-variedades

Varias mallas, dos representan 2-variedades y el resto no:



Izquierda: Carnegie Mellon Computer Graphics Course: slides (fall 2018):
15462.cs.cmu.edu/spring2018/lecture/meshes

Derecha: J.F. Hughes at al.: Computer Graphics: Principles and Practice (3rd ed.)

Conversión en 2-variedad

La topología de una malla que no es una 2-variedad puede modificarse para que lo sea, manteniendo la geometría:

- Se puede conseguir **replicando vértices**, es decir, añadiendo nuevos vértices con índices distintos en la misma posición de otros vértices ya existentes

En el ejemplo de la transparencia anterior, los dos conos unidos por el ápice se separan al insertar dos ápices en la misma posición, un ápice por cono.

- Esto puede implicar a veces también **replicar aristas**

En el ejemplo de la transparencia anterior: los dos cubos unidos con una arista en común se separan duplicando los dos vértices de dicha arista y la propia arista.

Aristas y vértices de frontera

Una arista es una **arista de frontera** (o de borde) (*boundary edge* o *border edge*) si es adyacente a una única cara.

Un vértice es un **vértice de frontera** si es adyacente a alguna arista de frontera.

A la derecha vemos una malla con diversos vértices y aristas de frontera (en amarillo)

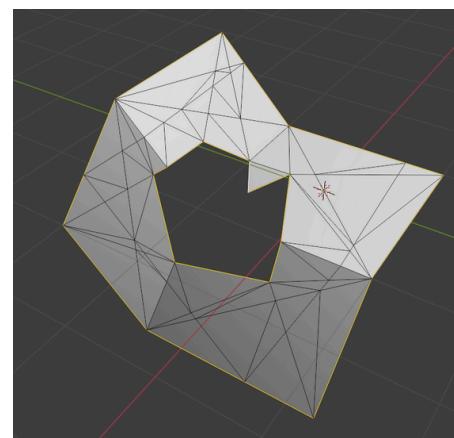


Imagen de StackOverflow:
stackoverflow.com/questions/78359671

Mallas abiertas y cerradas

Una malla es **cerrada** si y solo si no tiene aristas de frontera (todas las aristas son adyacentes a exactamente dos caras). Es *topológicamente equivalente* a una esfera (a la derecha).

Una malla es **abierta** si tiene al menos una cara de frontera (a la izquierda, las aristas de frontera están en rojo)

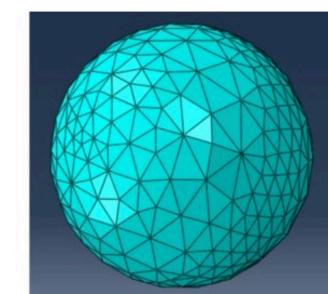
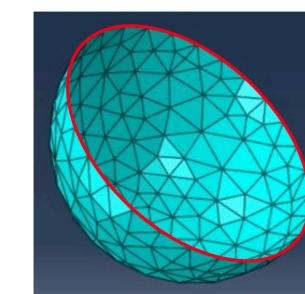


Imagen de Fangtao Yang: [Researchgate](#)

Lados y orientación de una cara.

En una cara se pueden identificar dos lados distintos, en función del orden en que aparecen los vértices en la secuencia de índices que define la cara:

- **Lado horario** es el lado en el cual se ven los vértices en orden de las agujas del reloj.
- **Lado antihorario** es el lado en el cual se ven los vértices en sentido contrario al de las agujas del reloj.

Cuando una cara se visualiza en una imagen **desde un punto de vista concreto**, se verá únicamente uno de los dos lados:

- Si se ve el lado horario, se dice que la cara **aparece orientada en sentido horario**.
- Si se ve el lado antihorario, se dice que la cara **aparece orientada en sentido antihorario**

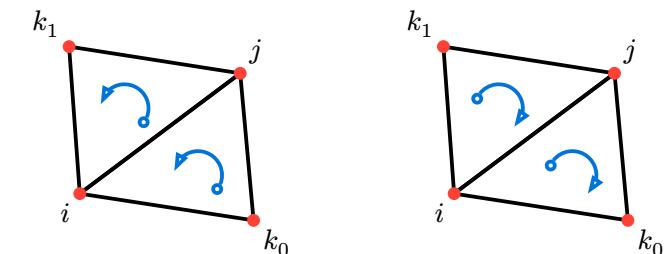
Cribado de caras traseras

Las APIs de rasterización suelen hacer una clasificación de las caras al visualizar una imagen. Según la orientación con que aparecen, se clasifican en **delanteras o traseras**.

- Esto permite el **cribado de caras traseras (*back face culling*)**, consiste en **visualizar en una imagen únicamente las caras delanteras**.
- Esto sirve para visualizar una malla cerrada opaca desde un punto de vista exterior a la malla, sin perder tiempo en rasterizar las caras que no se van a ver con seguridad, ya que están en la parte trasera (no visible) del objeto (las caras traseras).
- Este comportamiento se puede activar, desactivar, o configurar (se puede establecer que las caras delanteras son las de orientación horaria o antihoraria).
- En Godot el cribado está activado por defecto, y las caras delanteras son las que **aparecen en sentido horario**. Se puede configurar para cada malla.

Orientación coherente de las mallas.

Una malla tendrá **orientación coherente** si dadas dos caras adyacentes comparando una arista entre los vértices i y j , entonces, en una cara j es el siguiente a i , y en la otra cara i es el siguiente a j .



En un ejemplo de dos caras coplanares, es legal que se vean ambas caras orientadas en sentido horario, o ambas en sentido antihorario, pero no cada una con una orientación.

Marco de referencia de la malla.

La posición de cada vértice se representa en el ordenador por sus coordenadas respecto de un marco de referencia cartesiano \mathcal{R} único

- A dicho marco de referencia se le denomina **marco de referencia local de la malla**
- El i -ésimo vértice (en el punto p_i) tiene coordenadas $\mathbf{c}_i = (x_i, y_i, z_i, 1)$ en el marco \mathcal{R} , es decir:

$$p_i = \mathcal{R}c_i = \mathcal{R}(x_i, y_i, z_i, 1)^T$$

- A la tupla c_i se le denomina las **coordenadas locales** del vértice i -ésimo.
- No se suele almacenar en memoria la componente w ya que siempre es 1, aunque a veces se podría hacer para acortar algo el tiempo de procesamiento (a costa de usar más memoria).

Atributos de las mallas

Como modelos de objetos reales, las mallas suelen incluir más información geométrica o del aspecto del objeto:

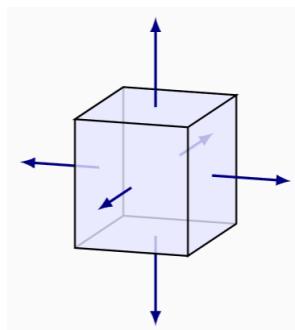
- **Normales (normals):** vectores de longitud unidad
 - ▶ **Normales de caras:** vector unitario perpendicular a cada cara, de longitud unidad, apuntando al exterior de la malla si es una malla cerrada. Precalculado a partir del polígono.
 - ▶ **Normales de vértices:** vector unitario perpendicular al plano tangente a la superficie en la posición del vértice.
- **Colores:** ternas (usualmente RGB) con tres valores entre 0 y 1.
 - ▶ **Colores de caras:** útil cuando cada cara representa un trozo de superficie de color homogéneo.
 - ▶ **Colores de vértices:** color de la superficie en cada vértice (la superficie varía de color de forma continua entre vértices).
- Otros atributos: coords. de textura, vectores tangente y bitangente, etc...

Subsección 2.3.

Atributos de vértices.

Normales de caras

Pueden ser útiles cuando el objeto que se modela con la malla está realmente compuesto de caras planas (p.ej., un cubo), o bien cuando se quiere hacer *sombreado plano*:



Para un polígono (con dos aristas \vec{a}, \vec{b} , vectores distintos, no nulos), su normal \vec{n} se define como:

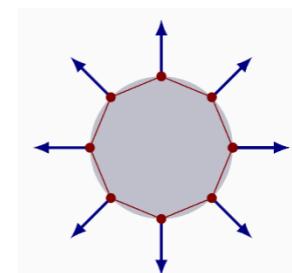
$$\vec{n} = \frac{\vec{m}}{\|\vec{m}\|} \quad \text{donde} \quad \vec{m} = \vec{a} \times \vec{b}$$

En estos casos la normal se puede precalcular y almacenar en la malla para lograr eficiencia en tiempo de visualización.

Normales de vértices para superficies suaves:

Tienen sentido cuando la malla aproxima una superficie curvada:

- A veces la superficie original es conocida, y las normales se definen fácilmente (p.ej. una esfera).
- Si la superficie original es desconocida, las normales se pueden definir exclusivamente usando la malla



Para un vértice (con k caras adyacentes) su normal \vec{n} se define como:

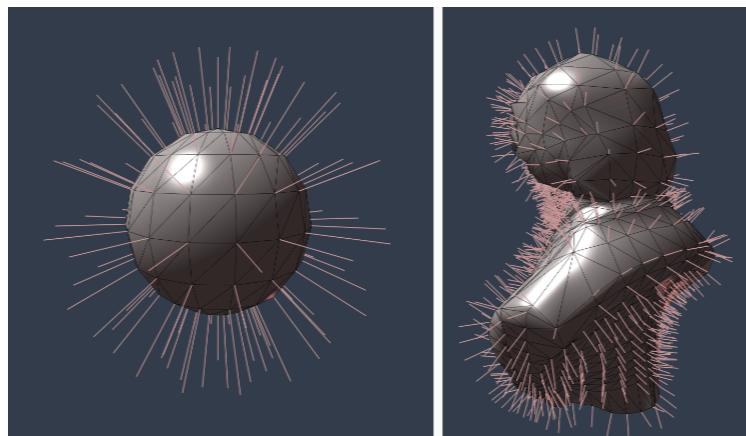
$$\vec{n} = \frac{\vec{s}}{\|\vec{s}\|} \quad \text{donde} \quad \vec{s} = \sum_{i=0}^{k-1} \vec{m}_i$$

donde $\vec{m}_0, \vec{m}_1, \dots, \vec{m}_{k-1}$ son las normales de las caras adyacentes al vértice.

Las coordenadas de estas normales también se pueden precalcular y almacenar.

Ejemplos de normales de vértices calculadas

Aquí vemos visualizadas las normales de una esfera de baja resolución (calculadas analíticamente), y de una malla arbitraria (calculadas promediando normales de caras):



Colores de vértices

En algunos casos, es conveniente asignar colores RGB a los vértices. La utilidad más frecuente de esto es hacer interpolación de color en las caras durante la visualización:

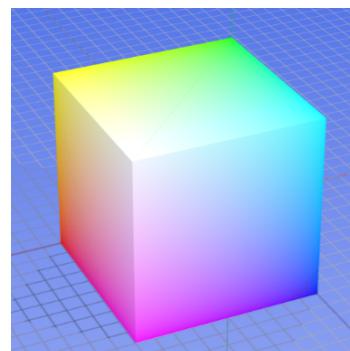
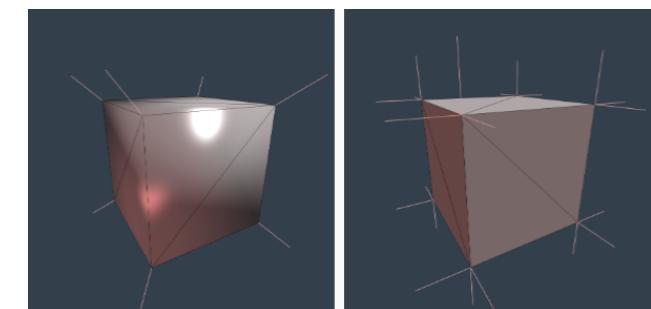


Imagen: en.wikipedia.org/wiki/File:RGB_color_solid_cube.png (Wikimedia Commons)

Discontinuidades de la normal

Algunos objetos reales presentan aristas o vértices donde la normal es discontinua (p.ej. un cubo), en ese caso promediar normales es mala idea, y es necesario replicar vértices y aristas (y después promediar):



El cubo de la izquierda tiene 8 vértices y el de la derecha 24 vértices. La iluminación es correcta a la derecha. El mismo problema puede aparecer con las coordenadas de textura, o los colores.

Sección 3. Representación de modelos de fronteras.

1. Triángulos aislados.
2. Tiras de triángulos.
3. Mallas indexadas de triángulos.
4. Aristas aladas
5. Formatos para archivos con mallas indexadas.

Representación en memoria

En esta sección veremos distintas formas de representar las mallas en la memoria de un ordenador:

- **Triángulos aislados, tiras de triángulos:** no representan explícitamente la topología.
- **Mallas indexadas:** lo más común, representan explícitamente la topología.
- **Aristas aladas:** extensión de las mallas indexadas para eficiencia en tiempo.

Godot está diseñado para visualizar directamente los triángulos aislados, las tiras de triángulos y las mallas indexadas, pero no las aristas aladas.

Por simplicidad, nos restringimos a **caras triángulares**, que es lo más común en la inmensa mayoría de las aplicaciones.

Subsección 3.1.
Triángulos aislados.

Tabla de triángulos aislados.

La más simple es usar una lista o tabla de **triángulos aislados**. La malla se representa como un vector o lista con tres entradas (tres variables de tipo **Vector3** o **Vector2**) para cada triángulo:

Malla TA (n triángulos)		
	x_0	y_0
0	x_0	y_0
1	x_1	y_1
2	x_2	y_2
3	x_3	y_3
4	x_4	y_4
5	x_5	y_5
:	:	
$3n - 3$	x_{3n-3}	y_{3n-3}
$3n - 2$	x_{3n-2}	y_{3n-2}
$3n - 1$	x_{3n-1}	y_{3n-1}

- Para cada triángulo se almacenan las coordenadas locales de cada uno de sus tres vértices (9 valores flotantes en total).
- La tabla se puede almacenar en memoria con todas las coordenadas contiguas.
- En total, incluye $9n$ valores flotantes.

Representación en Godot

La malla se representa como un array empaquetado de Godot con tres entradas (tres variables de tipo **Vector2** o **Vector3**) para cada triángulo:

```
var posiciones : PackedVector3Array = [
    Vector3(0,0,0), Vector3(1,0,0), Vector3(0,1,0), # Triángulo 1
    Vector3(1,0,0), Vector3(1,1,0), Vector3(0,1,0), # Triángulo 2
    ...
]
```

También se puede usar un array normal de Godot, pero entonces debe convertirse a un array empaquetado antes de añadir las tablas a un objeto **Mesh** para agregarlo a un nodo:

```
var posiciones : Array[Vector3] = [
    Vector3(0,0,0), Vector3(1,0,0), Vector3(0,1,0), # Triángulo 1
    Vector3(1,0,0), Vector3(1,1,0), Vector3(0,1,0), # Triángulo 2
    ...
]
var posiciones_em := PackedVector3Array( posiciones ) # ctor específico
```

Valoración.

Esta representación es **poco eficiente en tiempo y memoria**:

- Si un vértice es adyacente a k triángulos, sus coordenadas aparecen repetidas k veces en la tabla y se procesan k veces al visualizar.
- En una malla típica que representa una rejilla de triángulos, los vértices internos (la mayoría) son adyacentes a 6 triángulos, es decir, cada tupla aparece casi 6 veces como media.

Además, **no hay información explícita sobre la topología** de la malla:

- La topología se puede calcular comparando coordenadas de vértices, pero tendría una complejidad en tiempo cuadrática con el número de vértices y es poco robusto.

En algunos casos muy particulares, podría ser útil (objetos realmente compuestos de muchos triángulos realmente aislados, objetos muy sencillos).

Subsección 3.2.
Tiras de triángulos.

Tiras de triángulos: motivación y características

Las representación en memoria usando **tiras de triángulos (triangle strip)** pretende reducir la memoria y el tiempo que necesitan la representación de triángulos aislados:

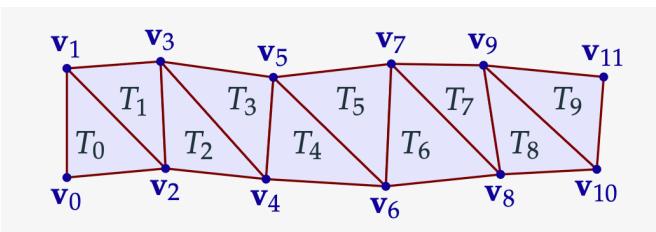
- Para conseguir esto, esta representación reduce el número de veces que aparecen replicadas unas coordenadas en memoria.
- Como consecuencia, se reduce el tiempo de procesamiento.

Sin embargo, esta representación

- No evita totalmente las redundancias (se siguen repitiendo coordenadas de vértices, aunque menos).
- Tampoco incluye información explícita sobre la topología de la malla.

Tiras de triángulos en una malla.

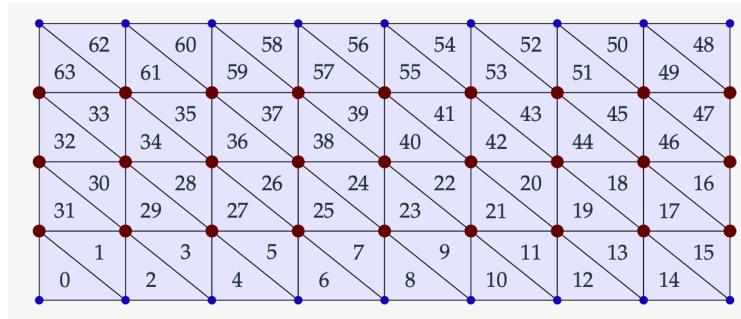
Podemos identificar una parte de una malla como una **tira de triángulos**: cada triángulo T_{i+1} en la secuencia es adyacente al anterior T_i , con lo cual T_{i+1} comparte con T_i una arista y sus dos vértices v_i y v_{i+1} , vértices cuyas coordenadas no tienen que ser repetidas en memoria:



- Cada tira de n triángulos necesita $n + 2$ tuplas de coordenadas de vértices (tres para el primer triángulo y después una más por cada triángulo adicional)
- Se almacena una tabla que en la i -ésima entrada almacena las coordenadas del i -ésimo vértice.

Tiras de triángulos para mallas no simples

En la mayoría de los casos, las tiras obligan a repetir algunas coordenadas de vértices (aunque en mucho menor grado que los triángulos aislados). Ejemplo de una tira en zig-zag:



- Las coords. de los vértices en rojo (grandes) se repiten dos veces.
- Las de los vértices en azul (pequeños) aparecen una sola vez.

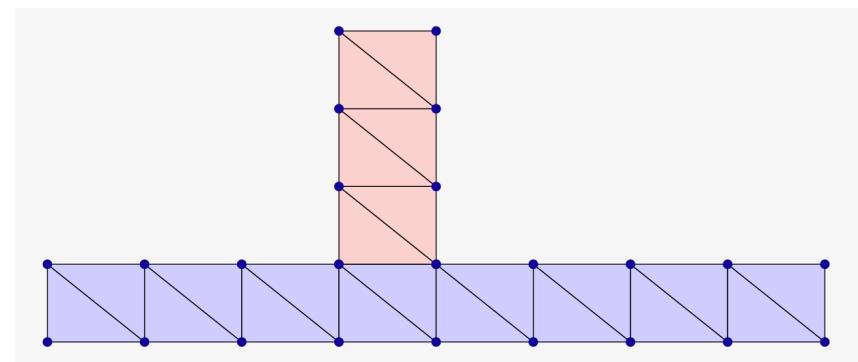
Representación en memoria

Una malla es una estructura con varias tiras. La tira número i (con n_i triángulos) es un array con $n_i + 2$ celdas, en cada una están las coordenadas maestras de un vértice.

Tira 0 (n_0 triángulos)	Tira 1 (n_1 triángulos)	Tira 2 (n_2 triángulos)
$x_0 \quad y_0 \quad z_0$ $x_1 \quad y_1 \quad z_1$ $x_2 \quad y_2 \quad z_2$ $x_3 \quad y_3 \quad z_3$ $x_4 \quad y_4 \quad z_4$ $\vdots \quad \vdots \quad \vdots$ $x_{n_0} \quad y_{n_0} \quad z_{n_0}$ $x_{n_0+1} \quad y_{n_0+1} \quad z_{n_0+1}$ $x_{n_0+2} \quad y_{n_0+2} \quad z_{n_0+2}$	$x_0 \quad y_0 \quad z_0$ $x_1 \quad y_1 \quad z_1$ $x_2 \quad y_2 \quad z_2$ $x_3 \quad y_3 \quad z_3$ $x_4 \quad y_4 \quad z_4$ $\vdots \quad \vdots \quad \vdots$ $x_{n_1} \quad y_{n_1} \quad z_{n_1}$ $x_{n_1+1} \quad y_{n_1+1} \quad z_{n_1+1}$ $x_{n_1+2} \quad y_{n_1+2} \quad z_{n_1+2}$	$x_0 \quad y_0 \quad z_0$ $\vdots \quad \vdots \quad \vdots$ $x_{n_2+2} \quad y_{n_2+2} \quad z_{n_2+2}$

Mallas con varias tiras

En algunos casos, es inevitable tener que recurrir a más de una tira para una única malla:



Por este motivo la implementación de una malla con tiras debe prever más de una tira en la misma malla.

Tiras de triángulos: valoración

La mejora de las tiras frente a los triángulos aislados es que usan menos memoria, sin embargo, tiene estos inconvenientes:

- Al requerir probablemente más de una tira de triángulos, la representación es algo más compleja.
- Se necesitan algoritmos (complejos) para calcular las tiras a partir de una malla representada de alguna otra forma. Se intenta optimizar de forma que el número de coordenadas a almacenar sea el menor posible.
- El numero promedio de veces que se repite cada coordenada en memoria es prácticamente siempre superior a la 1, y cercano a 2.
- Esta representación tampoco incorpora información explícita sobre la conectividad.

Tiras de triángulos: Implementación

Se pueden representar con un array de arrays, cada uno de los segundos con una tira:

```
var posiciones : Array[Array] = [
  [ # Tira 1
    Vector3(0,0,0), Vector3(1,0,0), Vector3(0,1,0),
    Vector3(1,1,0), Vector3(2,0,0), ....
  ],
  [ # Tira 2
    Vector3(3,0,0), Vector3(4,0,0), Vector3(3,1,0),
    Vector3(4,1,0), ....
  ],
  .... ## otras tiras..
]
```

Para crear un nodo con una de estas mallas, es necesario construir un objeto **Mesh** por cada tira, y crear un nodo de tipo **MeshInstance3D** para cada uno de ellos. Los nodos **MeshInstance3D** se añaden como hijos del nodo que contiene la malla completa.

Mallas Indexadas.

Para solucionar los problemas de uso de memoria y tiempo de procesamiento de las soluciones anteriores, se puede usar una estructura con dos tablas:

- **Tabla de vértices:** tiene una entrada por cada vértice, incluye sus coordenadas
- **Tabla de triángulos:** tiene una entrada por triángulo, incluye los índices de sus tres vértices en la tabla anterior.

En esta solución:

- **No se repiten coordenadas de vértices:** se ahorra memoria y se puede visualizar sin repetir cálculos (se repiten índices enteros).
- **Hay información explícita de la topología (conectividad):** se almacenan explícitamente los vértice adyacentes a un triángulo y se pueden calcular fácilmente el resto de adyacencias.

Subsección 3.3.
Mallas indexadas de triángulos.

Estructura de datos

La tabla de triángulos (para n triángulos), almacena un total de $3n$ índices de vértices (enteros sin signo), y la de vértices $3m$ valores reales:

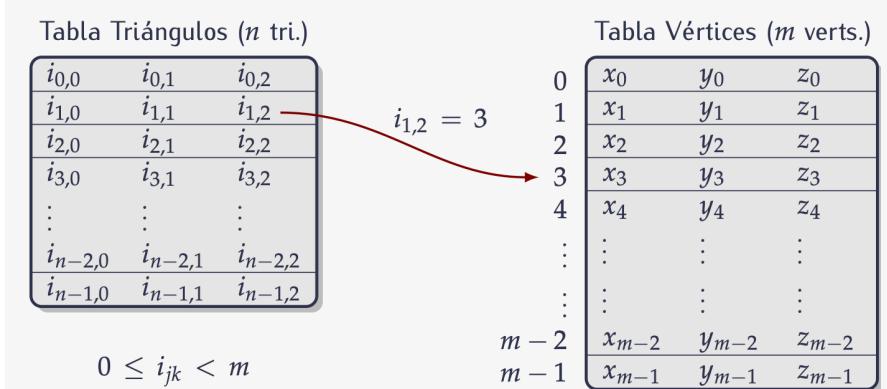


Tabla Triángulos (n tri.)			Tabla Vértices (m verts.)		
$i_{0,0}$	$i_{0,1}$	$i_{0,2}$	x_0	y_0	z_0
$i_{1,0}$	$i_{1,1}$	$i_{1,2}$	x_1	y_1	z_1
$i_{2,0}$	$i_{2,1}$	$i_{2,2}$	x_2	y_2	z_2
$i_{3,0}$	$i_{3,1}$	$i_{3,2}$	x_3	y_3	z_3
\vdots	\vdots	\vdots	x_4	y_4	z_4
$i_{n-2,0}$	$i_{n-2,1}$	$i_{n-2,2}$	\vdots	\vdots	\vdots
$i_{n-1,0}$	$i_{n-1,1}$	$i_{n-1,2}$	x_{m-2}	y_{m-2}	z_{m-2}
			x_{m-1}	y_{m-1}	z_{m-1}

$0 \leq i_{jk} < m$

Implementación de las mallas indexadas

En Godot se pueden usar arrays de Godot , es útil se queremos manipular la malla mediante algoritmos:

```
var posiciones : Array[Vector3] = [
    Vector3(0,0,0), Vector3(1,0,0), Vector3(0,1,0), ....
]
var triangulos : Array[Vector3i] = [
    Vector3i(0,1,2), Vector3i(1,3,2), Vector3i(1,4,3), ....
]
```

Para añadir la malla a un nodo, habrá que convertir estos array a empaquetados, se puede hacer así:

```
var pos_em : PackedVector3Array( posiciones ) # ctor específico
var tri_em : PackedInt32Array([]) # inicialmente vacía

for t in triangulos: # añadimos índices con un bucle
    tri_em.append( t[0] ); tri_em.append( t[1] ); tri_em.append( t[2] )
```

Tiras de triángulos indexadas

Es posible representar una malla como una tabla de vértices y varias tiras de triángulos. Cada tira almacena índices de vértices en lugar de coordenadas de vértices.

- Las coordenadas no se repiten en memoria.
- Se repiten en memoria los índices de vértices, pero menos veces que con tabla de vértices y triángulos.
- El modelado usando tiras es más complejo.

Se pueden añadir a un nodo usando el tipo de primitiva tiras y además usando una tabla de índices.

- Las coordenadas de vértice se envían y se procesan una sola vez
- Los índices de vértices se envían repetidos, pero solo un par de veces de media aprox.

Motivación

La estructura de malla indexada permite, por ejemplo, consultar con tiempo en $O(1)$ si un vértice es adyacente a un triángulo, pero:

- Para consultar si dos vértices son adyacentes, hay que buscar en la tabla de triángulos si los vértices aparecen contiguos en alguno: esto requiere un tiempo en $O(n_t)$.
- No se guarda información de las aristas. Las consultas relativas a aristas se resuelven también en $O(n_t)$.
- En general, las consultas sobre adyacencia son costosas en tiempo.

Para poder reducir los tiempos de cálculo de adyacencia a $O(1)$, se puede usar más memoria de la estrictamente necesaria para la malla indexada. Veremos la estructura de **aristas aladas** (para mallas que encierran un volumen, es decir: siempre hay **dos caras adyacentes a una arista**, no necesariamente triangulares)

Estructura de aristas aladas: tabla de aristas.

Una malla se puede codificar usando una tabla de vértices (**tver**) (similar a la de las mallas indexadas), mas una tabla de aristas (**tari**). Esta última:

- Tiene una entrada por cada arista, con dos índices:

- ▶ **vi** = índice de **vértice inicial**
- ▶ **vf** = índice de **vértice final**

(es indiferente cual se selecciona como inicial y cual como final).

- Tambien tiene (cada arista es adyacente a dos triángulos):

- ▶ **ti** = índice del **triángulo a la izquierda**
- ▶ **td** = índice del **triángulo a la derecha**

(izquierda y derecha entendidos según se recorre la arista en el sentido que va desde vértice inicial al final)

- Esto permite consultas en O(1) sobre adyacencia **arista-vértice** y **arista-tríangulo**.

Aristas siguiente y anterior

Además de los datos anteriores, se guarda, para cada arista, los índices de otras cuatro adyacentes a ella.

- Si se recorren las aristas del triángulo de la izquierda aparecerá la arista en cuestión entre otras dos, cuyos índices se guardan en la tabla:

- ▶ **aai** = índice de la **arista anterior**
- ▶ **asi** = índice de la **arista siguiente**

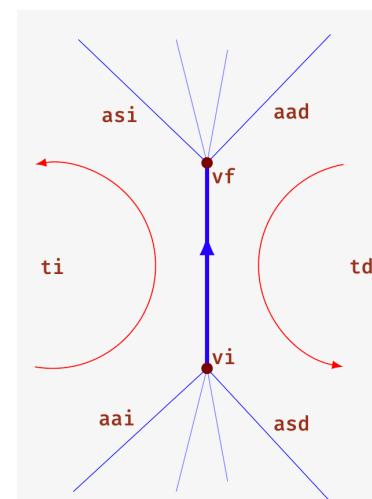
(el recorrido de las aristas se hace en sentido anti-horario cuando se observa el triángulo desde el exterior de la malla).

- Igualmente, se guardan los dos índices de arista anterior y siguiente relativas al recorrido anti-horario del triángulo de la derecha:

- ▶ **aad** = índice de la **arista anterior** (derecha)
- ▶ **asd** = índice de la **arista siguiente** (derecha)

Índices asociados a una arista

Índices (valores naturales) en la entrada correspondiente a la arista vertical en una malla (no necesariamente de triángulos):



Uso de las aristas siguiente y anterior.

El hecho de almacenar las aristas siguiente y anterior permite hacer recorridos por las entradas de la tabla de aristas siguiendo esos índices:

- Dada una arista y un triángulo adyacente (el izquierdo o el derecho), se pueden obtener estas listas:
 - ▶ aristas adyacentes al triángulo.
 - ▶ vértices adyacentes al triángulo
- Dada una arista y un vértice adyacente (el inicial o el final), se pueden obtener estas listas:
 - ▶ aristas que inciden en el vértice (esto permite resolver fácilmente adyacencias **arista-arista**)
 - ▶ triángulos adyacentes al vértice.

Con toda esta información (los 8 valores), se puede resolver directamente cualquier adyacencia que involucre una arista al menos (consultando su entrada). Aun no podemos resolver el resto.

Tablas adicionales. Uso.

Para hacer todas las consultas en $O(1)$, añadimos **taver** y **tatri**:

- **taver** = tabla de **aristas de vértice**: para cada vértice, almacenamos el índice de una arista adyacente cualquiera:
 - ▶ dado un vértice, permite recuperar todas las aristas, vértices y triángulos adyacentes.
 - ▶ por tanto, permite consultas de adyacencia **vértice-vértice** y **vértice-triángulo**.
- **tatri** = tabla de **aristas de triángulo**: para cada triángulo, se almacena el índice de una arista cualquiera adyacente:
 - ▶ dado un triángulo, permite recuperar todas las aristas, vértices y triángulos adyacentes,
 - ▶ por tanto, permite consultas de adyacencia **triángulo-triángulo** y **vértice-triángulo** (esta última se puede hacer de dos formas).

Subsección 3.5.

Formatos para archivos con mallas indexadas.

Formato PLY

El formato PLY fue diseñado por Greg Turk y otros en la Univ. de Stanford a mediados de los 90. Codifica una malla indexada en un archivo ASCII o binario (usaremos la versión ASCII). Tiene tres partes:

Cabecera: describe los atributos presentes y su formato, se indica el número de vértices y caras, ocupa varias líneas.

Tabla de vértices: un vértice por línea, se indican sus coordenadas X, Y y Z (flotantes) en ASCII, separadas por espacios.

Tabla de caras: una cara por línea, se indica el número de vértices de la cara, y después los índices de los vértices de la cara (comenzando en cero para el primer vértice de la tabla de vértices).

El formato es extensible de forma que un archivo puede incluir otros atributos (p.ej., colores de vértices). Su simplicidad hace fácil usarlo.

Ejemplo de archivo PLY: cabecera

En esta cabecera de ejemplo se indica que:

- hay 8 elementos **vertex** y 6 elementos **face** (caras),
- cada vértice tiene 3 **propiedades**, tipo (**float**) llamadas **x**, **y**, **z**,
- cada cara es una lista, primero su longitud (**uchar**) y luego una serie de enteros (**int**).

```
ply
format ascii 1.0
comment Archivo de ejemplo del formato PLY (8 vertices y 6 caras)
element vertex 8
property float x
property float y
property float z
element face 6
property list uchar int vertex_index
end_header
```

Ejemplo de archivo PLY: tablas de vértices y caras

A continuación vendría la tabla de vértices, con 8 líneas (una por vértice):

```
0.0 0.0 0.0
0.0 0.0 1.0
0.0 1.0 0.0
0.0 1.0 1.0
1.0 0.0 0.0
...
...
```

Y finalmente la tabla de caras, con 6 líneas (una por cara), en cada línea se indica el número de vértices de la cara seguido de los correspondientes índices (en este ejemplo todas son triángulos, por lo que el primer número es siempre 3):

```
3 0 1 2
3 0 3 1
3 4 0 1
3 4 1 5
...
...
```

Tablas de vértices, normales y coordenadas de textura

En un archivo **.obj**, cada línea puede comenzar con: **v** para vértices (coordenadas X,Y,Z), **vn** para normales (componentes X,Y,Z), **vt** para coordenadas de textura (componentes U,V). Por ejemplo:

```
v 0.123 0.234 0.345 1.0
v ...
...
vt 0.500 1 [0]
vt ...
...
vn 0.707 0.000 0.707
vn ...
...
```

Los vértices, normales y coordenadas de textura se numeran empezando en 0, **de forma independiente** entre ellos, es decir la numeración de los vértices es independiente de la de normales y de la de coordenadas de textura.

El formato OBJ

El formato **OBJ** (también llamado **Wavefront OBJ**) fue ideado por la empresa **Wavefront**, y se usa bastante hoy en día, es parecido a PLY, pero **con las normales y coordenadas de textura indexadas**:

- Incluye una tabla de vértices y una tabla de triángulos, igual que PLY
- Además, incluye tablas normales y coordenadas de textura. A diferencia de PLY, su tamaño no tiene porque coincidir con el de la tabla de vértices.
- En cada cara, cada vértice se representa por un índice de sus coordenadas de posición, y opcionalmente, otros índices independientes para su normal y sus coordenadas de textura.
- Permite añadir información de materiales y texturas (en archivos externos, con extensión **.mtl** y formato MTL, *Materials Template Library*).
- Godot puede importar archivos OBJ directamente.
- Librerías para carga disponibles: *Assimp*, *TinyOBJLoader*, etc.

Tabla de caras en formato OBJ

En un archivo **.obj**, cada cara se representa con una línea que comienza con **f**, seguida de una serie de grupos separados por espacios, uno por cada vértice de la cara. Cada grupo tiene la forma:

v_idx/vt_idx/vn_idx

donde **v_idx** es el índice del vértice en la tabla de vértices, **vt_idx** es el índice de las coordenadas de textura en la tabla de coordenadas de textura, y **vn_idx** es el índice de la normal en la tabla de normales. Los índices comienzan en 1.

Ejemplo de una cara triangular con vértices con índices 1, 8 y 16, coordenadas de textura con índices 34, 45 y 56, y normales con índices 0, 0 y 1:

```
f 1/34/0 8/45/0 16/56/1
```

Si no se usan coordenadas de textura o normales, los grupos pueden tener las formas **v_idx//vn_idx** o simplemente **v_idx**.

Formato OBJ: valoración

La ventaja frente a PLY (malla indexada de triángulos) es una mayor flexibilidad, lo que permite mayor eficiencia en memoria:

- Dos vértices en posiciones distintas pueden compartir normal (por ejemplo, una malla plana puede tener una única normal, en lugar de tantas como vértices)
- Un vértice único puede tener distintas coords. de textura o distintas normales en distintas caras, no es necesario replicarlo (por ejemplo, un cubo puede tener 8 vértices y solo 6 normales).

La principal desventaja es que las APIs de rasterización no pueden visualizar directamente este tipo de tablas, así que es necesario convertirlas a una malla indexada de triángulos, replicando normales y coordenadas de textura.

Tabla de aristas

En una malla indexada podría ser conveniente (para ganar tiempo de procesamiento en ciertas aplicaciones) almacenar explícitamente las aristas (ahora esa info. está implícita en la tabla de triángulos). Se puede hacer usando un **tabla de aristas**:

- Contiene una entrada por cada arista.
- En cada entrada hay dos índices (naturales) de vértices (los dos vértices en los extremos de la arista).
- El orden de los vértices en cada entrada es irrelevante, pero las aristas no deben estar duplicadas.

La disponibilidad de esta tabla permite, por ejemplo, dibujar en modo alambre con begin/end sin repetir dos veces el dibujo de aristas adyacentes a dos triángulos.

Los formatos glTF y GLB

El formato **glTF** (*GL Transmission Format*) es un formato estándar abierto para la transmisión y almacenamiento de modelos 3D y escenas. Fue desarrollado por el Khronos Group y está diseñado para ser eficiente en términos de tamaño de archivo y velocidad de carga.

- **glTF** utiliza una estructura basada en JSON para describir la geometría, materiales, texturas, animaciones y otros aspectos de un modelo 3D.
- El formato **GLB** es una versión binaria de glTF que empaqueta todos los datos en un solo archivo binario, lo que facilita su distribución y uso en aplicaciones web y móviles.
- Ambos formatos son ampliamente compatibles con motores gráficos modernos, incluyendo Godot, Unity y Unreal Engine.
- glTF/GLB soporta mallas indexadas, normales, coordenadas de textura, materiales PBR (*Physically Based Rendering*) y animaciones esqueléticas.

Problema: comparación de eficiencia en memoria (1/2)

Problema 4.1:

Supongamos que queremos codificar una esfera de radio $1/2$ y centro en el origen de dos formas:

- Por enumeración espacial, dividiendo el cubo que engloba a la esfera en celdas, de forma que haya k celdas por lado del cubo, todas ellas son cubos de $1/k$ de ancho. Cada celda ocupa un bit de memoria (si su centro está en la esfera, se guarda un 1, en otro caso un 0).
- Usando un modelo de fronteras (una malla indexada de triángulos), en el cual se usa una rejilla de triángulos y aristas que siguen los meridianos y paralelos, habiendo en cada meridiano y en cada paralelo un total de k vértices (se guarda únicamente la tabla de vértices y la de triángulos).

(continua en la siguiente transparencia)

Problema: comparación de eficiencia en memoria (2/2)

Problema 4.1 (continuación):

Asumiendo que un **float** y un **int** ocupan 4 bytes cada uno, contesta a estas cuestiones:

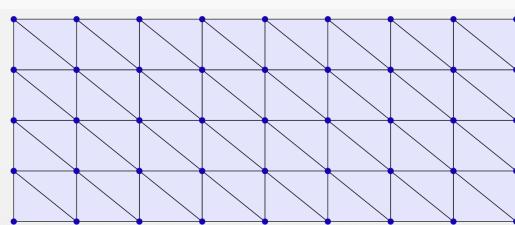
- Expresa el tamaño de ambas representaciones en bytes como una función de k .
- Suponiendo que $k = 16$ calcula cuantos KB de memoria ocupa cada estructura.
- Haz lo mismo asumiendo ahora que $k = 1024$ (expresa los resultados en MB)

Compara los tamaños de ambas representaciones en ambos casos ($k = 16$ y $k = 1024$).

Problema: uso de memoria en mallas indexadas (1/2)

Problema 4.2:

Considera una malla indexada (tabla de vértices y caras, esta última con índices de vértices) con topología de rejilla como la de la figura, en la cual hay n columnas de pares de triángulos y m filas (es decir, hay $n + 1$ filas de vértices y $m + 1$ columnas de vértices, con $n, m > 0$, en el ejemplo concreto de la figura, $n = 8$ y $m = 4$).



(continua en la siguiente transparencia)

Problema: uso de memoria en mallas indexadas (2/2)

Problema 4.2 (continuación):

En relación a este tipo de mallas, responde a estas dos cuestiones:

- Supongamos que un **float** ocupa 4 bytes (igual a un **int**) ¿ que tamaño en memoria ocupa la malla completa, en bytes ? (tener en cuenta únicamente el tamaño de la tabla de vértices y triángulos, suponiendo que se almacenan usando los tipos **float** e **int**, respectivamente). Expresa el tamaño como una función de m y n .
- Escribe el tamaño en KB suponiendo que $m = n = 128$.
- Supongamos que m y n son ambos grandes (es decir, asumimos que $1/n$ y $1/m$ son prácticamente 0). deduce que relación hay entre el número de caras n_C y el número de vértices n_V en este tipo de mallas.

Problema: uso de memoria en tiras y mallas indexadas (1/2)

Problema 4.3:

Imagina de nuevo una malla como la del problema anterior, supongamos que usamos una representación como tiras de triángulos, de forma que cada fila de triángulos (con $2n$ triángulos) se almacena en una tira, habiendo un total de m tiras.

La tabla de punteros a tiras tiene un entero (el número de tiras) y m punteros, cada puntero suponemos que tiene 8 bytes de tamaño. De nuevo, asume que las coordenadas son de tipo **float** (4 bytes).

Responde a estas cuestiones:

(continua en la siguiente transparencia)

Problema: uso de memoria en tiras y mallas indexadas (2/2)

Problema 4.3 (continuación):

- Indica que cantidad de memoria ocupa esta representación, en estos dos casos:
 - Como función de n y m , en bytes.
 - Suponiendo $m = n = 128$, en KB.
- Para m y n grandes (es decir, cuando $1/n$ y $1/m$ son casi nulos), describe que relación hay entre el tamaño en memoria de la malla indexada del problema anterior y el tamaño de la malla almacenada como tiras de triángulos.
- Si suponemos que la transformación de cada vértice se hace en un tiempo constante igual a la unidad, describe que relación hay entre los tiempos de procesamiento de vértices para esta malla cuando se representa como una malla indexada y como tiras de triángulos.

Problema: número de vértices, aristas y caras

Problema 4.4:

Supongamos una malla cerrada, simplemente conexa (topológicamente equivalente a una esfera), cuyas caras son triángulos y cuyas aristas son todas adyacentes a exactamente dos caras (la malla es un *poliedro simplemente conexo de caras triangulares*). Considera el número de vértices n_V , el número de aristas n_A y el número de caras n_C en este tipo de mallas.

Demuestra que cualquiera de esos números determina a los otros dos, en concreto, demuestra que se cumplen estas dos igualdades:

$$n_A = 3(n_V - 2)$$

$$n_C = 2(n_V - 2)$$

(nótese que, al igual que en el problema anterior, sigue siendo cierto que el número de caras es aproximadamente el doble que el de vértices).

Problema: creación de la tabla de aristas

Problema 4.5:

En una malla indexada, queremos añadir a la estructura de datos una tabla de aristas. Será un vector **ari**, que en cada entrada tendrá una tupla de tipo **Vector2i** (contiene dos **int**) con los índices en la tabla de vértices de los dos vértices en los extremos de la arista. El orden en el que aparecen los vértices en una arista es indiferente, pero cada arista debe aparecer una sola vez.

Escribe el código de una función GDScript para crear y calcular la tabla de aristas a partir de la tabla de triángulos. Intenta encontrar una solución con la mínima complejidad en tiempo y memoria posible. Suponer que el número de vértices adyacentes a uno cualquiera de ellos es como mucho un valor constante $k > 0$, valor que no depende del número total de vértices, que llamamos n .

(continua en la transparencia siguiente)

Problema: creación de la tabla de aristas

Problema 4.5 (continuación):

Considerar dos casos:

- (a) Los triángulos se dan con orientación *no coherente*: esto quiere decir que si un triángulo está formado por los vértices i, j, k , estos tres índices pueden aparecer en cualquier orden en la correspondiente entrada de la tabla de triángulos. Además, no sabemos si la malla es cerrada o no.
- (b) Los triángulos se dan con orientación *coherente*: esto quiere decir que si dos triángulos comparten una arista entre los vértices i y j , entonces en uno de los triángulos la arista aparece como (i, j) y en el otro aparece como (j, i) . Además, asumimos que la malla es *cerrada*, es decir, que cada arista es compartida por exactamente dos triángulos.

Problema: cálculo del área de una malla indexada

Problema 4.6:

Escribe el pseudo-código de la función para calcular el área total de una malla indexada de triángulos, a partir de la tabla de vértices y de triángulos. Será una función GDScript que acepta ambas tablas (arrays de `Vector3` y de `Vector3i`) y devuelve el área.

Fin de transparencias.