

# Informática Gráfica.

## Sesión 2: El *engine* Godot. Mallas..

Carlos Ureña, Sept 2025.  
Dept. Lenguajes y Sistemas Informáticos.  
Universidad de Granada.

## Índice

Introducción a Godot .....	3
Mallas en Godot. ....	26
Problemas .....	89

### 1. Introducción a Godot.

## Funcionalidad y características de Godot

Godot es un IDE (entorno integrado de desarrollo, *integrated development environment*) que permite:

- **Desarrollar aplicaciones gráficas** 2D y 3D interactivas, como videojuegos, simulaciones, visualizaciones, etc.
- **Ejecutar y depurar** esas aplicaciones desde Godot.
- **Generar archivos ejecutables** independientes con la aplicación creada, archivos que pueden ser ejecutados en diversas plataformas.

Sus características principales son:

- **Código abierto**
- **IDE Multiplataforma:** el IDE Godot se puede ejecutar en Linux, Windows, macOS, incluso en Web (aunque en Web con limitaciones)
- **Genera aplicaciones multiplataforma:** genera aplicaciones nativas independientes para Windows, macOS, Linux, Android, iOS, y aplicaciones Web.

- 
- Sección 1.
- ### Introducción a Godot
1. El lenguaje de programación GDScript
  2. La jerarquía de clases de Godot.
  3. El bucle principal de Godot.

## Elementos de Godot

**Editor:** aplicación tipo IDE con herramientas para crear y organizar los recursos del proyecto, diseñar escenas, programar scripts, etc... También permite **ejecutar** y **depurar** las aplicación en desarrollo.

**Proyecto:** conjunto de escenas, nodos, scripts y recursos asociados a una aplicación en desarrollo.

**Escenas:** una escena es una estructura jerárquica (un **árbol** llamado **árbol de escena**) de **nodos** que representan objetos y elementos de la aplicación en desarrollo.

**Nodos:** elementos de las escenas, cada uno es de una **clase** de Godot.

**Scripts:** código que define el comportamiento definido por el programador para los nodos y escenas. Se pueden usar los lenguajes orientados a objetos **GDScript** (similar a Python) o **C#**, y también su propio **lenguaje visual** (*visual script*).

**Recursos:** archivos de imágenes, audios, videos, modelos 3D, etc... que se usan para desarrollar una aplicación.

## Características de GDScript

GDScript es un lenguaje de programación interpretado, de alto nivel, orientado a objetos, diseñado específicamente para Godot.

Sus características principales son:

- Sintaxis similar a Python: se usa **indentación** para definir bloques, sin punto y coma al final de las líneas (excepto si se quieren unir dos sentencias en una línea).
- Las variables pueden tener asociado un tipo conocido o no.
- Es **orientado a objetos**: incluye clases, herencia, y polimorfismo.
- Integración con el editor de Godot (permite crear y manipular nodos y escenas fácilmente)
- Gestión automática de memoria mediante conteo de referencias

### Subsección 1.1.

#### El lenguaje de programación GDScript

## Variables: declaración y tipo

GDScript permite tipos estáticos y dinámicos. Las variables pueden:

- Declararse con un **tipo explícito**: la variable no puede cambiar de tipo en su tiempo de vida.
 

```
var x : float = 10.0 # tipo 'float' (explícito)
```
- Declararse con un tipo **implícito (inferido)**: el tipo se calcula (se infiere) a partir de la expresión del valor inicial. Tampoco pueden cambiar de tipo después.
 

```
var y := 20 # tipo 'int' (es el tipo de la expresión '20').
```
- Declararse **sin tipo**: en este caso la variable puede cambiar de tipo en su tiempo de vida. Es de tipo **Variant**.
 

```
var z = 30.0 # sin tipo (inicialmente 'float', pero puede cambiar)
```

## Ejemplo de archivo fuente GDScript

Tienen la extensión `.gd` y siempre definen una clase (`MiNodo` en este caso, aunque podría ser anónima) que siempre hereda de otra (`Node3D` en este caso):

```
extends Node3D # obligatorio: indica la clase base
class_name MiNodo # opcional: si no está es una clase anónima

var velocidad : float = 100.0 # variable de instancia (tipo opc.)

func v_cuadrado() -> float :
    return velocidad * velocidad # devuelve la velocidad al cuadrado

func _init():
    pass # constructor, puede tener parámetros
        # 'pass' indica que está vacío

func _ready():
    print("Nodo listo") # imprime en la consola o terminal

func _process( delta: float ):# método de proceso por frame
    position.x += velocidad * delta # usa 'position' de Node2D
```

Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 9 / 100.

## Tipos contenedores: arrays

GDScript incluye estos tipos arrays **dinámicos** (pueden crecerse o reducirse en tiempo de ejecución):

- **Array**: contiene elementos **Variant**, es decir, de cualquier tipo.
- **Array[T]**: todos sus valores son de tipo *T* (arrays homogéneos).
- **Arrays empaquetados (packed arrays)**: arrays homogéneos con elementos contiguos en memoria, se usan para enviar datos a la GPU. Los elementos pueden ser:
  - ▶ Bytes o enteros: `PackedByteArray`, `PackedInt32Array`, `PackedInt64Array`.
  - ▶ Flotantes de simple y doble precisión: `PackedFloat32Array`, `PackedFloat64Array`.
  - ▶ Vectores de 2, 3 o 4 componentes: `PackedVector2Array`, `PackedVector3Array`, `PackedVector4Array`.
  - ▶ Colores: `PackedColorArray`.

## Tipos predefinidos en GDScript y Godot

Tipos básicos predefinidos en GDScript:

- **bool**: valores lógicos o booleanos (`true` o `false`).
- **int**: enteros (números sin parte decimal), de 64 bits.
- **float**: números reales (con parte decimal), de doble precisión (64 bits).
- **String**: cadenas de caracteres codificadas en *Unicode*.

Tipos para vectores predefinidos en GDScript:

- **Vector2**, **Vector3**, **Vector4**: tuplas con 2, 3 o 4 elementos flotantes de simple precisión (32 bits).
- **Vector2i**, **Vector3i**, **Vector4i**: tuplas con 2, 3 o 4 elementos enteros.
- **Transform2D**, **Transform3D**: matrices de transformación en 2D o 3D.

Tipos predefinidos en Godot:

- **Color**: colores en formato RGBA (rojo, verde, azul, alfa o transparencia).

Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 10 / 100.

## Ventajas del tipado estático

El uso de variables con tipo (tipado estático) es **aconsejable siempre**, ya que:

- Contribuye a **detectar más errores** en tiempo de desarrollo (en el editor) en lugar de en tiempo de ejecución, lo cual acorta el tiempo de desarrollo y disminuye la probabilidad de que el usuario final sufra errores.
- Permite **ejecutar la aplicación mucho más rápido**, ya que disminuye la sobrecarga del intérprete: no es necesario comprobar el tipo de una variable antes de realizar cualquier operación con ella.
- **Facilita la lectura y comprensión** del código: el programador puede añadir los tipos para que el lector (incluido él mismo en el futuro) entienda mejor el código.
- El uso de tipo implícito (inferido) permite **mayor expresividad** (aunque a veces disminuye la legibilidad).

## Clase **Object** y derivadas

Las clases de Godot se organizan en una **jerarquía de herencia**.

La clase **Object** es clase raíz de la jerarquía de herencia (todas las demás clases heredan directa o indirectamente de ella). Destacamos estas clases derivadas directamente de **Object**:

**Node**: clase base para todos los nodos de las escenas. Incluye nodos para visualización 2D o 3D, elementos del interfaz de usuario (*controles*), cámaras, fuentes de luz, materiales, escenas, y otros muchos.

**Viewport**: representa una zona rectangular de una ventana (o una ventana completa) donde se renderiza una escena 2D o 3D. Cada proyecto tiene un **Viewport** por defecto que no aparece explícitamente en el grafo de escena.

**MainLoop**: clase abstracta con definiciones de métodos para implementar el bucle principal de la aplicación. Godot tiene una clase derivada que implementa por defecto los métodos, llamada **SceneTree**.

---

### Subsección 1.2.

#### La jerarquía de clases de Godot.

- 1. Introducción a Godot.
- 1.2. La jerarquía de clases de Godot..

## Otras clases derivadas de **Object**

**RefCounted**: clase base para objetos en memoria dinámica gestionada automáticamente mediante la técnica de *cuenta de referencias*. Hay múltiples clases derivadas, destacamos:

**Resource**: clase base para objetos que contienen recursos de Godot. También tiene muchas clases derivadas, destacamos:

**Mesh**: clase base para mallas 2D o 3D (estructuras de datos que codifican primitivas geométricas).

**Material**: clase base para materiales que definen la apariencia visual de objetos 2D o 3D.

**Image, Texture**: clases base para imágenes en 2D o 3D, y para texturas que se aplican a objetos 3D o superficies de objetos 3D.

**Shader**: clase base para shaders (programas que se ejecutan en la GPU).

Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 14 / 100.

## Tipos de nodos

Godot incorpora muchísimos tipos de nodos. Destacamos estas clases que se derivan directamente de la clase **Node**:

**CanvasItem**: clase base para elementos que se visualizan en 2D (es decir, que se definen en el plano). Tiene dos clases derivadas:

**Control**: clase base para elementos del interfaz de usuario (botones, menús, barras de progreso, etc...).

**Node2D**: clase base para nodos que representan objetos 2D en una escena 2D (imágenes, formas geométricas, textos, etc...).

**Node3D**: clase base para nodos que representan objetos 3D en una escena 3D. Su principal clase derivada es:

**VisualInstance3D**: objetos visuales en 3D: mallas y fuentes de luz.

**Camera3D**: nodo que representa una cámara en una escena 3D, permite visualizar la escena desde diferentes ángulos y posiciones.

## Clases para mallas

Una **malla** es una estructura de datos que codifica una o varias primitivas geométricas (puntos, líneas, triángulos) y constituye la base para representar objetos 2D o 3D en gráficos por ordenador.

La clase **Mesh** es la clase base para mallas, un objeto de este tipo puede ser referenciado desde múltiples nodos (es derivada de **RefCounted**). Tiene estas clases derivadas:

**ArrayMesh**: malla definida por programador a partir de arrays de vértices y atributos (normales, colores, coordenadas de textura, etc...), se envía una vez a la GPU y permanece ahí para múltiples *frames*

**ImmediateMesh**: similar a la anterior, pero se envía a la GPU en cada *frame*.

**PrimitiveMesh**: clase base para diversas primitivas geométricas predefinidas, entre otras están: **BoxMesh**, **CylinderMesh**, **PlaneMesh**, **PointMesh**, **PrismMesh**, **SphereMesh**, **TextMesh**, **TorusMesh**.

## Nodos 3D

Entre las clases derivadas de **VisualInstance3D** destacan las dedicadas a representar instancias de mallas y luces en 3D:

**GeometryInstance3D**: tiene varias subclases, entre ellas:

**MeshInstance3D**: nodo que instancia una malla (clase **Mesh**) en una escena 3D, asignándole una posición, orientación o escala. Un mismo objeto **Mesh** puede ser instanciado múltiples veces en una escena o en escenas diferentes.

**MultimeshInstance3D**: similar al anterior, pero permite múltiples instancias.

**Light3D**: clase base para diversos tipos de fuentes de luz, a saber:

**DirectionalLight3D**: luz lejana (en una dirección)

**OmniLight3D**: luz puntual (ilumina en todas las direcciones igual)

**SpotLight3D**: luz puntual que ilumina en direcciones en un cono.

## Nodos 2D

Entre las clases derivadas de **Node2D** destacan:

**MeshInstance2D**: nodo que instancia una malla (clase **Mesh**) en una escena 2D, asignándole una posición, orientación o escala, y opcionalmente una textura. Un mismo objeto **Mesh** puede ser instanciado múltiples veces en una escena o en escenas diferentes.

**MultiMeshInstance2D**: similar al anterior, pero permite múltiples instancias.

**Camera2D**: nodo que representa una *cámara* en 2D, es decir, determina qué parte del plano 2D se visualiza en el viewport.

**Sprite2D**: un rectángulo con una textura o una parte de una textura (clase **Texture**) visible en su interior.

**AnimatedSprite2D**: similar al anterior, pero contiene más de una textura, de forma que se pueden reproducir como una animación.

## El bucle principal: métodos

La clase abstracta `MainLoop` define los métodos que permiten configurar el comportamiento de cualquier aplicación creada con Godot. Los métodos son `_initialize`, `_process` y `_finalize`.

Cuando se ejecuta la aplicación en desarrollo, se dan estos pasos:

1. Se crea una instancia de un clase derivada de `MainLoop` (típicamente `SceneTree`, pero pueden definirse otras).
2. Se invoca el método `_initialize` para inicializar esa instancia.
3. Mientras no se termine la aplicación:
  1. Se invoca el método `_process` en dicha instancia para actualizar el estado de los objetos de la aplicación y renderizar la escena.
  2. Si es necesario, se hace una espera hasta que sea el momento del siguiente frame.
4. Se invoca el método `_finalize` para liberar recursos y finalizar la aplicación.

Subsección 1.3.

### El bucle principal de Godot.

## La clase `SceneTree`

La clase `SceneTree` es una implementación concreta de `MainLoop` que se caracteriza por incluir un árbol de nodos (una escena) y gestionar:

- La creación del árbol al inicio, según el diseño creado por el usuario en el editor.
- La adición y eliminación de nodos al árbol, de forma dinámica, en tiempo de ejecución.
- La actualización y renderizado de la escena en cada frame.
- Los cálculos asociados a la simulaciones físicas.
- Las entrada de usuario (teclado, ratón, etc...).
- La ejecución de scripts asociados a nodos y escenas.
- Terminación y liberación de recursos al finalizar la aplicación.

En esta asignatura únicamente se usará `SceneTree` para el bucle principal (no se intentan crear bucles personalizados).

## Creación de nodos

En Godot el usuario (programador) puede diseñar el árbol de escena, creando los nodos que lo componen, y configurándolos por programas de dos formas:

Antes de ejecutar, en el editor

- Creando un nodo de una clase de Godot y asignándole después a ese nodo un `script` (archivo de código) que redefine uno o varios métodos de la clase `Node`,
- Creando un nodo de clase definida por el usuario, que herede directa o indirectamente de `Node`, y asignándole al nodo esa clase en el editor. Esa clase puede tener sus propios métodos específicos

En tiempo de ejecución, mediante un `script`

- Creando un nodo `h` con el método `new` de su clase y añadiéndolo al árbol como un hijo de un nodo `p` existente, mediante: `p.add_child(h)` (se usa el método `add_child` del nodo padre)

## Redefinición de métodos

El comportamiento de un nodo puede adaptarse redefiniendo métodos de `Node` u `Object` que se invocan por `SceneTree` en determinados momentos durante la ejecución:

- `_init` : es el constructor del nodo (método de `Object`), se invoca al crear un nodo para inicializar sus variables de instancia. Puede tener parámetros.
- `_enter_tree` : se invoca al añadir un nodo al árbol, cuando su padre se ha añadido, pero antes de añadir sus hijos.
- `_ready` : se invoca al añadir el nodo, después de `_enter_tree`, cuando ya se han añadido los nodos hijos.
- `_process(delta)` : se invoca antes de cada frame para actualizar el estado del nodo. El parámetro `delta` indica el tiempo (en segundos) transcurrido desde el último frame.
- `_input( event )` : invocado cuando se produce un evento de entrada (teclado, ratón, etc...). El parámetro `event` lleva información del evento.

## Introducción

En esta sección se introducen los arrays de vértices como una forma de representar conjuntos de primitivas gráficas en general, y se indica como se pueden usar para crear objetos gráficos 2D y 3D en Godot.

- La idea esencial es que una secuencia de vértices (puntos del espacio representados, como mínimo, por sus coordenadas), puede codificar polilíneas, segmentos, triángulos, polígonos, etc...
- En Godot a los objetos gráficos (2D y 3D) definidos de esta forma se les denomina de forma genérica **mallas** (*meshes*).
- Una clase particular de arrays de vértices (de mallas) son las **mallas indexadas de triángulos**, que son el tipo más común de representar las superficies de los objetos en 3D.

---

Sección 2.  
**Mallas en Godot.**

1. Tipos de primitivas
2. Atributos de vértices
3. Modos de envío de datos a la GPU
4. Representación de mallas en Godot
5. Mallas en 2D.
6. Mallas en 3D.
7. Mallas indexadas de triángulos en 3D

---

Subsección 2.1.  
**Tipos de primitivas**

## Especificación de primitivas. Tipos de primitivas.

En Godot (y resto de engines y APIs de rasterización), cada primitiva o conjunto de primitivas se especifica mediante una secuencia ordenada de coordenadas de vértices:

- Un vértice es un punto de un espacio afín 3D.
- Se representa en memoria mediante una tupla de coordenadas en algún marco de coordenadas de dicho espacio afín.
- Puede tener asociados otros valores, llamados **atributos** (p.ej. un color).

Existen tres clases de primitivas: **puntos**, **segmentos** y **triángulos**:

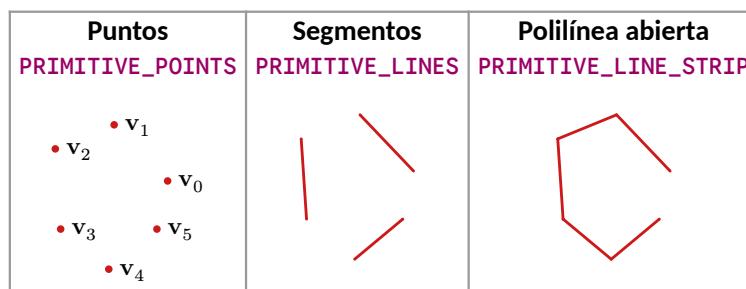
- Por tanto, además de la secuencia de vértices, es necesario tener información acerca de que tipo de primitiva representa dicha secuencia.

A cada forma de codificar primitivas en una secuencia se le llama un **tipo de primitivas**, en GDScript se definen diversas constantes para eso, del tipo enumerado **PrimitiveType**, en la clase **Mesh**.

## Primitivas de tipo puntos y segmentos.

Una secuencia de coordenadas ( $v_0, v_1, \dots, v_{n-1}$ ) pueden formar: puntos, o segmentos o polilíneas abiertas.

Aquí se ilustran los posibles tipos de primitivas (con puntos o segmentos) para una secuencia con  $n = 6$ :



## Tipos de primitivas: puntos y segmentos

Una lista de  $n$  coordenadas de vértices (con  $n \neq 1$ ) puede usarse para codificar puntos o segmentos. Más en concreto, puede codificar:

Tipo de primitiva	Descripción
<b>Mesh.PRIMITIVE_POINTS</b>	$n$ puntos aislados ( $n$ arbitrario)
<b>Mesh.PRIMITIVE_LINES</b>	$n/2$ segmentos independientes ( $n$ debe ser par)
<b>Mesh.PRIMITIVE_LINE_STRIP</b>	$n - 1$ segmentos formando una <b>polilínea abierta</b> ( $n$ debe ser mayor o igual a 2)
<b>Mesh.PRIMITIVE_TRIANGLES</b>	$n/3$ triángulos ( $n$ debe ser múltiplo de 3)
<b>Mesh.PRIMITIVE_TRIANGLE_STRIP</b>	$n - 2$ triángulos compartiendo aristas ( <b>tira de triángulos</b> ), ( $n$ debe ser mayor o igual que 3)

## Triángulos delanteros y traseros. Cribado.

Cada primitiva de tipo triángulo (también llamada **cara**, **face**) es clasificada por Godot como de **orientación delantera** o **orientación trasera**:

- Será **delantera** si sus vértices se visualizan en pantalla en el sentido de las agujas del reloj.
- Será **trasera** si sus vértices se visualizan en pantalla en el sentido contrario de las agujas del reloj

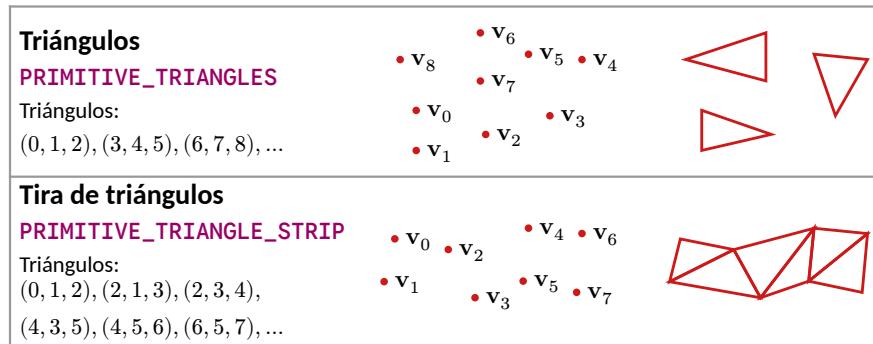
Godot usa la orientación para decidir si una cara se visualiza o no:

- Por defecto, Godot visualiza solo las delanteras (se llama hacer **cribado de caras**, **face culling**)
- Puede ser configurado para visualizar solo las delanteras, solo las traseras o todas. Se hace cambiando los parámetros de los *spatial shaders*.

Esta clasificación tiene utilidad especialmente en visualización 3D.

## Primitivas de tipos triángulos (rellenos)

En estos tipos de primitivas la secuencia codifica uno o más triángulos, todos ellos rellenos. Aquí vemos los puntos en las coordenadas y a la derecha un esquema de las aristas de los triángulos que se formarán.



## Secuencias indexadas

Para solucionar el problema, las APIs y engines permiten especificar una secuencia de vértices (con repeticiones) a partir de una secuencia de vértices únicos:

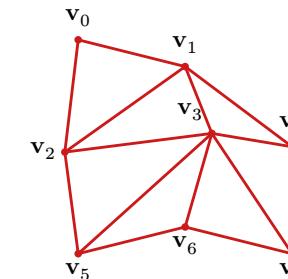
- Se parte de una secuencia  $V_n$  de  $n$  coordenadas arbitrarias de vértices  $V_n = \{v_0, v_1, \dots, v_{n-1}\}$ .
- Se usa una secuencia  $I_m$  de  $m$  **índices**  $I_m = \{i_0, i_1, \dots, i_{m-1}\}$  donde cada valor  $i_j$  es un entero entre 0 y  $n - 1$  (ambos incluidos). Puede haber valores repetidos.
- La secuencia de vértices  $V_n$  y la de índices determinan otra secuencia  $S_m$  de  $m$  vértices:

$$S_m = \{v_{i_0}, v_{i_1}, \dots, v_{i_{m-1}}\}$$

que tiene las mismas coordenadas de vértices de  $V_n$  pero en el orden especificado por los índices en  $I_m$ .

## Problema de vértices replicados

A veces necesitamos repetir coordenadas de un vértice, p.ej. si queremos visualizar estos 7 triángulos (en modo aristas):



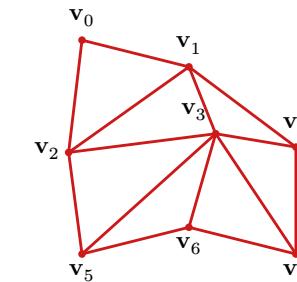
Usando **GL\_TRIANGLES**, necesitamos esta secuencia de vértices:

$v_0, v_2, v_1, v_1, v_2, v_3, v_1, v_3, v_4, v_3, v_5, v_3, v_5, v_6, v_3, v_6, v_7, v_3, v_7, v_4$

Supone **emplear más memoria y/o tiempo para visualizar del necesario**. La secuencia tiene 21 coordenadas de vértices, pero solo hay 8 distintos (p.ej.,  $v_2$  aparece 4 veces y  $v_3$  aparece 6 veces).

## Ejemplo de secuencia indexada

En este ejemplo que hemos visto antes



usaríamos una lista de índices (cada tres forman un triángulo):

$$I_21 = \{0, 2, 1, 1, 2, 3, 1, 3, 4, 2, 5, 3, 3, 5, 6, 3, 6, 7, 3, 7, 4\}$$

## Atributos de vértices

Las coordenadas de su posición se considera un **atributo** de los vértices, es un atributo imprescindible, pero en rasterización se pueden opcionalmente usar otros atributos, por ejemplo:

- El **color** del vértice (una terna RGB con valores entre 0 y 1).
- La **normal**: una vector unitario con tres coordenadas reales, determina la orientación de la superficie de un objeto en el punto donde está el vértice. Se usa para iluminación.
- Las **coordenadas de textura**: típicamente un par de valores reales, que se usan para determinar que punto de la textura se fija al vértice (lo veremos)

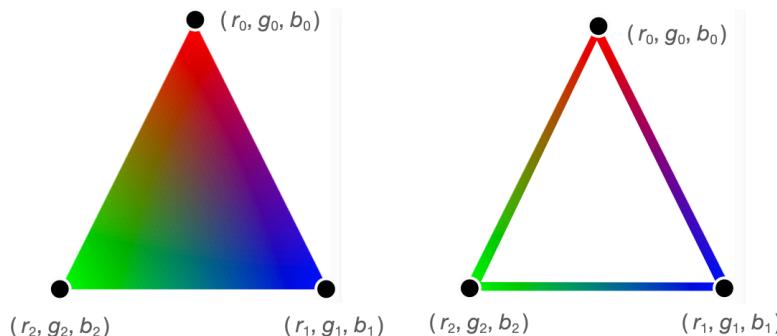
En Godot se pueden definir estos atributos y algunos más. En último término, el significado de un atributo (excepto el de la posición), está fijado en los *shaders* del cauce en uso. Godot permite atributos arbitrarios.

---

Subsección 2.2.  
**Atributos de vértices**

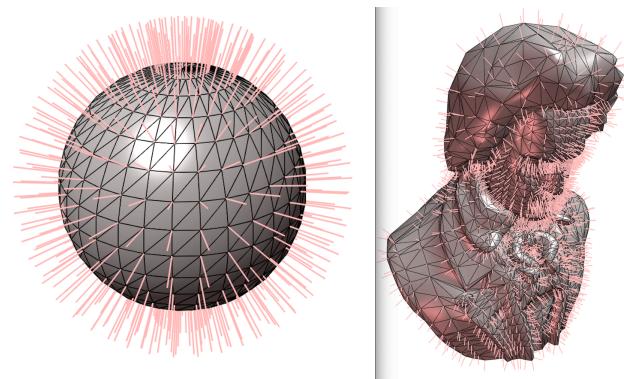
### Atributos: colores de vértices

Es posible asignar un color a cada vértice, es una terna RGB con tres reales ( $r, g, b$ ), (con valores entre 0 y 1) o bien una cuádrupla RGBA (RGB+transparencia). En el interior (o en las aristas) del polígono se usa interpolación para calcular el color de cada pixel.



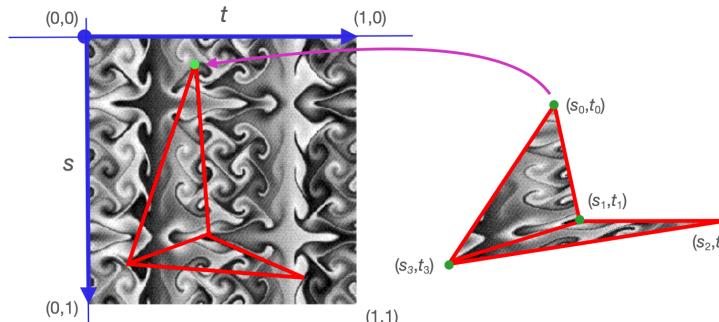
### Atributos: normales

En visualización 3D, a cada vértice se le puede asociar un vector de 3 componentes ( $x, y, z$ ) (su vector **normal**) que determina la orientación de la superficie en ese vértice y sirve para hacer el sombreado y la iluminación:



## Atributos: coordenadas de textura

Para usar imágenes (texturas) en lugar de colores, podemos asociar a cada vértice un par de reales  $(s, t)$  (sus **coordenadas de textura**), típicamente en  $[0, 1]^2$ . Esto determina como se aplica la imagen (a la izquierda) a las primitivas (a la derecha):



## Definición de valores de atributos

En Godot a cada vértice **siempre** se le asocia una tupla por cada atributo.

- Es decir, todo vértice tiene siempre asociado una posición, un color, una normal y unas coordenadas de textura (u otros atributos definidos por la aplicación).
- Según la configuración del cauce, algunos atributos serán usados o no. P.ej., si un objeto no tiene textura, no se usarán sus coordenadas de textura, o si no está activada la iluminación, no se usará la normal.
- Podemos definir único valor de un atributo para todos los vértices de una primitiva, o bien especificar un valor para cada vértice.

El valor de cada atributo está definido en cada pixel donde se proyecta la primitiva. Estos valores se calculan durante la rasterización usando **interpolación**.

## Envío de datos de la CPU a la GPU

Las GPUs modernas están diseñadas para visualizar secuencias de vértices y atributos **almacenados en la memoria de la GPU**:

- Esto se debe a que desde la GPU el acceso a su propia memoria es muchísimo más rápido que el acceso a la memoria del sistema.
- Sin embargo, el origen de los datos siempre estará en la memoria de la aplicación (la memoria del sistema, es decir, la accesible por la CPU). Esto se debe a que los datos pueden leerse de un archivo o generarse, pero siempre quedan inicialmente en la CPU.
- Por tanto, es necesario realizar un **envío de datos** desde la CPU a la GPU, previo a la visualización.

Hay diversas formas de realizar ese envío, en base a la decisión sobre cuando se hace.

## Modos de envío de datos a la GPU

Las dos formas de enviar las secuencias de vértices y sus atributos son:

- Envío en **modo inmediato**:

- ▶ Cada vez que queremos visualizar un frame, se envían los atributos e índices a la GPU por el bus del sistema.
- ▶ Este modo de visualización es muy ineficiente en tiempo (requiere transferir muchos datos por cada cuadro o frame), ya que el ancho de banda del bus del sistema es limitado (menor que los accesos a memoria en la GPU).

- Envío en **modo diferido**:

- ▶ Los datos de la secuencia de vértices se envían a la GPU una sola vez, usualmente como parte de la inicialización de la aplicación.
- ▶ Emplea mucho menos tiempo por cuadro que el anterior, ya que la transferencia de datos se hace menos veces, usualmente una sola vez.

## Modos de envío en Godot

El **engine Godot**:

- Usa normalmente el envío en **modo diferido**, para los nodos que contienen mallas (secuencia de vértices) en 2D y 3D.
- Se puede usar el envío en **modo inmediato**, usando nodos de tipos específicos para ello.
- Incluye una API de visualización 2D en modo inmediato (funciones para dibujar explícitamente líneas, rectángulos, polígonos, círculos, texto, etc...). Al usar estas funciones, todas las coordenadas y atributos se envían en cada llamada (la cual se hace en cada frame).

## Uso de los modos de envío

Por defecto, en gráficos se usa casi siempre el envío en **modo diferido** dada su mayor eficiencia. Sin embargo, a veces es conveniente usar el modo inmediato, únicamente cuando se dan cada una de estas dos condiciones:

- La secuencia de vértices y atributos es actualizada (cambia) por la aplicación (desde la CPU) con mucha frecuencia (p.ej. cada frame).
- La secuencia es pequeña (p.ej. menos de unos pocos miles de vértices).

En estos casos el envío previo a cada frame de los datos actualizados es realizable ya que no penaliza mucho el tiempo debido a que la secuencia de vértices no ocupa mucha memoria.

En mallas grandes se usa el envío en **modo diferido**:

- En cada frame pueden estar instanciadas en una posición, orientación o escala distinta (lo veremos más adelante).
- Si algunas coordenadas u otros atributos cambian, se pueden usar `vertex` o `geometry shaders` para programar esos cambios directamente en la GPU.

---

### Subsección 2.4.

#### Representación de mallas en Godot

---

## Almacenamiento de vértices y atributos: AOS y SOA (1/2)

Las APIs y engines suelen ofrecer dos formas de almacenar arrays de posiciones de vértices y sus atributos:

- **Array de estructuras (Array Of Structures, AOS):** se usa un array o vector, donde cada entrada contiene las coordenadas de un vértice y todos sus atributos.
- **Estructura de arrays (Structure Of Arrays, SOA):** se usa una estructura con varios (punteros a) arrays de número de elementos. Uno de ellos contiene las coordenadas y los otros contienen cada uno una tabla de atributos (colores, normales, coordenadas de textura).

Los **índices** (si hay) siempre están contiguos en su propio array.

En Godot se usa la opción SOA.

## Tuplas de reales para posiciones y otros atributos

En GDScript, se contemplan diversos tipos de datos para guardar tuplas con valores reales, a saber:

- **Vector2:** tupla de dos reales (p.ej., para posiciones 2D o coordenadas de textura)
- **Vector3:** tupla de tres reales (p.ej., para posiciones 3D o normales)
- **Color:** tupla de cuatro reales (p.ej., para colores RGB o RGBA)

Características:

- Los elementos de las tuplas son valores reales (números de coma flotante de precisión simple, 32 bits, según la norma IEEE 754).
- Estos tipos son clases, incluyen métodos para hacer operaciones con ellos (suma, producto por un real, etc...).

## Almacenamiento de vértices y atributos: AOS y SOA (2/2)

En AOS hay una única secuencia de valores reales:

$$\text{verts.} \equiv \left\{ \underbrace{\overbrace{x_0, y_0, z_0}_{\text{posición 0}}, \underbrace{r_0, g_0, b_0}_{\text{color 0}}, \underbrace{n_{x0}, n_{y0}, n_{z0}}_{\text{normal 0}}, \underbrace{s_0, t_0}_{\text{cc.t. 0}}}^{\text{vértice 0}}, \underbrace{x_1, y_1, z_1}_{\text{posición 1}}, \underbrace{r_1, g_1, b_1}_{\text{color. 1}}, \dots, \underbrace{s_{n-1}, t_{n-1}}_{\text{cc.t. } n-1} \right\}$$

En SOA hay una secuencia de reales por cada atributo (posibl. vacía):

$$\text{posiciones} \equiv \left\{ \underbrace{\overbrace{x_0, y_0, z_0}_{\text{posición 0}}, \underbrace{x_1, y_1, z_1}_{\text{posición 1}}, \underbrace{x_2, y_2, z_2}_{\text{posición 2}}, \dots, \underbrace{x_{n-1}, y_{n-1}, z_{n-1}}_{\text{posición } n-1}} \right\}$$

$$\text{colores} \equiv \left\{ \underbrace{\overbrace{r_0, g_0, b_0}_{\text{color 0}}, \underbrace{r_1, g_1, b_1}_{\text{color 1}}, \underbrace{r_2, g_2, b_2}_{\text{color 2}}, \dots, \underbrace{r_{n-1}, g_{n-1}, b_{n-1}}_{\text{color } n-1}} \right\}$$

$$\text{normales} \equiv \left\{ \underbrace{\overbrace{n_{x,0}, n_{y,0}, n_{z,0}}_{\text{normal 0}}, \underbrace{n_{x,1}, n_{y,1}, n_{z,1}}_{\text{normal 1}}, \underbrace{n_{x,2}, n_{y,2}, n_{z,2}}_{\text{normal 2}}, \dots, \underbrace{n_{x,n-1}, n_{y,n-1}, n_{z,n-1}}_{\text{normal } n-1}} \right\}$$

$$\text{cc.text.} \equiv \left\{ \underbrace{\overbrace{s_0, t_0}_{\text{cc.t. 0}}, \underbrace{s_1, t_1}_{\text{cc.t. 1}}, \underbrace{s_2, t_2}_{\text{cc.t. 2}}, \dots, \underbrace{s_{n-1}, t_{n-1}}_{\text{cc.t. } n-1}} \right\}$$

## Identificación de atributos y tablas asociadas

Se suele asociar un valor entero a cada atributo de vértice. En GDScript se definen el tipo enumerado **ArrayType** en la clase **Mesh**. Los posibles valores son:

Identificador	Valor	Significado
<b>Mesh.ARRAY_VERTEX</b>	0	Coordenadas de posición (obligatorias)
<b>Mesh.ARRAY_NORMAL</b>	1	Vector normal
<b>Mesh.ARRAY_TANGENT</b>	2	Vector tangente
<b>Mesh.ARRAY_COLOR</b>	3	Color RGB
<b>Mesh.ARRAY_TEX_UV</b>	4	Coordenadas de textura
<b>Mesh.ARRAY_TEX_UV2</b>	5	Segundas coordenadas de textura
<b>Mesh.ARRAY_CUSTOM0-3</b>	6 – 9	Atributos definidos por el usuario

Existen otros posibles atributos relacionados con *skinning* y *rigging*, pero no los veremos aquí. Hay un total de **Mesh.ARRAY\_MAX** (13 en la actualidad) posibles atributos.

## Almacenamiento en arrays empaquetados de GDScript

En GDScript las tablas de atributos de vértices se pueden codificar en arrays **empaquetados (packed)**, son un tipo (una clase) con un arrays de elementos que garantizan que todos los valores están adyacentes en memoria.

Atributo	Clase array	Tipo elem.
Posiciones 2D	PackedVector2Array	Vector2
Posiciones 3D	PackedVector3Array	Vector3
Colores	PackedColorArray	Color
Normales	PackedVector3Array	Vector3
Coords. textura	PackedVector2Array	Vector2

Los índices (si hay) se almacenan en un array empaquetado de enteros, de tipo **PackedInt32Array** con enteros valiendo hasta  $2^{31}$ , lo cual es suficiente para prácticamente cualquier malla (aunque se podría usar **PackedInt64Array** si hiciera falta).

## Clases para mallas en Godot

Para definir objetos gráficos mediante mallas (arrays de vértices) en Godot, se usan diversas clases:

- **ArrayMesh**: derivada de **Mesh**, contiene una malla que se visualizará en **modo diferido**.
- **ImmediateMesh**: derivada de **Mesh**, contiene una malla que se visualizará en **modo inmediato**.
- **SurfaceTool**: derivada de **RefCounted**, es una clase que facilita la creación de mallas (de tipo **ArrayMesh** o **ImmediateMesh**) a partir de llamadas a métodos que van añadiendo vértices y atributos.
- **MeshInstance2D**, derivada de **Node2D**, permite *instanciar* una malla (**Mesh**) en una posición, orientación y escalas dadas en el plano 2D.
- **MeshInstance3D**, derivada de **Node3D**, permite *instanciar* una malla (**Mesh**) en una posición, orientación y escalas dadas en el espacio 3D.

## Creación de mallas en GDScript

En las siguientes transparencias vemos diversos ejemplos de creación de mallas con GDScript:

- En primer ejemplos en 2D y después en 3D.
- Usando mallas no indexadas y mallas indexadas.
- Los envíos son en modo diferido o inmediato.
- En algunos ejemplos se usa una tabla de colores por vértice (se interpola el color).
- En 3D se supone que podemos calcular las normales (no se muestra cómo) y asignar material a las superficies.

## Malla no indexada 2D, color plano (1/3)

Ejemplo de inicialización de un `MeshInstance2D` para visualizar dos triángulos no adyacentes (6 vértices) con un color plano (sin atributos de color por vértice).

```
extends MeshInstance2D

func _ready():

## declarar y dimensionar un array para las tablas
var tablas : Array = []
tablas.resize( Mesh.ARRAY_MAX )

## añadir la tabla con las coordenadas de posición de 6 vértices
## .... (siguiente transparencia) ....

## cambiar el color de la malla (atributo 'modulate' del nodo)
modulate = Color( 1.0, 0.5, 0.5 )

## inicializar el atributo 'mesh' de este nodo
mesh = ArrayMesh.new() ## crea malla en modo diferido, vacía
mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )
```

Sesión 2: El engine Godot. Mallas.

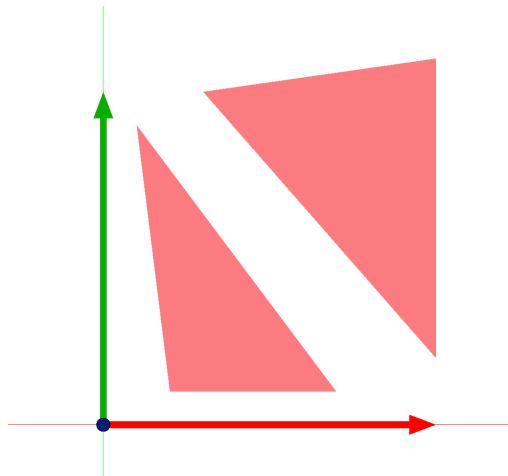
Created 2025-12-01

Page 57 / 100.

2. Mallas en Godot..  
2.5. Mallas en 2D..

## Malla no indexada 2D, color plano (3/3)

Aquí vemos los dos triángulos en el plano 2D (se han añadido los ejes de coordenadas para tener una referencia de la escala y posición):



## Malla no indexada 2D, color plano (2/3)

La creación de la tabla de posiciones (3 vértices) se puede hacer directamente creando un `PackedVector2Array` con las coordenadas de los vértices y situandolo en la entrada correspondiente del array `tablas`:

```
## añadir la tabla con las coordenadas de posición de 6 vértices:
tablas[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([
    Vector2( 0.2, 0.1 ), Vector2( 0.7, 0.1 ), Vector2( 0.1, 0.9 ),
    Vector2( 0.3, 1.0 ), Vector2( 1.0, 0.2 ), Vector2( 1.0, 1.1 ),
])
```

Otra posibilidad es usar `push_back` (permitirá algoritmos complejos):

```
## añadir la tabla con las coordenadas de posición de 6 vértices:
var posv := PackedVector2Array() # crear array posic. verts. vacío

posv.push_back(Vector2(0.2,0.1)); posv.push_back(Vector2(0.7,0.1))
posv.push_back(Vector2(0.1,0.9)); posv.push_back(Vector2(0.3,1.0))
posv.push_back(Vector2(1.0,0.2)); posv.push_back(Vector2(1.0,1.1))

tablas[ Mesh.ARRAY_VERTEX ] = posv # asignar la tabla de posiciones
```

Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 58 / 100.

2. Mallas en Godot..  
2.5. Mallas en 2D..

## Mallas con colores de vértices en 2D (1/3)

En este ejemplo se usa una tabla de colores de vértices, lo cual hace que esos colores se interpolen en el interior de los triángulos. No es necesario actualizar `modulate`

```
extends MeshInstance2D

func _ready():

## declarar y dimensionar un array para las tablas
var tablas : Array = []
tablas.resize( Mesh.ARRAY_MAX )

## añadir las tablas con posiciones y colores de 6 vértices
## .... (siguiente transparencia) ....

## inicializar el atributo 'mesh' de este nodo
mesh = ArrayMesh.new() ## crea malla en modo diferido, vacía
mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )
```

## Mallas con colores de vértices en 2D (2/3)

La tabla de colores se debe proporcionar como un `PackedColorArray`, con un color por cada vértice. Por ejemplo:

```
## añadir las tablas con posición y colores de 6 vértices
tablas[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([ ... ]) # igual
tablas[ Mesh.ARRAY_COLOR ] = PackedColorArray([
    Color( 1.0,0.0,0.0 ), Color( 0.0,1.0,0.0 ), Color( 0.0,0.0,1.0 ),
    Color( 1.0,1.0,0.0 ), Color( 0.0,1.0,1.0 ), Color( 1.0,0.0,1.0 ),
])
```

También es posible inicializar la tabla de colores con `push_back`:

```
var posv := PackedVector2Array() # crear array posic. verts. vacío
var colv := PackedColorArray() # crear array colores vértices vacío

posv.push_back(Vector2(0.2,0.1)); colv.push_back(Color(1.0,0.0,0.0))
# .... posiciones y colores del resto de vértices ...

tablas[ Mesh.ARRAY_COLOR ] = colv # asignar la tabla de colores
tablas[ Mesh.ARRAY_VERTEX ] = posv # asignar la tabla de posiciones
```

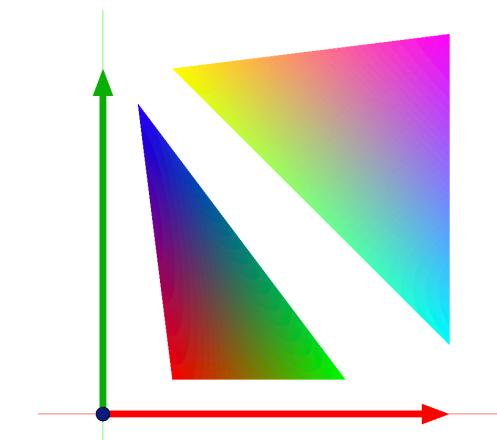
## Mallas indexadas en 2D (1/2)

Malla de 4 vértices, que forman dos triángulos adyacentes ( $3 \times 2$  índices).

```
extends MeshInstance2D
func _ready():
    var tablas : Array = [] ## array con tablas de atributos
    tablas.resize( Mesh.ARRAY_MAX ) ## redimensionar el array
    ## añadir las tablas: posiciones y colores
    tablas[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([
        Vector2( 0.2, 0.1 ), Vector2( 0.7, 0.2 ),
        Vector2( 0.3, 1.1 ), Vector2( 1.2, 1.1 ),
    ])
    tablas[ Mesh.ARRAY_COLOR ] = PackedColorArray([
        Color( 1.0, 0.0, 0.0 ), Color( 0.0, 1.0, 0.0 ),
        Color( 0.0, 0.0, 1.0 ), Color( 1.0, 1.0, 0.0 )
    ])
    ## definir la tabla de índices (6 índices)
    tablas[ Mesh.ARRAY_INDEX ] = PackedInt32Array([ 0,1,2, 2,1,3 ])
    ## crear un 'Mesh' en modo diferido y añadirle las tablas
    mesh = ArrayMesh.new()
    mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )
```

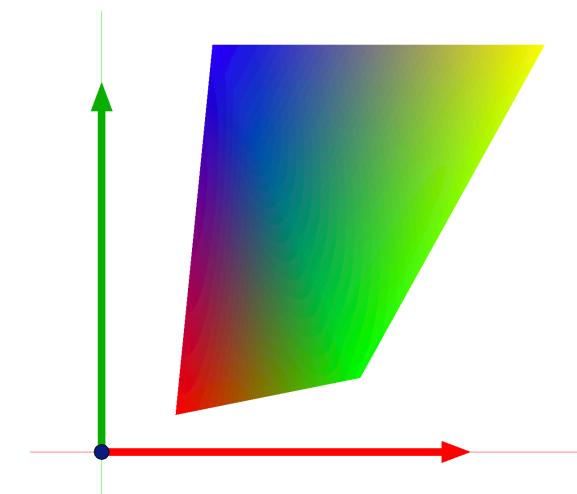
## Malla no indexada 2D, color plano (3/3)

Aquí vemos los dos triángulos con las gradaciones de colores. En el interior de los triángulos se realiza una interpolación lineal cuyos valores extremos son los colores de los vértices:



## Mallas indexadas en 2D (2/2)

Aquí vemos los dos triángulos adyacentes, con interpolación de colores entre los 4 colores:



## Mallas 2D indexada con texturas (1/2)

Creamos las coordenadas de textura y asignamos el atributo `texture`

```
extends MeshInstance2D

func _ready():
    ## definir tablas para 4 vértices formando 2 triángulos:
    var tablas : Array = [] ; tablas.resize( Mesh.ARRAY_MAX )
    tablas[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([ ... ]) # igual
    tablas[ Mesh.ARRAY_TEX_UV ] = PackedVector2Array([
        Vector2( 0.0,0.0 ), Vector2( 1.0,0.0 ),
        Vector2( 0.0,1.0 ), Vector2( 1.0,1.0 )
    ])
    tablas[ Mesh.ARRAY_INDEX ] = PackedInt32Array([ 0,1,2, 2,1,3 ])

    ## crear el objeto 'mesh'
    mesh = ArrayMesh.new()
    mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )

    ## cargar la textura y asignarla a este objeto
    texture = CargarTextura( "imgs/madera2.jpg" )
```

Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 65 / 100.

## Carga de texturas en GDScript

La función `CargarTextura` carga una imagen almacenada en un archivo y crea un objeto `ImageTexture` a partir de ella:

```
func CargarTextura( arch : String ) -> ImageTexture :

    ## crear un objeto 'Image' con la imgen
    var imagen := Image.new()
    assert( imagen.load(arch) == OK, "Error cargando '"+arch+"'." )

    ## crear un objeto 'ImageTexture' a partir del objeto 'Image'
    var textura := ImageTexture.create_from_image( imagen )
    print("Textura cargada desde archivo: '",arch,"'.")

    ## devolver la textura
    return textura
```

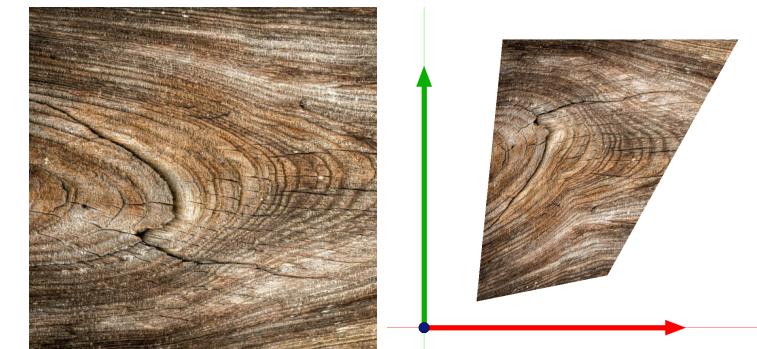
Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 67 / 100.

## Mallas 2D indexada con texturas (2/2)

Aquí vemos la textura y resultado de aplicarla a los dos triángulos:



Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 66 / 100.

## Envío en modo inmediato en 2D (1/2)

En cada frame se define de nuevo la malla no indexada con 1 triángulo. Ahora usamos el método `_process`, que se ejecuta en cada frame. En este ejemplo, se calcula un vector "d" que varía con el tiempo y luego se redefine la malla.

```
extends MeshInstance2D

var t : float = 0 # tiempo total desde inicio en segundos.

func _ready():
    mesh = ImmediateMesh.new() # crear 'Mesh' vacío al inicio.

func _process( delta: float ):

    # actualiza tiempo transcurrido
    t = t+delta

    # calcular un vector 'd' que va cambiando con el tiempo
    var d := 0.2*Vector2( cos(4.0*t), sin(4.0*t) )

    ## volver a definir las tablas
    ## ...(ver siguiente transparencia) ....
```

Sesión 2: El engine Godot. Mallas.

Created 2025-12-01

Page 68 / 100.

## Envío en modo inmediato en 2D (2/2)

Para redefinir en cada frame la malla, se usan los métodos de `ImmediateMesh` que permiten ir añadiendo cada vértice y sus atributos a la malla. Para cada vértice se especifica su posición después de haber fijado sus otros atributos:

```
## volver a definir las tablas
mesh.clear_surfaces(). ## limpia el mesh
mesh.surface_begin( Mesh.PRIMITIVE_TRIANGLES ) # dar tipo de primitiva

## definir vértice 0
mesh.surface_set_color( Color(1.0,0.0,0.0) ). 
mesh.surface_add_vertex_2d( Vector2(0.2,0.2) + d ) ## usa 'd' animado
## definir vértice 1
mesh.surface_set_color( Color(0.0,1.0,0.0) ) ## opcional: si no conserva
mesh.surface_add_vertex_2d( Vector2(1.0,0.5) )
## definir vértice 2
mesh.surface_set_color( Color(0.0,0.0,1.0) ) ## opcional: si no conserva
mesh.surface_add_vertex_2d( Vector2(0.5,1.0) )

mesh.surface_end() # fin de los vértices
```

## Malla indexada 3D

En 3D es necesario definir un *material* para la malla. Un *material* es un objeto que determina como los triángulos reflejan la luz proveniente de las fuentes de luz.

```
extends MeshInstance3D

func _ready():
    ## declarar y dimensionar un array para las tablas (igual que en 2D)
    var tablas : Array = []
    tablas.resize( Mesh.ARRAY_MAX )

    ## añadir las tablas posiciones, colores e índices de un triángulo
    ## .... (ver siguiente transparencia) ....

    ## inicializar el atributo 'mesh' de este nodo (igual que en 2D)
    mesh = ArrayMesh.new() ## crea malla en modo diferido, vacía
    mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )

    ## asignarle un material sin iluminación ni sombreado
    ## ... (ver siguientes transparencias) ....
```

---

### Subsección 2.6. Mallas en 3D.

---

## Malla indexada 3D: tablas

La creación de la tabla de posiciones (3 vértices) se puede hacer directamente creando un `PackedVector3Array`. Los colores y los índices (si están) se crean igual que en 2D.

Aquí vemos la creación de una malla con un único triángulo:

```
## añadir las tablas posiciones, colores e índices de un triángulo
tablas[ Mesh.ARRAY_VERTEX ] = PackedVector3Array([
    Vector3(0.6,0.6,0.1), Vector3(0.1,0.8,0.1), Vector3(0.2,0.1,0.1)
])
tablas[ Mesh.ARRAY_COLOR ] = PackedColorArray([
    Color( 1, 0, 0 ), Color( 0, 1, 0 ), Color( 0, 0, 1 )
])
tablas[ Mesh.ARRAY_INDEX ] = PackedInt32Array([
    0, 1, 2
])
```

Al igual que en 2D, cualquiera de estas tablas **se puede crear usando `push_back`** en lugar de inicializarla directamente.

## Malla indexada 3D: material con colores de vértices (1/2)

En este ejemplo se añade un material sin iluminación ni sombreado, quiere decir que los colores de los triángulos no se ven afectados por las fuentes de luz presentes en la escena.

```
# asignarle un material sin iluminación ni sombreado
var mat := StandardMaterial3D.new() # crear objeto 'StandardMaterial3D'

## configurar el objeto con el material ('mat')
mat.vertex_color_use_as_albedo = true # usar colores de vértices
mat.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADED # sin sombr.
mat.cull_mode = BaseMaterial3D.CULL_DISABLED # sin cribado

## redefinir atributo 'material_override' del nodo 'MeshInstance3D'
material_override = mat
```

Se desactiva el cribado de caras para que se vean todos los triángulos independientemente de en qué orden se especifiquen sus tres vértices.

## Malla indexada 3D: material con color plano (1/2)

Si no hay colores de vértices y queremos un color plano, el material se debe configurar definiendo el atributo `albedo_color` del material (y poniendo a `false` el atributo `vertex_color_use_as_albedo`):

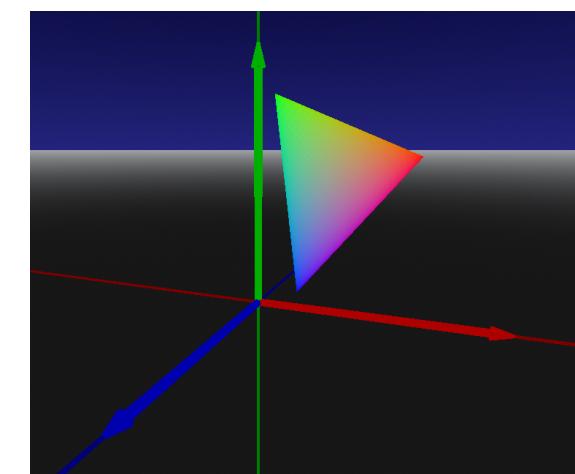
```
# asignarle un material sin iluminación ni sombreado
var mat := StandardMaterial3D.new() # crear objeto 'StandardMaterial3D'

## configurar el objeto con el material ('mat')
mat.vertex_color_use_as_albedo = false # no usar colores de vértices
mat.albedo_color = Color( 1.0, 0.4, 0.4 ) # color plano de la malla
mat.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADED # sin sombr.
mat.cull_mode = BaseMaterial3D.CULL_DISABLED # sin cribado
mat.lighting = false # no usar la luces

## redefinir atributo 'material_override' del nodo 'MeshInstance3D'
material_override = mat
```

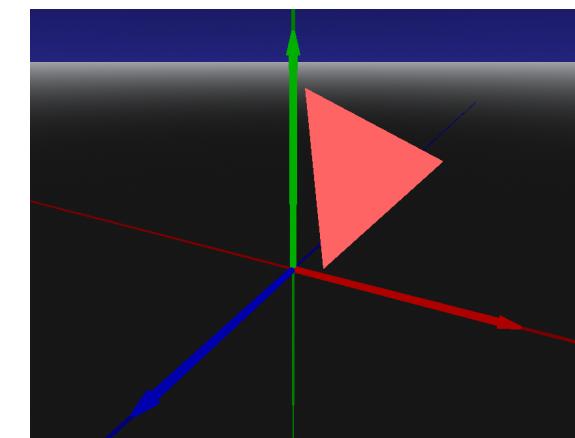
## Mallas indexada 3D: : material con colores de vértices (2/2)

Aquí vemos el triángulo junto con unos ejes en 3D:



## Mallas indexada 3D: : material con color plano (2/2)

Aquí vemos el triángulo con el color plano:



## Sombreado con iluminación

En los ejemplos 3D anteriores, el material no tiene texturas ni responde a la iluminación

- En las muchas aplicaciones 3D se usan materiales que responden a la iluminación y además muchas veces tienen asociadas texturas.
- En Godot se pueden definir materiales complejos, con texturas y que responden a la iluminación, usando la clase `StandardMaterial3D` (o creando materiales personalizados con *shaders*).
- También se pueden usar texturas.
- Las texturas requieren definir una *tabla de coordenadas de textura*, que es un `Vector2` por cada vértice.
- La iluminación requiere que cada vértice tenga asociada un vector normal que determina la orientación de la superficie en ese vértice.

## Especificación de un material con iluminación

La creación de un material con iluminación se puede hacer como se indica aquí:

```
var mat := StandardMaterial3D.new()

mat.cull_mode = BaseMaterial3D.CULL_DISABLED # sin cribado
mat.albedo_color = Color( 1.0, 0.5, 1.0 ) # color base de la malla
mat.metallic = 0.25
mat.roughness = 0.15
material_override = mat
```

En caso de que se quiera usar una textura para el color base (en lugar de un color plano), se puede asignar el atributo `albedo_texture` en lugar de `albedo_color`:

```
mat.albedo_texture = CargarTextura( "imgs/grid1.png" )
```

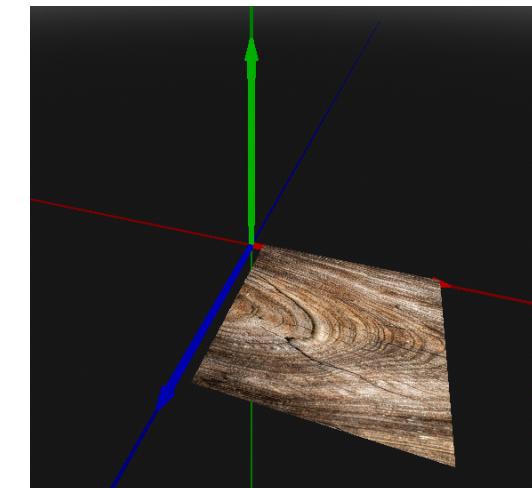
## Especificación de normales y coordenadas de textura

En este ejemplo 3D se define una malla rectangular con dos triángulos, todos ellos con la misma normal (dirección Y+). Las coordenadas de textura se crean de forma que la imagen de textura se vea completa en el rectángulo:

```
tablas[ Mesh.ARRAY_VERTEX ] = PackedVector3Array([
    Vector3( 0.1, 0.1, 0.1 ), Vector3( 0.9, 0.1, 0.1 ),
    Vector3( 0.9, 0.1, 0.9 ), Vector3( 0.1, 0.1, 0.9 )
])
tablas[ Mesh.ARRAY_NORMAL ] = PackedVector3Array([
    Vector3( 0.0, 1.0, 0.0 ), Vector3( 0.0, 1.0, 0.0 ),
    Vector3( 0.0, 1.0, 0.0 ), Vector3( 0.0, 1.0, 0.0 ),
])
tablas[ Mesh.ARRAY_TEX_UV ] = PackedVector2Array([
    Vector2( 0.0, 0.0 ), Vector2( 1.0, 0.0 ),
    Vector2( 1.0, 1.0 ), Vector2( 0.0, 1.0 )
])
tablas[ Mesh.ARRAY_INDEX ] = PackedInt32Array([
    0, 1, 2, 0, 2, 3
])
```

## Material con iluminación y textura

Aquí vemos la malla de dos triángulos con iluminación y textura:



## Uso de *SurfaceTool* para generar mallas en 3D

Los objetos de tipo **ArrayMesh** pueden, adicionalmente, crearse usando un objeto de tipo **SurfaceTool**.

- Un objeto **SurfaceTool** guarda las posiciones y otros atributos de los vértices de una malla.
- Se puede inicializar especificando el tipo de primitiva, y luego la posición y atributos de cada vértice, uno a uno.
- Permite crear arrays indexados o no indexados
- La ventaja frente a los otros métodos vistos reside en que **SurfaceTool** incluye métodos para:
  - ▶ Calcular automáticamente las normales (y las tangentes) de los vértices
  - ▶ Obtener la caja englobante de la malla.

## Ejemplo de uso de *SurfaceTool* para malla no indexada

En este ejemplo se crea una malla no indexada con dos triángulos, y a todos los vértices se les asigna el mismo color, aunque se podría cambiar el color antes de cada **add\_vertex** (solo muestro la creación de **mesh**):

```
var st = SurfaceTool.new()

st.begin( Mesh.PRIMITIVE_TRIANGLES ) ## iniciar vertices
st.set_color( Color( 1.0, 0.5, 0.5 ) ) ## color de siguientes vértices

st.add_vertex( Vector3( 0.1, 0.0, 0.1 ) ) ## añade vert. 0
st.add_vertex( Vector3( 0.9, 0.0, 0.1 ) ) ## añade vert. 1
st.add_vertex( Vector3( 0.1, 0.0, 0.9 ) ) ## añade vert. 2

st.add_vertex( Vector3( 1.0, 0.0, 0.2 ) ) ## añade vert. 3
st.add_vertex( Vector3( 0.2, 0.0, 1.0 ) ) ## añade vert. 4
st.add_vertex( Vector3( 1.1, 0.0, 1.1 ) ) ## añade vert. 5

mesh = st.commit() ## crea objeto 'ArrayMesh' y lo asigna
```

## Ejemplo de uso de *SurfaceTool* para malla indexada

Ahora se añade una tabla de índices y se crea un malla indexada (2 triángulos adyacentes, 4 vértices):

```
var st = SurfaceTool.new()

st.begin( Mesh.PRIMITIVE_TRIANGLES ) ## iniciar vertices
st.set_color( Color( 0.0, 1.0, 1.0 ) ) ## color de siguientes vértices

## añadir los 4 vértices
st.add_vertex( Vector3( 0.1, 0.0, 0.1 ) ) ## añade vert. 0
st.add_vertex( Vector3( 0.9, 0.0, 0.1 ) ) ## añade vert. 1
st.add_vertex( Vector3( 0.9, 0.0, 0.9 ) ) ## añade vert. 2
st.add_vertex( Vector3( 0.1, 0.0, 0.9 ) ) ## añade vert. 3

## añadir los 6 índices (2 triángulos)
st.add_index(0) ; st.add_index(1) ; st.add_index(2) # 1er tri.
st.add_index(0) ; st.add_index(2) ; st.add_index(3) # 2º tri.

mesh = st.commit() ## crea objeto 'ArrayMesh' y lo asigna
```

### Subsección 2.7.

#### Mallas indexadas de triángulos en 3D

## Mallas Indexadas.

En gráficos (y sobre todo en visualización 3D), es común que una secuencia de vértices codifique una malla de triángulos que comparten vértices. A esas secuencias las llamamos **Mallas indexadas** (de triángulos). En la aplicación se suelen representar usando 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. Se puede considerar como una secuencia de valores enteros consecutivos en memoria.
- **Tablas de atributos:** por cada atributo puede haber una tabla de valores flotantes (colores, normales, coordenadas de textura, etc...). Tiene un número de entradas igual al número de vértices. Cada entrada puede ser una tupla de 2, 3 o 4 flotantes.

## Estructura de datos

La tabla de índices en realidad se llama *tabla de triángulos* y (para  $n$  triángulos) tiene  $3n$  índices de vértices (enteros sin signo), y la de vértices  $3m$  valores reales:

Tabla Triángulos ( $n$ tri.)			Tabla Vértices ( $m$ verts.)			
$i_{0,0}$	$i_{0,1}$	$i_{0,2}$	0	$x_0$	$y_0$	$z_0$
$i_{1,0}$	$i_{1,1}$	$i_{1,2}$	1	$x_1$	$y_1$	$z_1$
$i_{2,0}$	$i_{2,1}$	$i_{2,2}$	2	$x_2$	$y_2$	$z_2$
$i_{3,0}$	$i_{3,1}$	$i_{3,2}$	3	$x_3$	$y_3$	$z_3$
:	:	:	4	$x_4$	$y_4$	$z_4$
$i_{n-2,0}$	$i_{n-2,1}$	$i_{n-2,2}$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$i_{n-1,0}$	$i_{n-1,1}$	$i_{n-1,2}$	$m-2$	$x_{m-2}$	$y_{m-2}$	$z_{m-2}$
			$m-1$	$x_{m-1}$	$y_{m-1}$	$z_{m-1}$

$0 \leq i_{jk} < m$

## Generación y manipulación procedural de mallas indexadas en Godot

En Godot, las mallas indexadas se pueden crear y manipular en GDScript de diversas formas:

- Creando en cada frame la malla como **ImmediateMesh**, vértice a vértice.
- Usando al inicio **SurfaceTool** para crear un **ArrayMesh** vértice a vértice.
- Creando inicio arrays Godot (empaquetados o no) con atributos de vértices e índices, y luego creando con esas tablas objetos **ArrayMesh**.
- Usando la clase **MeshDataTool** para manipular arrays de vértices e índices a partir de un **ArrayMesh** ya existente. La clase **MeshDataTool** incorpora diversos algoritmos de manipulación de mallas y estructuras de datos auxiliares (por ejemplo, la tabla de aristas).

## Ejemplo de algoritmos

Diversos ejemplos de algoritmos para mallas indexadas en 3D:

- Generación de mallas como superficies parámetricas (*B-splines*, *NURBS*, etc...)
- Generación de mallas de revolución (ver práctica 2)
- Cálculo de normales de superficies suaves (ver práctica 2).
- Cálculo de tangentes.
- Calculo de aristas para visualización, o bien como entrada de otros algoritmos.
- Subdivisión de caras para mejorar la calidad.
- Asignación procedural de coordenadas de texura.

## Sección 3. Problemas

1. Problemas de creación de mallas 2D
2. Problemas de creación de mallas 3D

3. Problemas.
- 3.1. Problemas de creación de mallas 2D.

### Introducción

Vemos problemas de mallas 2D en Godot. Para hacer los problemas, debes de:

1. Crear un proyecto nuevo en Godot.
2. En la escena principal, añadirle un nodo raíz de tipo **Node2D**
3. Descargar el archivo **raiz\_problemas\_2D.gd** de los materiales de teoría y situarlo en la carpeta del proyecto.
4. Adjuntar el script **raiz\_problemas\_2D.gd** al nodo raíz de la escena. Se encarga de dibujar los ejes, poner el fondo a blanco, y permitir interacción con el ratón.
5. Para cada problema 2D, crear un nodo hijo del nodo raíz de tipo **Node2D** y adjuntarle el script que resuelve el problema.
6. Ejecutar. Mediante el uso del ratón, se puede mover y hacer zoom en la vista 2D. Inicialmente esa vista está centrada en el origen y muestra un cuadrado de lado 2 (de -1 a 1 en ambos ejes). Para ver únicamente el nodo de un problema, en el árbol de escena haz todos los nodos no visibles excepto ese nodo.

## Subsección 3.1. Problemas de creación de mallas 2D

3. Problemas.
- 3.1. Problemas de creación de mallas 2D.

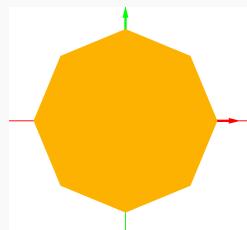
### Polígono regular relleno de color plano

#### Problema 2.1:

Implementa un nodo de tipo **MeshInstance** con una malla (**no indexada**) para un polígono regular de **n** lados relleno de color naranja plano (RGB (1.0, 0.7, 0.0)), con radio **r** y centro en el origen (ver figura).

El polígono estará formado por **n** triángulos, cada uno con un vértice en el centro y los otros dos en el contorno. Los valores de **n** y **r** se declaran como dos constantes de GDScript (**const**), como se indica aquí:

```
const n : int = 8  
const r : float = 0.8
```



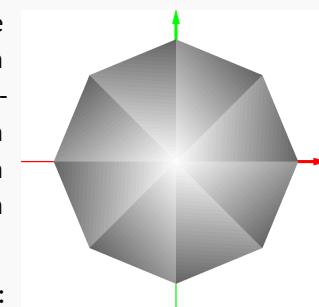
Los valores de estas constantes se podrán cambiar sin tocar nada del resto del script.

## Polígono regular relleno con gradaciones

### Problema 2.2:

Crea otro **Node2D**, y asígnale un script para visualizar el mismo polígono regular que antes (también con una malla **no indexada**), solo que ahora debes asignar colores a los vértices para que los triángulos aparezcan con una graduación en tonos de gris como en la figura. Cada triángulo que forma el polígono regular será blanco en el vértice del centro, gris claro en otro y gris oscuro en el tercero.

Responde razonadamente a esta cuestión:  
 ¿ cuantos vértices debe tener la tabla de vértices ?



## Polígono regular hecho de líneas

### Problema 2.3:

Repite los dos problemas anteriores, con los mismos requerimientos, pero ahora usando **mallas indexadas**, de forma que el número de vértices e índices sea mínimo.

Responde razonadamente a estas cuestiones:

- ¿ cuantos vértices debe tener ahora la tabla de vértices en cada caso ?
- ¿ y cuantos índices debe haber ?

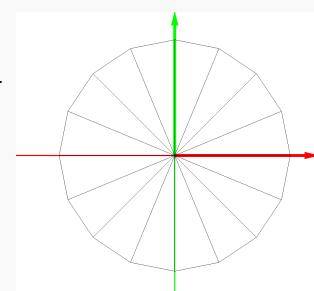
## Aristas del polígono regular

### Problema 2.4:

Crea un nuevo nodo **MeshInstance2D** de forma que ahora veamos simplemente las aristas del polígono regular descrito en los anteriores problemas. En la figura se ve para **n** a 16 y el mismo radio.

Considera dos casos:

- Usando una malla no indexada.
- Usando una malla indexadas.



## Generación de malla con segmentos de normales

### Problema 2.5:

Usando el código de la práctica 2, crea un script global (*autoload*) con una función que genere un objeto de tipo `MeshInstance3D` con una malla no indexada con los segmentos en las normales de una malla dada. La función tendrá la siguiente declaración:

```
func genSegNormales( verts, norms : PackedVector3Array,
    lon : float, color : Color ) -> MeshInstance3D :
```

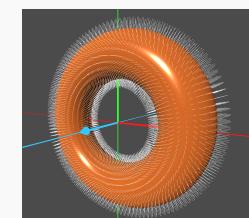
donde `verts` es la tabla de vértices de la malla, `norms` la tabla de normales, `lon` la longitud de los segmentos y `color` el color de los segmentos. Usa el tipo de primitiva *lineas*, y asegúrate de que a los segmentos no les afecta la iluminación.

(continua...)

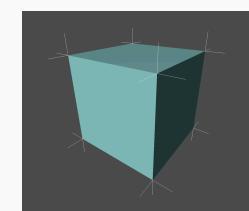
## Generación de malla con segmentos de normales

### Problema 2.5 (continuación):

Una vez tengas la función disponible, úsala en la función `_ready` de alguna malla (por ejemplo, el *Donut* o los cubos de la práctica 2), para añadir al objeto un nodo hijo con la malla de segmentos creada por la función.



Puedes capturar el evento de pulsación de la tecla N del objeto para activar y desactivar la visualización de las normales en ese objeto. Para ello, usa un valor lógico y el atributo de visibilidad de la malla de segmentos.



## Generación de normales con ponderación por área

### Problema 2.6:

Las normales de una malla indexada de triángulos se pueden calcular promediando en cada vértice las normales de los triángulos adyacentes al vértice (el código está en el guión de la práctica 2).

Sin embargo, una variante de este algoritmo consiste en modificar ese promedio, de forma que el peso de cada normal en el promedio sea proporcional al área del triángulo correspondiente a dicha normal, lo cual se supone que puede aproximar mejor la orientación real de la superficie en el vértice.

Usando el código de la práctica 2, escribe y prueba una variante del algoritmo de cálculo de normales que implemente este método. Puedes partir de una copia del código del guion de prácticas, haciendo las mínimas modificaciones o añadidos necesarios.

## Fin de transparencias.