

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

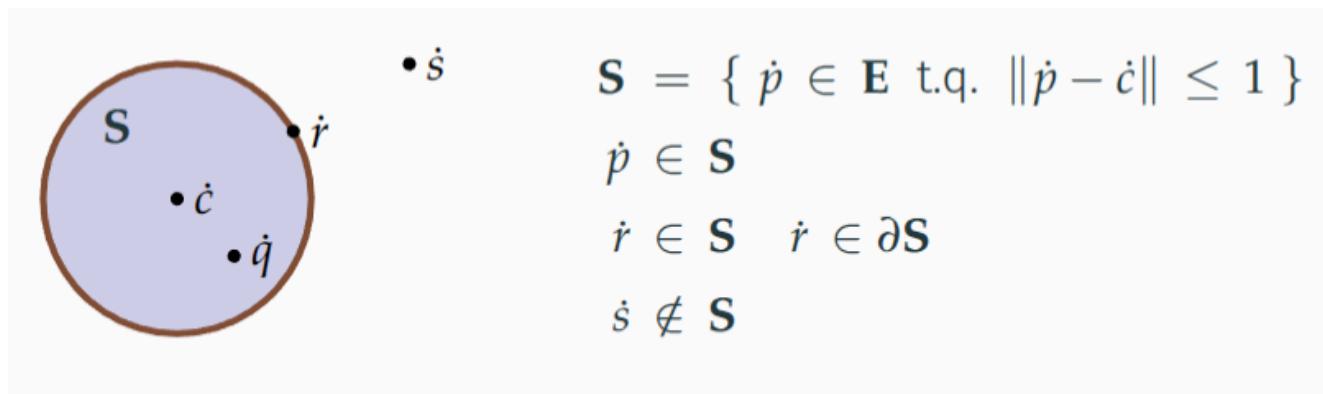
Un **modelo geométrico** es un modelo matemático abstracto que sirve para representar un objeto geométrico que existe en un espacio afín E (2D o 3D).

- 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.

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)

Representaciones computacionales

El modelo basado en subconjuntos de puntos del espacio:

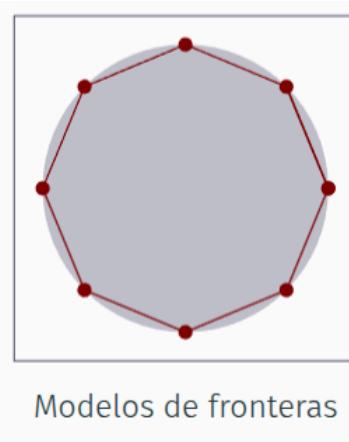
- Es un modelo válido para cualquier geometría (es el modelo **más general** posible).
- En muchos casos, **no se puede representar en la memoria** (finita, discreta) de un ordenador.

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..

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).



- **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:

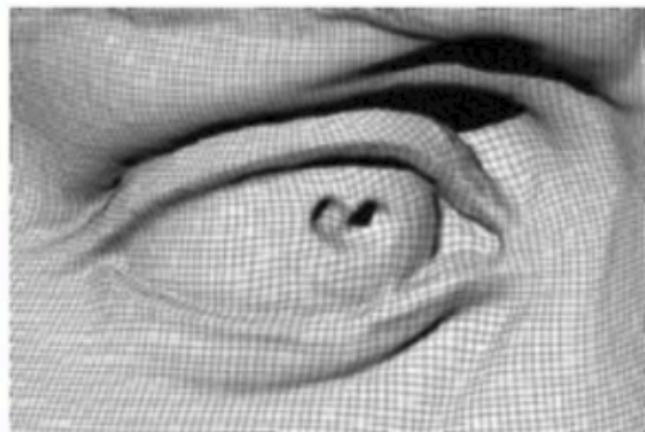


1. Modelos geométricos..

1.1. Modelos geométricos. Introducción..

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

Otros modelos de fronteras: nubes de puntos

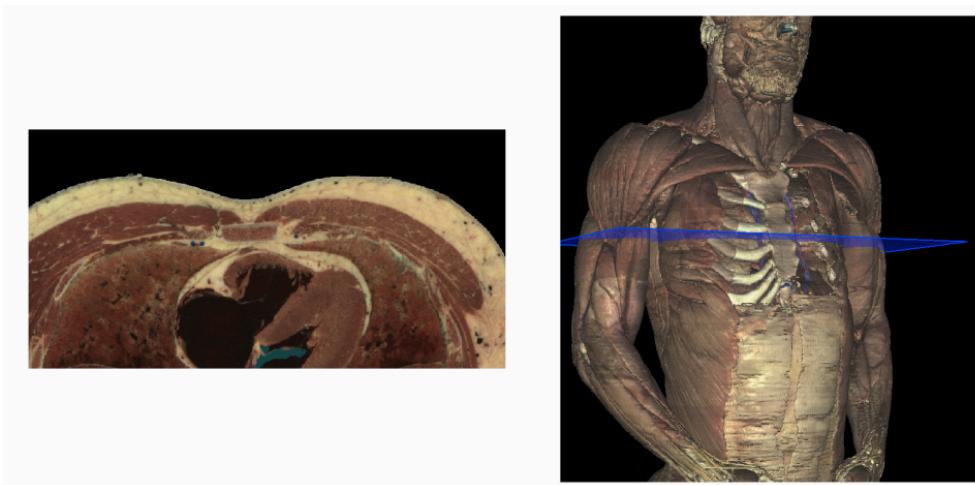
Una **nube de puntos** es un conjunto finito de puntos en el espacio 3D, cada uno con una posición y un color, que aproximan la superficie del objeto. No hay conectividad entre los puntos. Se suelen obtener a partir de escaneado 3D (LIDAR, o fotogrametría)



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

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 Distance 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 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:

$$\text{SDF}_O(\mathbf{p}) = \|\mathbf{p} - \mathbf{c}\| - r$$

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

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 vídeos 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.

Sección 2.

Modelos de fronteras: mallas de polígonos.

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

Subsección 2.1.
Introducción

Mallas de polígonos.

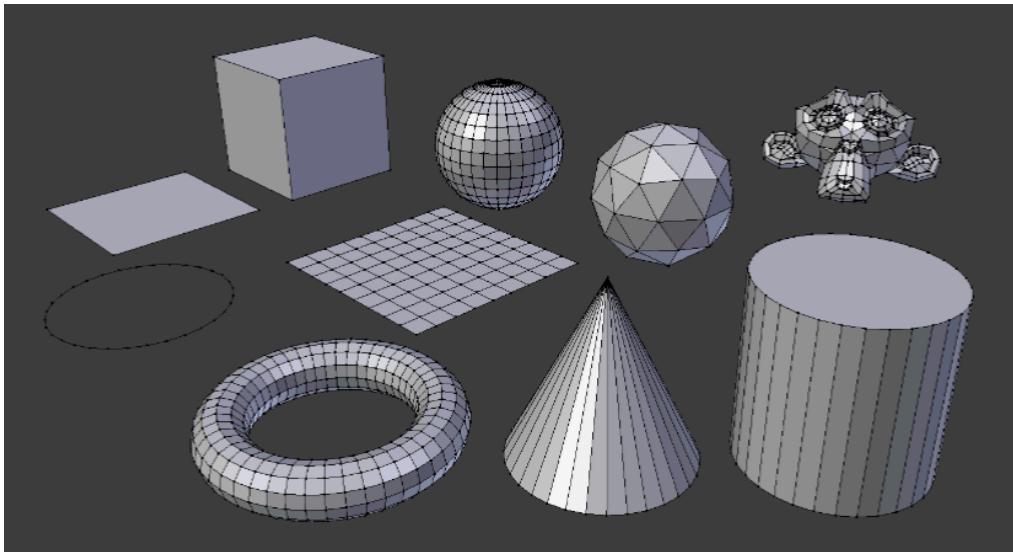
Una **malla de polígonos** (*polygon Mesh*) es un conjunto de puntos de un espacio afín que forman **caras** (*faces*) planas, usualmente adyacentes entre ellas, y que aproxima la frontera de un objeto en el espacio 3D

- 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***.

Ejemplos de mallas

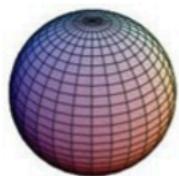
Aquí vemos varios ejemplos de mallas de polígonos (excepto la circunferencia que no lo es). Algunas son cerradas y otras abiertas:



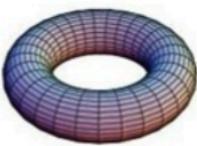
Catálogo de objetos predefinidos de la aplicación *Blender* para modelado 3D:
docs.blender.org/manual/en/latest/modeling/meshes/primitives.html

Características de las mallas

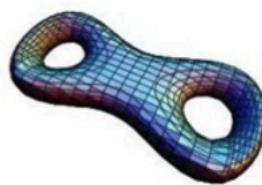
- Las mallas cerradas pueden tener cualquier **género topológico (genus)**, aquí vemos mallas de género 0, 1, 2 y 3.
- Las mallas abiertas pueden tener huecos entre los polígonos:



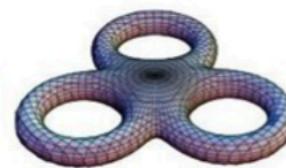
Genus 0



Genus 1



Genus 2



Genus 3



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

Derecha: [Stackoverflow](#)

Subsección 2.2.

Elementos y adyacencia.

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).

- 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.

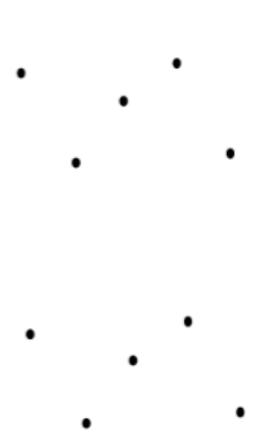
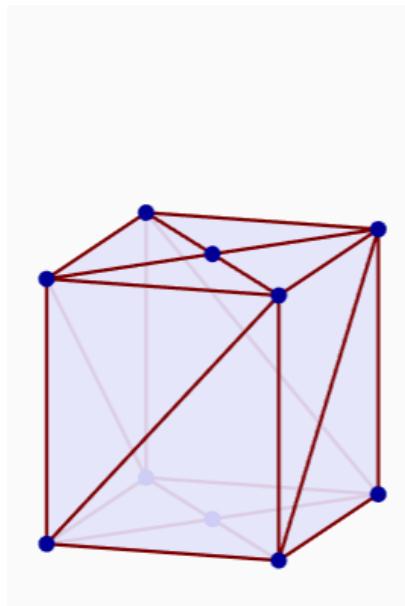
- Los dos índices de vértice de una arista no pueden coincidir.
- El orden en el que aparecen los índices en el par es irrelevante (las aristas no están orientadas).
- Dos aristas distintas no pueden tener el mismo par de índices de vértice, ni siquiera en distinto orden.

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

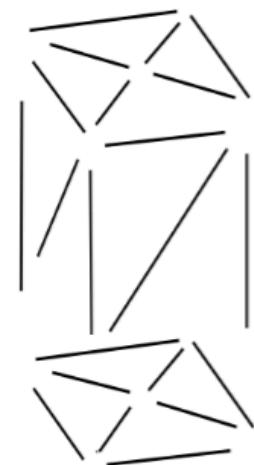
- la secuencia $\{\dot{p}_0, \dot{p}_1, \dots, \dot{p}_{n-1}\}$ de **posiciones** de sus n vértices.
- la secuencia de **caras**, cada una de ellas representada como una secuencia de k índices de vértice: $\{i_0, \dots, i_{k-1}\}$ (k puede ser distinto en cada cara).

Vértices, caras y aristas

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



vertices



edges



faces

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

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.

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).

Características de las 2-variedades

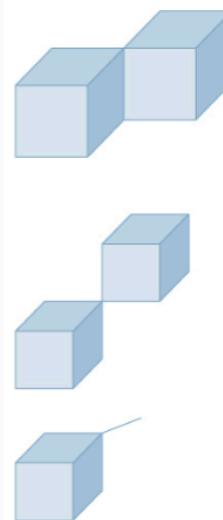
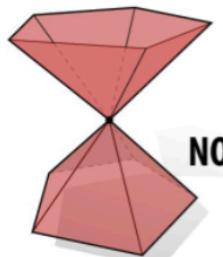
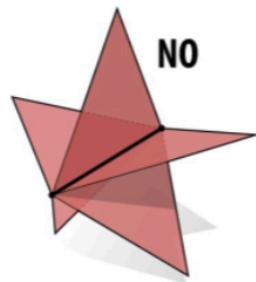
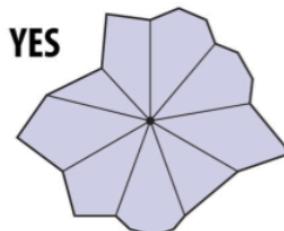
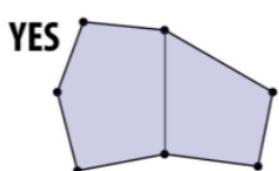
Vamos a usar exclusivamente mallas que son una **2-variedad (2-manifold)**, esto implica que:

- 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.courses.cs.cmu.edu/spring2018/lecture/meshes

Derecha: J.F. Hughes et 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)

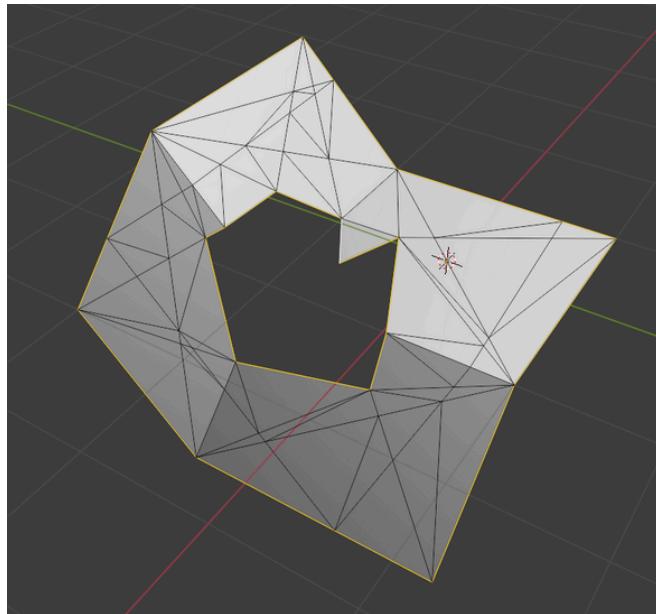


Imagen de StackOverflow:
stackoverflow.com/questions/78359671

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)

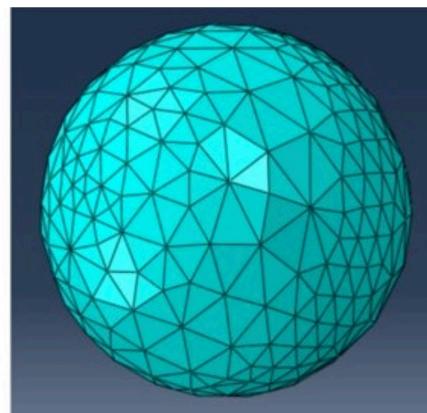
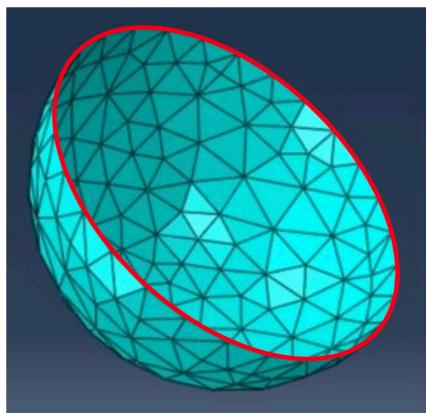


Imagen de Fangtao Yang: [Researchgate](#)

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:

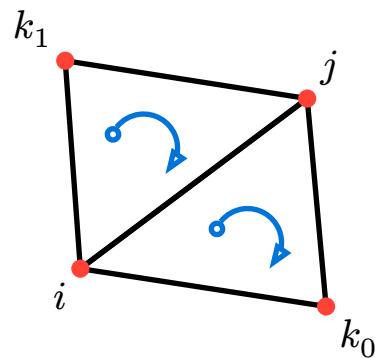
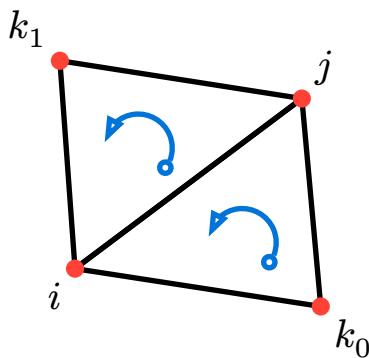
- **Lado horario** es el lado en el cual se ven los vértices en orden de las agujas del reloj.
- **Lado antihorario** es el lado en el cual se ven los vértices en sentido contrario al de las agujas del reloj.

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**

Orientación coherente de las mallas.

Una malla tendrá **orientación coherente** si dadas dos caras adyacentes comparten 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.

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.

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 \dot{p}_i) tiene coordenadas $\mathbf{c}_i = (x_i, y_i, z_i, 1)$ en el marco \mathcal{R} , es decir:

$$\dot{p}_i = \mathcal{R}\mathbf{c}_i = \mathcal{R}(x_i, y_i, z_i, 1)^T$$

- A la tupla \mathbf{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).

Subsección 2.3.

Atributos de vértices.

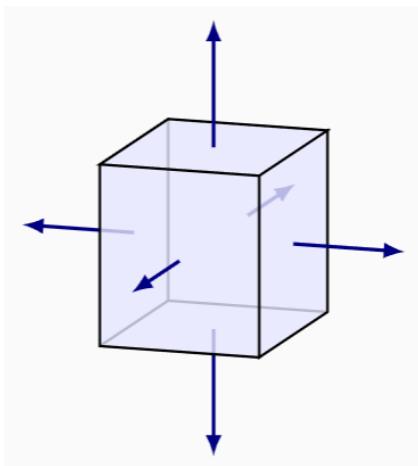
Atributos de las mallas

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...

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:

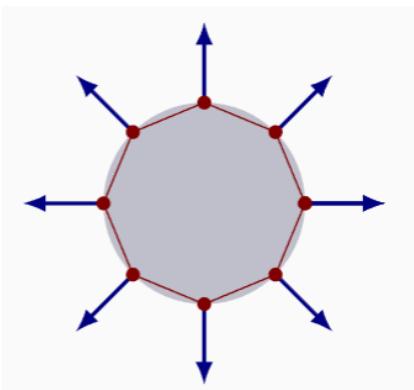
$$\vec{n} = \frac{\vec{m}}{\|\vec{m}\|} \quad \text{donde } \vec{m} = \vec{a} \times \vec{b}$$

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:

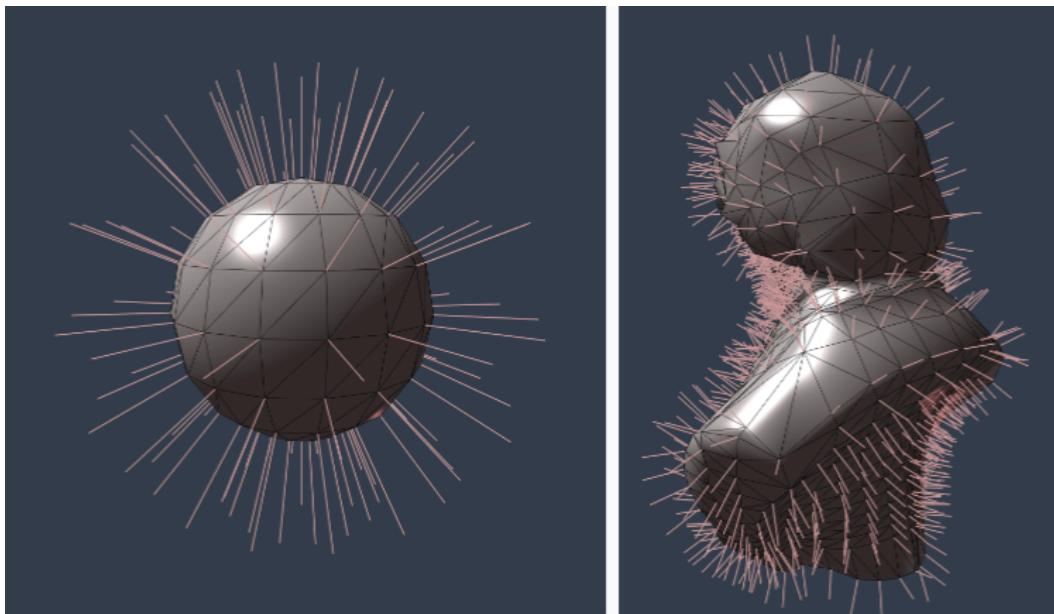
$$\vec{n} = \frac{\vec{s}}{\|\vec{s}\|} \quad \text{donde } \vec{s} = \sum_{i=0}^{k-1} \vec{m}_i$$

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.

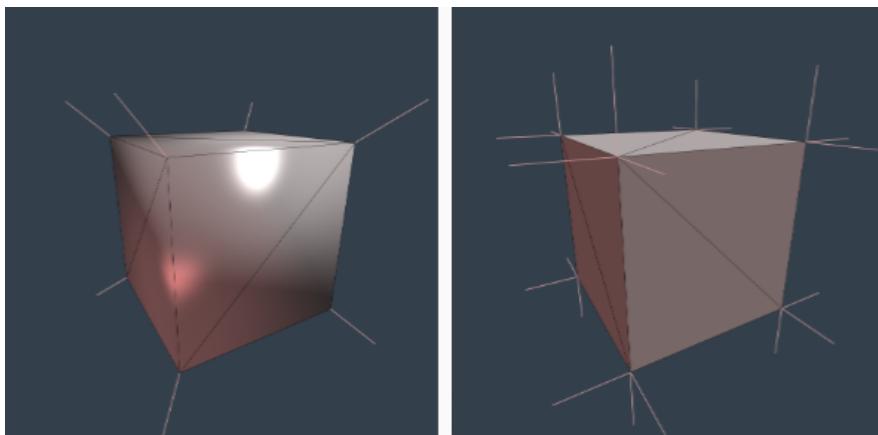
Ejemplos de normales de vértices calculadas

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



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.

- 2. Modelos de fronteras: mallas de polígonos..
- 2.3. Atributos de vértices..

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:

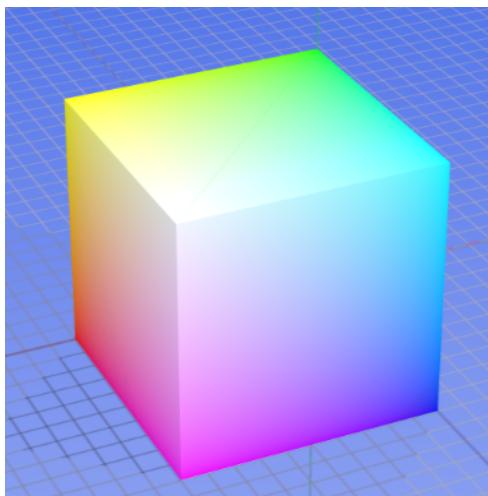


Imagen: en.wikipedia.org/wiki/File:RGB_color_solid_cube.png (Wikimedia Commons)

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 triangulares**, 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)

0	x_0	y_0	z_0
1	x_1	y_1	z_1
2	x_2	y_2	z_2
3	x_3	y_3	z_3
4	x_4	y_4	z_4
5	x_5	y_5	z_5
:	:	:	
$3n - 3$	x_{3n-3}	y_{3n-3}	z_{3n-3}
$3n - 2$	x_{3n-2}	y_{3n-2}	z_{3n-2}
$3n - 1$	x_{3n-1}	y_{3n-1}	z_{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:

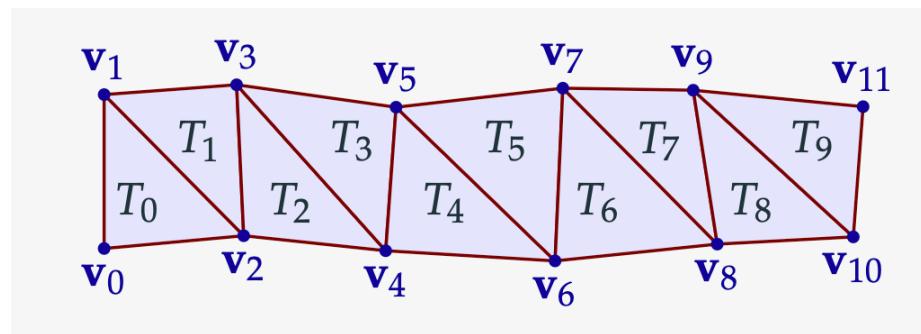
- Para conseguir esto, esta representación reduce el número de veces que aparecen replicadas unas coordenadas en memoria.
- Como consecuencia, se reduce el tiempo de procesamiento.

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.

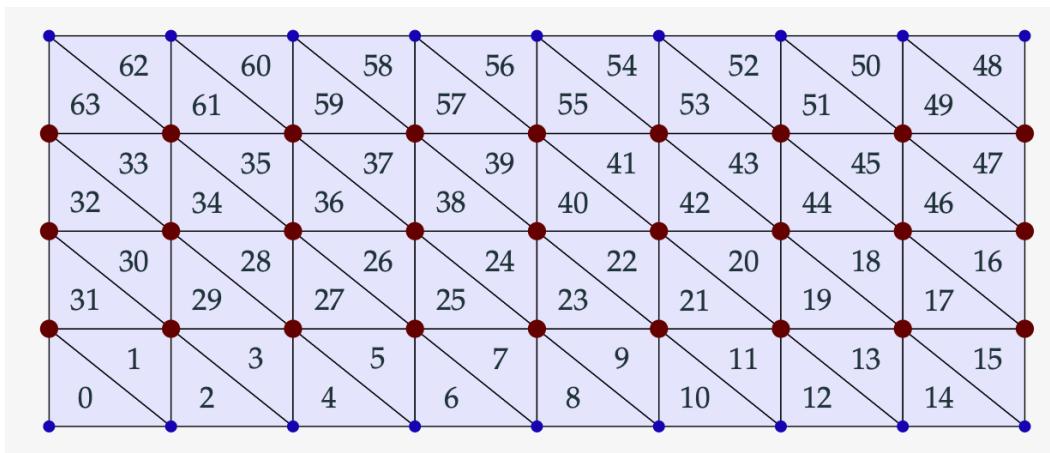
Podemos identificar una parte de una malla como una **tira de triángulos**: cada triángulo T_{i+1} en la secuencia es adyacente al anterior T_i , con lo cual T_{i+1} comparte con T_i una arista y sus dos vértices v_i y v_{i+1} , vértices cuyas coordenadas no tienen que ser repetidas en memoria:



- 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

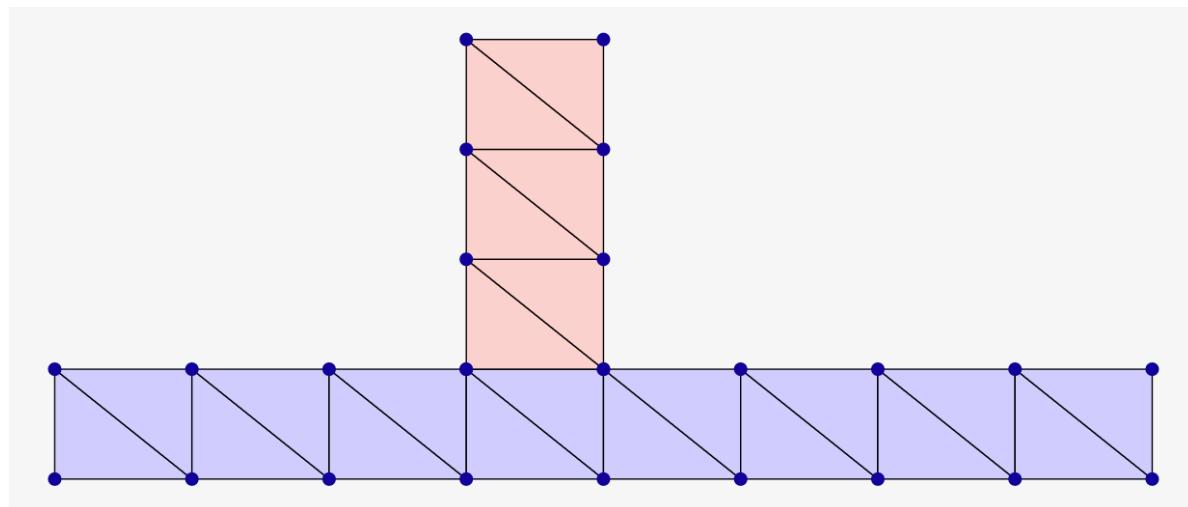
En la mayoría de los casos, las tiras obligan a repetir algunas coordenadas de vértices (aunque en mucho menor grado que los triángulos aislados). Ejemplo de una tira en zig-zag:



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

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.

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)

x_0	y_0	z_0
x_1	y_1	z_1
x_2	y_2	z_2
x_3	y_3	z_3
x_4	y_4	z_4
:	:	:
x_{n_0}	y_{n_0}	z_{n_0}
x_{n_0+1}	y_{n_0+1}	z_{n_0+1}
x_{n_0+2}	y_{n_0+2}	z_{n_0+2}

Tira 1 (n_1 triángulos)

x_0	y_0	z_0
x_1	y_1	z_1
x_2	y_2	z_2
x_3	y_3	z_3
x_4	y_4	z_4
:	:	:
x_{n_1}	y_{n_1}	z_{n_1}
x_{n_1+1}	y_{n_1+1}	z_{n_1+1}
x_{n_1+2}	y_{n_1+2}	z_{n_1+2}

Tira 2 (n_2 triángulos)

x_0	y_0	z_0
:	:	:
x_{n_2+2}	y_{n_2+2}	z_{n_2+2}

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.

Subsección 3.3.

Mallas indexadas de triángulos.

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.

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.)

$i_{0,0}$	$i_{0,1}$	$i_{0,2}$
$i_{1,0}$	$i_{1,1}$	$i_{1,2}$
$i_{2,0}$	$i_{2,1}$	$i_{2,2}$
$i_{3,0}$	$i_{3,1}$	$i_{3,2}$
:	:	:
$i_{n-2,0}$	$i_{n-2,1}$	$i_{n-2,2}$
$i_{n-1,0}$	$i_{n-1,1}$	$i_{n-1,2}$

$$0 \leq i_{jk} < m$$

$$i_{1,2} = 3$$

Tabla Vértices (m verts.)

0	x_0	y_0	z_0
1	x_1	y_1	z_1
2	x_2	y_2	z_2
3	x_3	y_3	z_3
4	x_4	y_4	z_4
:	:	:	:
$m - 2$	x_{m-2}	y_{m-2}	z_{m-2}
$m - 1$	x_{m-1}	y_{m-1}	z_{m-1}

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.

Subsección 3.4.
Aristas aladas

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**

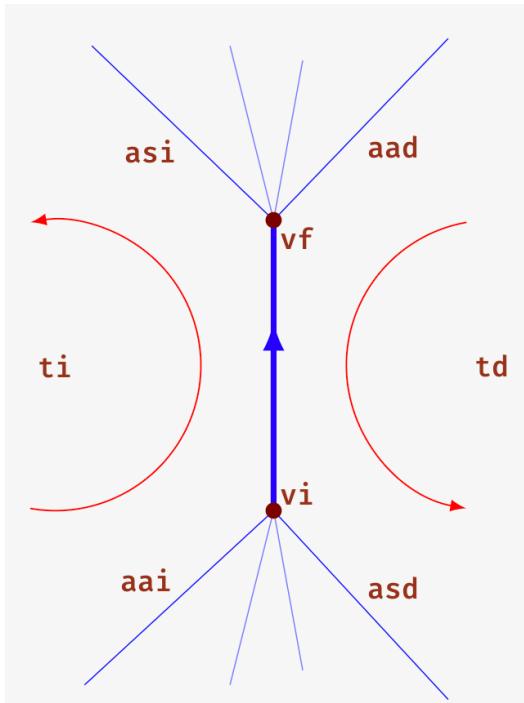
(el recorrido de las aristas se hace en sentido anti-horario cuando se observa el triángulo desde el exterior de la malla).

- 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
...
...
```

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.

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.

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 explicitamente 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.

Sección 4.

Problemas

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:

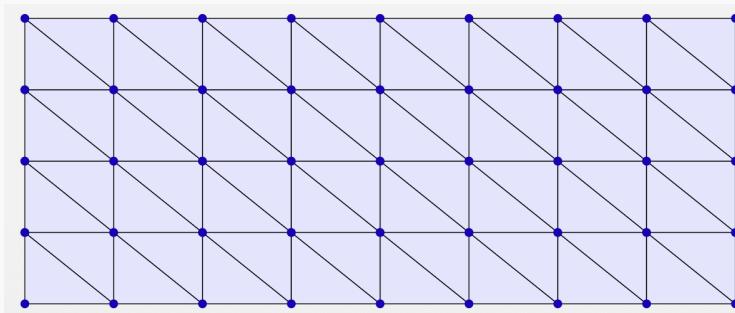
- Expresa el tamaño de ambas representaciones en bytes como una función de k .
- Suponiendo que $k = 16$ calcula cuantos KB de memoria ocupa cada estructura.
- Haz lo mismo asumiendo ahora que $k = 1024$ (expresa los resultados en MB)

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):

- (a) Indica que cantidad de memoria ocupa esta representación, en estos dos casos:
- (1) Como función de n y m , en bytes.
 - (2) Suponiendo $m = n = 128$, en KB.
- (b) 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.
- (c) 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.

Informática Gráfica. **Sesión 5: Modelos Jerárquicos.**

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

Índice

Modelos Jerárquicos	3
Grafos de escena en Godot	10
Ejemplo de un árbol 2D	36
Problemas	57

Sección 1.
Modelos Jerárquicos

1. Grafos de escena.

Subsección 1.1.

Grafos de escena.

Modelos jerárquicos.

En Informática Gráfica, un **Modelo Jerárquico** es una estructura de datos en forma de grafo que representa las relaciones espaciales entre las **componentes** de una aplicación interactiva.

- Es una herramienta fundamental que permite diseñar y gestionar modelos complejos mediante el uso de **componentes complejas constituidas de otras componentes más simples**
- Una **componente** es un objeto geométrico 2D o 3D sencillo, como una malla, o un grupo de varias componentes.
 - ▶ Permite la **reutilización** de componentes en distintas partes de un proyecto o en distintos proyectos.
- Permite a distintos desarrolladores (o al mismo desarrollador en distintos instantes) trabajar en diferentes componentes de un proyecto sin interferir entre sí, facilitando la colaboración en proyectos grandes.

Grafos de escena.

Un **Grafo de Escena** es un **Grafo Dirigido Acíclico**, cada uno de cuyos nodos contiene un modelo de un objeto. El grafo en sí es también un modelo (jerárquico) que representa un objeto compuesto (llamado **escena**) formado por **réplicas de los objetos cuyos modelos están en los nodos**.

- Un nodo N del grafo puede ser:

Terminal: si objeto asociado a N no está compuesto de otros objetos, típicamente mallas con vértices en determinadas posiciones en el espacio.

No terminal: si el objeto está compuesto de otros objetos en otros nodos del grafo, nodos llamados **nodos hijos** de N .

- En el grafo hay un **arco dirigido** desde cada nodo padre a cada uno de sus nodos hijos. Cada arco tiene asociada una **transformación geométrica** (afín).
- Todo nodo del grafo tiene al menos un parent (o más), excepto exactamente un nodo sin padres, que es el **nodo raíz**.

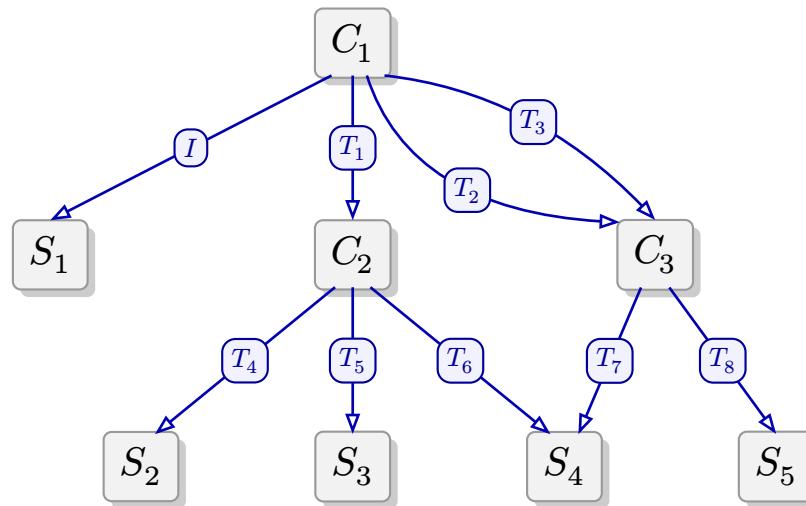
Marcos afines y transformaciones asociados a los nodos.

La escena asociada a un grafo está compuesta de **objetos instanciados**.

- Un **objeto instanciado** es una réplica del objeto asociado a un nodo del grafo, pero con las coordenadas de sus vértices transformadas por una transformación afín propia de esa réplica.
- Un objeto puede aparecer instanciado varias veces en una escena, cada instancia puede tener asociada una transformación afín distinta.
- Las coordenadas de los vértices de un nodo son relativas a un marco afín propio de cada nodo, llamado **marco local** del nodo. A las coordenadas se les llama **coordenadas locales**.
- El marco local del nodo raíz se llama **marco de escena** y sus coordenadas se llaman **coordenadas de escena**.
- La transformación asociada a cada nodo determina como se convierten las coordenadas locales del nodo (relativas al marco del nodo) en coordenadas de escena (relativas al marco de escena).

Ejemplo de grafo de escena

En un grafo de escena, cada arco dirigido tiene asociada una transformación afín:



- cada objeto compuesto de otros es un **nodo no terminal** (C_1, C_2, C_3)
- cada objeto simple (no compuesto) es un **nodo terminal** (S_1, S_2, S_3, S_4, S_5)
- cada arco se etiqueta con una transformación geométrica (I, T_1, T_2, \dots, T_8)

Instancias de objetos en el grafo de ejemplo

En la escena representada por un grafo **hay una instancia de cada nodo por cada camino posible desde la raíz al nodo**. Esa instancia tiene asociada la transformación que resulta de componer las transformaciones de los arcos del camino (de izquierda a derecha según se va desde la raíz al nodo).

A modo de ejemplo, en el grafo anterior hay las siguientes instancias de objetos terminales (no compuestos):

- Una instancia de S_1 , con la transformación I (la transformación identidad).
- Una instancia de S_2 , con la transformación $T_1 \circ T_4$.
- Una instancia de S_3 , con la transformación $T_1 \circ T_5$.
- Tres instancias de S_4 , con las transformaciones $T_1 \circ T_6$, $T_2 \circ T_7$ y $T_3 \circ T_7$.
- Dos instancias de S_5 , con las transformaciones $T_2 \circ T_8$ y $T_3 \circ T_8$.

En total hay 8 instancias de objetos terminales. Además, si se permite que los nodos no terminales tengan mallas (además de hijos), habría que añadir 3 instancias de los objetos C_2 y C_3 .

Sección 2.

Grafos de escena en Godot.

1. Escenas y nodos.
2. Transformaciones de los nodos.
3. La transformación de los nodos 2D.
4. La transformación de los nodos 3D.
5. Creación y actualización de árboles en tiempo de ejecución.

Subsección 2.1.

Escenas y nodos.

Proyectos, escenas y nodos en Godot

El desarrollo de aplicaciones gráficas en Godot se basa en los conceptos de **proyecto, escena y nodo**.

- Un **proyecto** es un conjunto de archivos que se usan para construir una aplicación. Se guardan en una *carpeta del proyecto* donde, entre otros, habrá un archivo **.godot** con datos del mismo.
- Una **escena** es un árbol de nodos, que se llama **árbol de escena**, con al menos un nodo. Una escena siempre tiene asociado un **nodo raíz** de la misma.
- Un proyecto incluye **una o varias escenas**. Una de las escenas será la **escena principal** del proyecto. En la carpeta del proyecto habrá un archivo **.tscn** por cada escena de dicho proyecto.
- Un **nodo** es un objeto (instancia de alguna clase Godot) que representa un elemento de la aplicación. Un nodo se incluye en una única escena, y aparece una sola vez en el árbol de dicha escena, por tanto un nodo tiene un único parent (excepto el nodo raíz de una escena).
- El nodo raíz de la escena principal se considera el **nodo raíz del proyecto**.

Clasificación de los nodos

Los nodos puede clasificarse en estas dos categorías:

- Nodos **regulares**: son objetos instancias de la clase **Node** o sus derivadas, principalmente:
 - ▶ **CanvasItem**: para objetos 2D, puede ser de la clase **Node2D** o derivadas (objetos visibles) o **Control** (objetos de interfaz gráfica).
 - ▶ **Node3D**: objetos que contienen, entre otras cosas, mallas 3D, cámaras, fuentes de luz etc...
- Nodos **instancia de escena**: guardan **una referencia a una escena del proyecto** (que no puede ser la escena principal del proyecto). Este nodo es único en el árbol, pero varios nodos de este tipo pueden tener referencias a la misma escena.

Así que la única forma de que un nodo pueda instanciarse más de una vez en un grafo es hacer que ese nodo sea una escena de Godot.

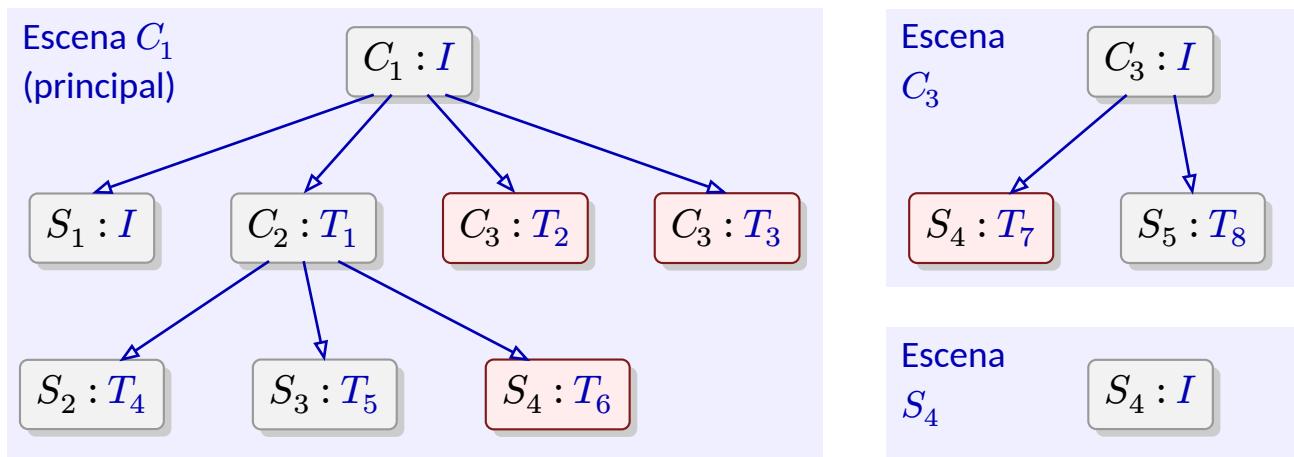
Instancias de nodos y escenas

En un árbol de escena:

- Cada nodo en un árbol de una escena tiene exactamente un parente (excepto el nodo raíz de una escena).
- Para cada escena (distinta de la escena principal del proyecto), puede haber varios nodos de tipo **instancia de escena** que refieran a esa escena, por tanto una escena puede estar:
 - ▶ instanciada en varios árboles de escena diferentes, o además
 - ▶ instanciada varias veces en un mismo árbol de escena.
- En el panel con el *árbol de escena* (arriba a la izquierda del editor), los nodos sub-escena aparecen como un nodo normal, pero con un icono de una placa. No se pueden expandir.
- En el panel del *sistema de archivos* (abajo a la izquierda del editor), podemos ver un archivo de extensión **.tscn** que guarda información de una escena. Si lo seleccionamos, en el panel *árbol de escena* pasaremos a ver el árbol de escena de esa escena.

Esquema de varias escenas de un proyecto Godot

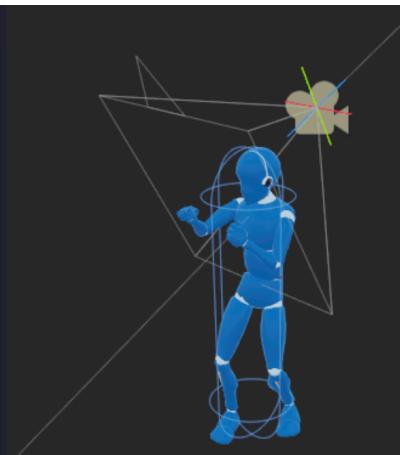
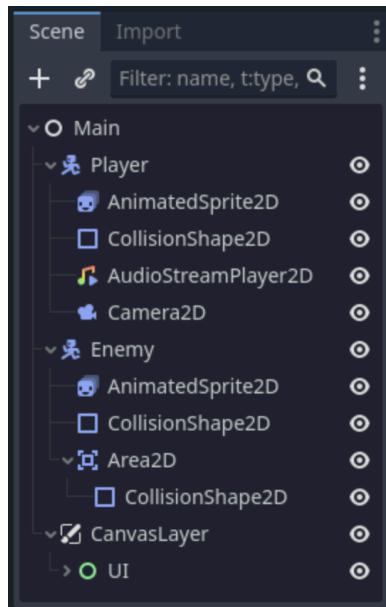
Vemos aquí las escenas equivalentes (en Godot) al grafo de ejemplo que ya vimos, pero ahora cada nodo tiene asociada una transformación (después de :) y los nodos instancia de escena tienen fondo rosa.



Hay tres escenas, cuyas raíces son: C_1 , C_3 y S_4 . Godot ignora las transformaciones de los nodos raíz (ya que se redefinen en las instancias), excepto la transformación del nodo raíz de la escena principal.

Ejemplos de árboles de escena de Godot.

Aquí vemos dos ejemplos de dos árboles de escena de Godot.



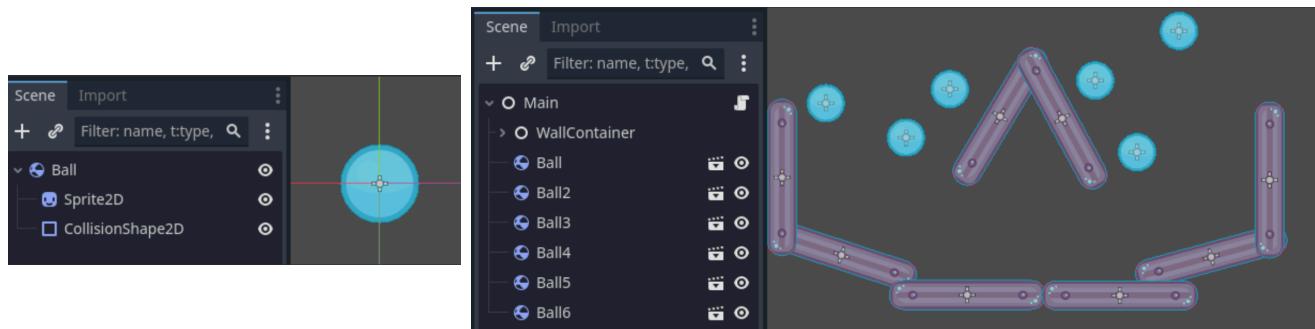
Imágenes obtenidas obtenidas de la documentación oficial de Godot:

docs.godotengine.org/es/4.5/getting_started/step_by_step/nodes_and_scenes.html

Escena instanciada en otra

A la izquierda vemos el disco azul que está modelado en la escena **Ball**, es un árbol formado por tres nodos.

A la derecha, los nodos **Ball**, **Ball2**, **Ball3**, etc.... son instancias de la escena **Ball**, cada una con su propia transformación (en concreto, están trasladadas a distintas posiciones del plano). En el panel aparecen con un icono de plaqeta.



Imágenes obtenidas de la documentación oficial de Godot:

docs.godotengine.org/es/4.5/getting_started/step_by_step/instancing.html

Reutilización de mallas

En las aplicaciones gráficas, los objetos tipo **Mesh** (mallas 2D o 3D) contienen los vértices y las coordenadas de textura, las normales, etc... y por tanto pueden ocupar mucha memoria. Por tanto:

- Se hace necesario evitar la duplicación de mallas en memoria, por ejemplo con dos nodos que incluyan la misma malla.
- Se pueden usar nodos de tipo **MeshInstance2D** o **MeshInstance3D** que contienen una referencia a una malla (propiedad **mesh**), pero no la malla en sí.
- En una aplicación puede haber distintos nodos referenciando la misma malla.
- Puesto que la clase **Mesh** es derivada de **RefCounted** (objetos con cuenta de referencias), en Godot se gestiona automáticamente la memoria de las mallas, siendo esta liberada cuando no hay más referencias a la misma.

Subsección 2.2.

Transformaciones de los nodos.

El marco y la transformación asociados a un nodo.

Sea N un nodo (**Node2D** y **Node3D**) situado en un árbol de escena de Godot:

- El nodo N siempre asociado un marco afín (2D o 3D) que se llama **marco del nodo N** , lo llamamos \mathcal{N}
- El nodo N también tendrá asociado un **marco padre**, lo llamamos \mathcal{P} :
 - ▶ Si N no es el nodo raíz de una escena, \mathcal{P} es el marco del único nodo padre de N en el árbol de esa escena.
 - ▶ Si N es nodo raíz de una escena, entonces \mathcal{P} es el llamado **marco global de la escena**. Si esa escena es la escena principal de Godot, entonces a \mathcal{P} también se le llama **marco del mundo**.
- La transformación que convierte el marco \mathcal{P} en el marco \mathcal{N} se representa mediante una matriz M_N llamada **transformación (o matriz) del nodo**, tiene en sus **columnas** la base y el origen del marco \mathcal{N} , **expresadas en coordenadas relativas al marco \mathcal{P}** . Esto implica que

$$\mathcal{P}M_N = \mathcal{N}$$

El espacio de coordenadas local y el espacio padre

En relación a las coordenadas de los vértices en las mallas u objetos guardados en un nodo N :

- Se considera que están siempre expresadas en el marco \mathcal{N} del nodo, y se llaman **coordenadas locales de N** , también se dice que están expresadas en el **espacio de coordenadas local del nodo**.
- Al aplicarseles la transformación M_N , se convierten en coordenadas relativas al marco \mathcal{P} , se dice que esas coordenadas están expresadas en el **espacio de coordenadas padre de N** .
- En última instancia, todas las coordenadas de los vértices en una aplicación Godot acaban convertidas en la GPU en coordenadas relativas al marco global de la escena principal o marco del mundo, esas son **coordenadas de mundo**.
- Las coordenadas del mundo se usan por Godot para proyectar los vértices en pantalla y producir la imagen por rasterización en la GPU.

La propiedad **transform** de un nodo en Godot

Para cualquier nodo N , el objeto asociado tiene una propiedad de tipo **Transform2D** o **Transform3D**, llamada **transform**, que guarda la matriz asociada M_N .

La matriz de un nodo N se puede modificar asignando o actualizando su **transform**, se puede hacer de estas formas:

- En el editor, podemos cambiar el valor inicial de **transform** (le asignamos una posición, escalado o rotaciones).
- En tiempo de ejecución del videojuego, podemos modificar $N.\text{transform}$ directamente en el código GDScript. Se puede hacer de dos formas:
 - ▶ Mediante asignaciones a $N.\text{transform}$ o sus componentes.
 - ▶ Mediante asignaciones a propiedades relacionadas con **transform** (**position**, **rotation**, etc..., ver siguiente transparencia).
 - ▶ Usando métodos las clases **Node2D** o **Node3D** los cuales, aplicados a un nodo N , modifican $N.\text{transform}$.

Subsección 2.3.

La transformación de los nodos 2D.

Atributos de transformación de los nodos 2D

Para cada nodo, en el panel de la derecha del editor podemos ver las propiedades que definen su transformación inicialmente. Godot mantiene los atributos siempre coherentes con la matriz **transform** del nodo. Los atributos son:

position: vector 2D (traslación)

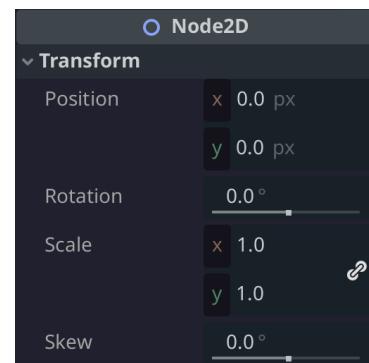
rotation: ángulo de rotación en radianes

scale: vector 2D (factores de escala)

skew: real, ángulo de skew en grados (tipo de cizalla)

La matriz **transform** se corresponde con la composición de estas transformaciones (en este orden de derecha a izquierda):

1. Escalado por los factores **scale**
2. Skew por el ángulo **skew**
3. Rotación de **rotation** radianes
4. Traslación por vector **position**



Actualización de la transformación 2D en un script

Las propiedades de transformación se pueden cambiar en tiempo de ejecución. El siguiente código siempre produce el mismo resultado, independientemente del valor de **asignar_transform** y de las variables declaradas en el código

```
var factores_escala      := Vector2( 2.0, 1.0 )
var angulo_rot_radianes := 1.0
var vector_traslacion   := Vector2( 1.0, 0.0 )

if asignar_transform :
    var esc := Transform2D().scaled( factores_escala )
    var ske := Transform2D() ## identidad (skew nulo)
    var rot := Transform2D().rotated( angulo_rot_radianes )
    var tra := Transform2D().translated( vector_traslacion )
    transform = tra * rot * ske * esc ## el orden es esencial
else:
    scale     = factores_escala      ## (1)
    skew     = 0.0                  ## (2)
    rotation = angulo_rot_radianes ## (3)
    position = vector_traslacion  ## (4)
```

Modificación de la transformación de un nodo 2D.

Estos métodos, aplicados a un nodo N , modifican las propiedades de transformación, y después se recalcula la matriz **transform** del nodo.

- $N.\text{rotate}(\theta)$: añade θ radianes a la propiedad **rotation**, es decir, es equivalente a:

```
rotation += theta
```

- $N.\text{apply_scale}(s)$: multiplica los factores en la propiedad **scale** por los dos factores en s .

```
scale = Vector2( scale.x * s.x, scale.y * s.y )
```

- $N.\text{translate}(t)$: suma el vector t a la propiedad **position**.

```
position += vt
```

Subsección 2.4.

La transformación de los nodos 3D.

Atributos de transformación de los nodos 3D

Godot también tiene propiedades que definen la transformación de un nodo 3D. En el caso 3D no hay *skew* y además la rotación es la composición de 3 rotaciones entorno a los ejes en el orden Y, X, Z (aunque se puede cambiar el orden, o usar un cuaternión, o dar directamente los ejes transformados).

position: vector 3D (traslación)

rotation: vector con tres ángulos de rotación (Y, X, Z)

scale: vector 3D (3 factores de escala)

La matriz **transform** se corresponde con la composición de estas transformaciones (en este orden de derecha a izquierda):

1. Escalado por los factores **scale**
2. Rotaciones elementales con los ángulos contenidos en **rotation** (en radianes).
3. Traslación por vector **position**



Actualización de la transformación: rotaciones por la izquierda.

Al igual que en 2D, en 3D existen diversos métodos que permiten modificar las propiedades **rotation**, **scale** de un nodo 3D. En estos casos se componen matrices de rotación por la izquierda de la matriz actual.

- **N.rotate(v, θ)**: compone la rotación codificada en **rotation** con una rotación entorno al eje **v** (del padre). Si **position** es el vector nulo, entonces es equivalente a:

```
var rot := Transform3D().rotated( v, theta )
N.transform = rot * N.transform
```

- **N.rotate_x(θ)**: idem, donde **v** es el eje X.
- **N.rotate_y(θ)**: idem, eje Y.
- **N.rotate_z(θ)**: idem, eje Z.

Al usarse composición por la izquierda, el vector 3D **v** con el eje de rotación se interpreta como expresado en **el marco de coordenadas del padre**.

Actualización de la transformación: composición por la derecha.

Existen otros métodos que permite componer matrices por la derecha, es decir, la matriz que se compone actúa sobre las coordenadas **antes** que la transformación previa, es decir, se aplica a las coordenadas locales en primer lugar. Equivalente a:

```
var M : Transform3D = ... ## la matriz que se quiere componer  
N.transform = N.transform * M
```

- *N.rotate_object_local(v,θ)*: $M :=$ rotación de θ entorno al eje v
- *N.translate_object_local(t)*: $M :=$ traslación por el vector t
- *N.translate(t)*: equivalente a la anterior.
- *N.scale_object_local(s)*: $M :=$ matriz de escalado con factores s.

Al usarse composición por la derecha, eso implica que las tuplas v,t y s (de tipo **Vector3**) que se indican aquí se interpretan como expresadas en el **marco de coordenadas local del nodo N**.

Subsección 2.5.

Creación y actualización de árboles en tiempo de ejecución.

Creación, inserción y consulta de nodos en un árbol.

Los nodos se pueden crear en tiempo de ejecución con el método de clase `new()`. Si el constructor tuviese parámetros, habría que proporcionarlos como argumentos de `new()`. Por ejemplo, para crear un nodo de tipo `Node2D`:

```
var nodo := Node2D.new() ## al crearlo está "huérfano"
```

Para añadir un nodo (por ejemplo `hijo`) al árbol de escena, habría que usar el método `add_child()` del nodo padre (nodo `padre`). Lo añade como último nodo hijo. Por ejemplo:

```
padre.add_child( hijo )
```

Los hijos directos de un nodo padre *p* se pueden consultar con estos métodos:

- `p.get_child_count()`: devuelve el número de hijos del nodo.
- `p.get_child(i)`: devuelve el hijo número *i* (entero, empezando en 0).
- `p.get_children()`: devuelve un `Array` con todos los hijos del nodo.

Nombres de los nodos y búsqueda

Todo nodo tiene una propiedad **name** (cadena de caracteres, **String**) que se puede asignar y consultar en el editor o en tiempo de ejecución con scripts. Sirve para **identificar un nodo en el árbol**, y debe ser único entre los nodos hijos de un mismo padre. Los métodos para buscar nodos en un árbol son:

- ***p.get_node(s)***: devuelve un nodo descendiente (directo o indirecto) del nodo *p*, siguiendo la ruta (*path*) dada en la cadena *s*. La ruta tiene nombres de nodos separados por **/**, empezando por un hijo de *p*. Si no se encuentra da error. Aquí se obtiene el nodo de nombre «*bisnieto*»:

```
var nodo := n.get_node("hijo/nieto/bisnieto")
var nodo := $hijo/nieto/bisnieto ## forma equivalente.
```

- ***p.get_node_or_null(s)***: igual que el anterior, pero si no encuentra nada devuelve **null** en vez de dar error.
- ***p.find_node(s)***: igual que el anterior, pero permite caracteres comodín ***** y **?** en *s* y devuelve el primer nodo que encaja con la cadena, en un recorrido en profundidad del árbol. Si no encuentra nada, devuelve **null**.

Desconexión y destrucción de nodos

Se puede **desconectar un nodo hijo** h de su nodo padre p con el método $p.\text{remove_child}(h)$. Esto lo elimina del árbol, pero no se destruye el nodo (aunque queda *huérfano* con seguridad, ya que todo nodo tiene un único parente, y no será visible en la escena).

Para **destruir** un nodo huérfano (es decir, liberar la memoria que ocupa ese nodo y todos sus hijos), se usa el método $n.\text{queue_free}()$. Esto registra la solicitud de eliminar el nodo, lo cual ocurrirá al final del siguiente frame.

A modo de ejemplo, para desconectar y destruir un nodo hijo (con nombre «*borrar*») de un nodo parente p , haríamos:

```
var h := p.get_node("borrar")
p.remove_child( h ) ## queda huérfano
h.queue_free()      ## se puede usar hasta el final del siguiente frame.
```

Objetos de tipo Mesh compartidos

Como se ha indicado, en un árbol es conveniente **no replicar objetos Mesh complejos**. A modo de ejemplo, este código crea un **ArrayMesh** y dos **MeshInstance3D** que lo comparten. Cada **MeshInstance3D** está instanciado en una posición distinta:

```
var tablas := []
tablas.resize( Mesh.ARRAY_MAX )
GenerarTablas( tablas ) ## función que genera vértices, normales, etc...
var am := ArrayMesh.new()
am.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )

var instancia1 := MeshInstance3D.new()
instancia1.mesh      = am
instancia1.position = Vector3( -3.0, 0.0, 0.0 )

var instancia2 := MeshInstance3D.new()
instancia2.mesh      = am
instancia2.position = Vector3( +3.0, 0.0, 0.0 )
```

Sección 3.

Ejemplo de un árbol 2D

1. Diseño del grafo.
2. Implementación en Godot.
3. Implementación de Grafos parametrizados (y animados).

Subsección 3.1.

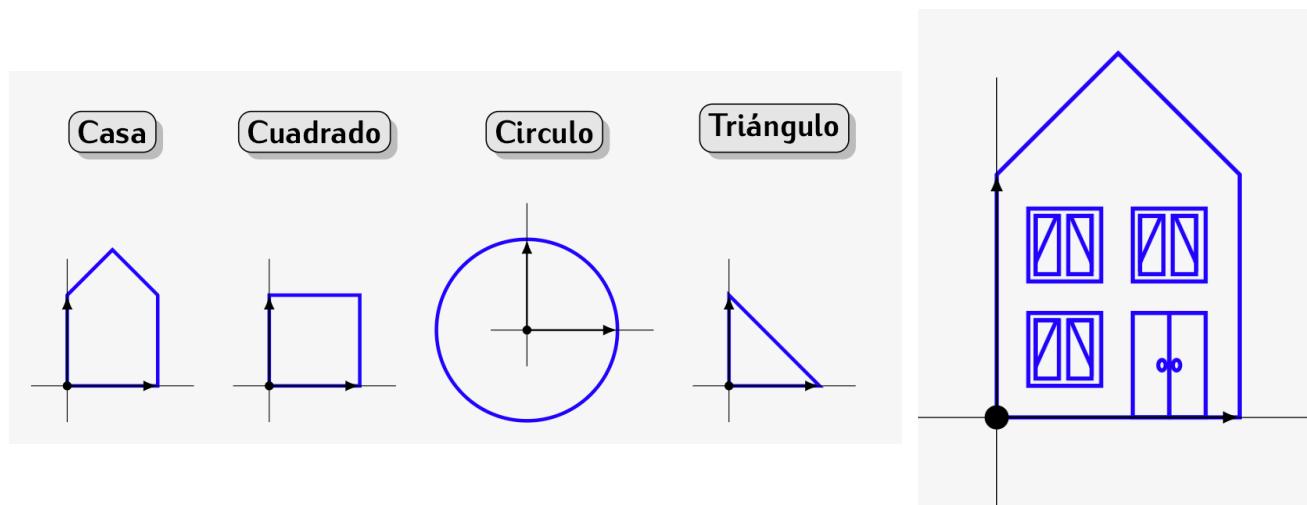
Diseño del grafo.

3. Ejemplo de un árbol 2D.

3.1. Diseño del grafo..

Objetos simples

Supongamos que queremos construir una figura como la de la derecha, usando varios objeto **ArrayMesh** simples (un cuadrado, un triángulo y un círculo), tal y como aparecen a la izquierda.



Cada objeto aparece junto a su marco de referencia local, de forma que podamos entender mejor las transformaciones.

Notación abreviada para grafos de escena

Para este diseño, usaremos una notación más expresiva para especificar el grafo de escena:

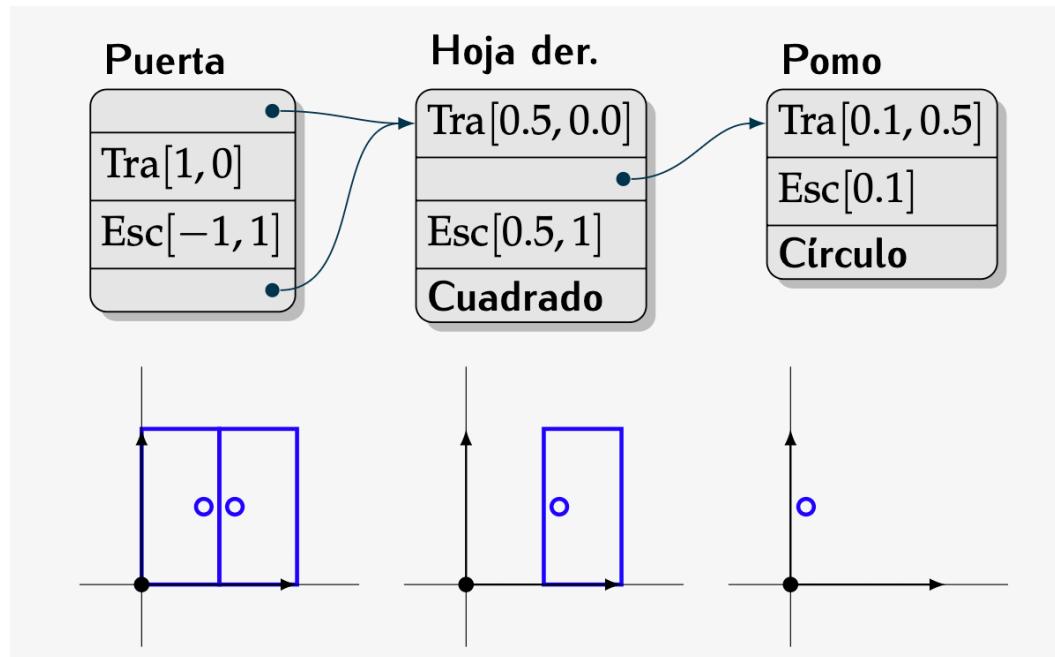
- Usaremos un grafo dirigido acíclico, en vez de un simple árbol.
- Los nodos son listas de:
 - ▶ Elementos simples (como los de la transparencia anterior), en negrita.
 - ▶ Instancias de otros nodos (subárboles), con una flecha a otro nodo.
 - ▶ Transformaciones: afectando a todas las entradas de la lista que le siguen en el nodo (y se aplican de abajo hacia arriba).
- La implementación de estos grafos en Godot requiere convertirlos en árboles, bien duplicando nodos, bien usando instancias de escenas.
- En nuestro caso, usaremos nodos duplicados, aunque compartirán objetos **ArrayMesh**.

3. Ejemplo de un árbol 2D.

3.1. Diseño del grafo..

Objetos puerta, hoja derecha y pomo.

El nodo **Puerta**, contiene dos instancias de un nodo llamado **HojaDerecha**, que a su vez tiene un objeto **Pomo**

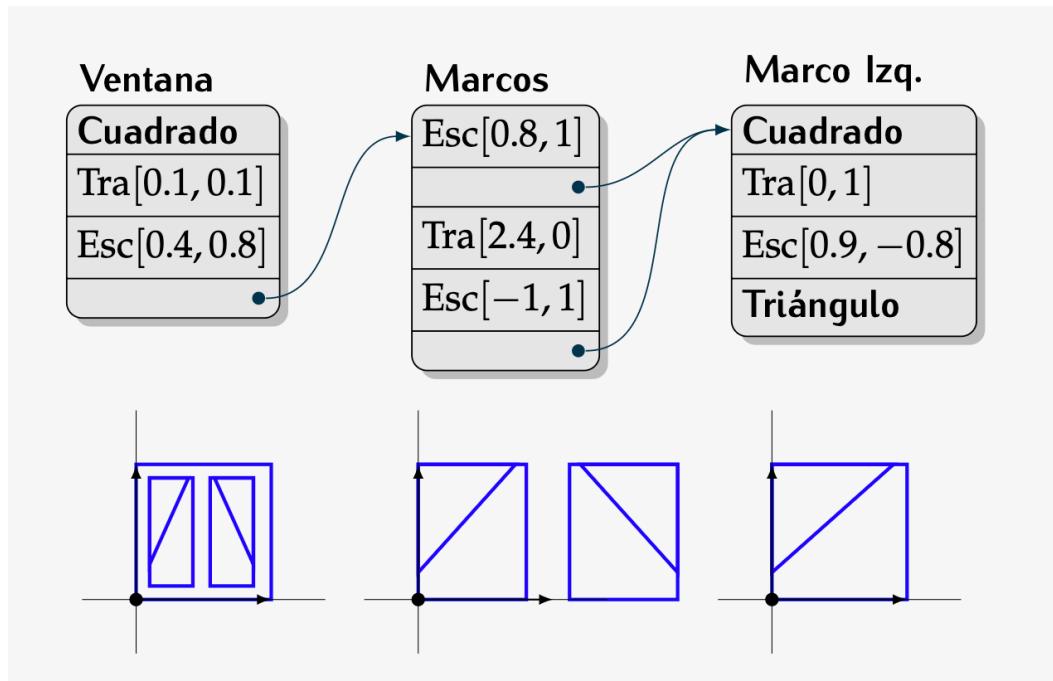


3. Ejemplo de un árbol 2D.

3.1. Diseño del grafo..

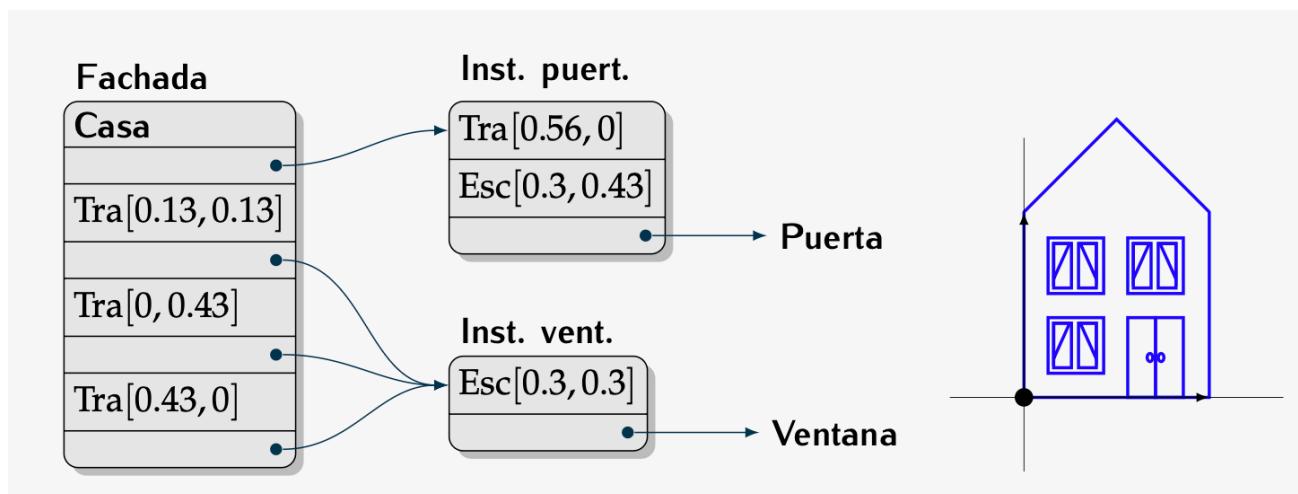
Objetos ventana, marcos y marco izquierdo

Objetos **Ventana**, compuesto de un **Cuadrado** y un objeto **Marcos**, a su vez compuesto de un **MarcoIzquierdo** (que es un **Cuadrado** y un **Triángulo**):



Objeto fachada, instancias de puertas y de ventana

Objeto **Fachada**, que es la raíz del árbol de escena. Contiene los objetos: **Casa**, **InstanciaPuerta** e **InstanciaVentana** (que a su vez incluyen **Puerta** y **Ventana**, respectivamente).



Subsección 3.2.

Implementación en Godot.

Estrategias de implementación.

El diseño del grafo de escena puede ser implementado en Godot de diversas formas:

- Creando los nodos en el editor, sin scripts, excepto para crear los objetos **ArrayMesh**
- Creación en tiempo de ejecución con uno o varios scripts.
- Combinación de ambos: algunos nodos creados en el editor, y otros en tiempo de ejecución.

En nuestro caso, usaremos un **único script asociado al nodo raíz**. Dicho nodo raíz se puede crear en el editor, y después, en tiempo de ejecución, en la función **_ready()**, se crean todos los nodos descendientes.

Funciones auxiliares.

Esta función crea un objeto **ArrayMesh** a partir de un array de posiciones de vértices (con tipo de primitiva polilínea abierta, **PRIMITIVE_LINE_STRIP**):

```
func CrearArrayMesh( v : PackedVector2Array ) -> ArrayMesh :
    var tablas : Array = [] ; tablas.resize( Mesh.ARRAY_MAX )
    tablas[ Mesh.ARRAY_VERTEX ] = v
    var am : ArrayMesh = ArrayMesh.new()
    am.add_surface_from_arrays( Mesh.PRIMITIVE_LINE_STRIP, tablas )
    return am
```

Esta función crea un nodo **MeshInstance2D** a partir de un **ArrayMesh** dado:

```
func CrearMeshInstance2D( am : ArrayMesh, tr : Transform2D ) ->
    MeshInstance2D:
    var mi := MeshInstance2D.new()
    mi.transform = tr
    mi.modulate = Color( 0.0, 0.0, 0.7 ) ## azul
    mi.mesh = am
    return mi
```

Objeto simples: cuadrado, triángulo, casa

Los objetos simples se puede crear como variables únicas, de tipo **ArrayMesh**, de forma que se compartan entre todas las instancias que los usen.

```
var cuadrado := CrearArrayMesh( PackedVector2Array([
    Vector2( 0.0, 0.0 ), Vector2( 1.0, 0.0 ),
    Vector2( 1.0, 1.0 ), Vector2( 0.0, 1.0 ),
    Vector2( 0.0, 0.0 )
]))  
  
var triangulo := CrearArrayMesh( PackedVector2Array([
    Vector2( 0.0, 0.0 ), Vector2( 1.0, 0.0 ), Vector2( 0.0, 1.0 ),
    Vector2( 0.0, 0.0 )
]))  
  
var casa := CrearArrayMesh( PackedVector2Array([
    Vector2( 0.0, 0.0 ), Vector2( 1.0, 0.0 ),
    Vector2( 1.0, 1.0 ), Vector2( 0.5, 1.4 ), Vector2( 0.0, 1.0 ),
    Vector2( 0.0, 0.0 )
]))
```

3. Ejemplo de un árbol 2D.

3.2. Implementación en Godot..

Objetos simples: circunferencia

La variable correspondiente a la circunferencia se puede crear así

```
var circunferencia : ArrayMesh = CrearCircunferencia( 64 )
```

Se usa una función auxiliar que genera los vértices de una circunferencia de radio unidad, con el número de segmentos indicado:

```
func CrearCircunferencia( n : int ) -> ArrayMesh :  
    var v := PackedVector2Array()  
    for i in range( n+1 ):  
        var a : float = (float(i) * 2.0 * PI )/ float(n)  
        v.append( Vector2( cos(a), sin(a) ) )  
    return CrearArrayMesh( v )
```

Otra función auxiliar útil compone una transformación por la izquierda:

```
func TransformaNode2D( n : Node2D, tr : Transform2D ) -> Node2D :  
    n.transform = tr * n.transform  
    return n
```

Funciones para el pomo, hoja derecha y puerta

```
func Pomo() -> Node2D :  
    var tra = Transform2D().translated( Vector2( 0.1, 0.5 ) )  
    var esc = Transform2D().scaled( Vector2( 0.06, 0.06 ) )  
    return CrearMeshInstance2D( circunferencia, tra*esc )  
  
func HojaDer() -> Node2D :  
    var tra = Transform2D().translated( Vector2( 0.5, 0.0 ) )  
    var esc = Transform2D().scaled( Vector2( 0.5, 1.0 ) )  
    var n = Node2D.new()  
    n.add_child( TransformaNode2D( Pomo(), tra ) )  
    n.add_child( CrearMeshInstance2D( cuadrado, tra*esc ) )  
    return n  
  
func Puerta() -> Node2D :  
    var tra = Transform2D().translated( Vector2( 1.0, 0.0 ) )  
    var esc = Transform2D().scaled( Vector2( -1.0, 1.0 ) )  
    var n = Node2D.new()  
    n.add_child( HojaDer() )  
    n.add_child( TransformaNode2D( HojaDer(), tra*esc ) )  
    return n
```

Marco izquierdo y marcos

```
func MarcoIzq() -> Node2D :  
    var tra = Transform2D().translated( Vector2( 0.0, 1.0 ) )  
    var esc = Transform2D().scaled( Vector2( 0.9, -0.8 ) )  
    var n = Node2D.new()  
    n.add_child( CrearMeshInstance2D( cuadrado, tr_identidad ) )  
    n.add_child( CrearMeshInstance2D( triangulo, tra*esc) )  
    return n  
  
func Marcos() -> Node2D :  
    var esc1 = Transform2D().scaled( Vector2( 0.8, 1.0 ) )  
    var tra = Transform2D().translated( Vector2( 2.4, 0.0 ) )  
    var esc2 = Transform2D().scaled( Vector2( -1.0, 1.0 ) )  
    var n = Node2D.new()  
    n.add_child( TransformaNode2D( MarcoIzq(), esc1 ) )  
    n.add_child( TransformaNode2D( MarcoIzq(), esc1*tra*esc2 ) )  
    return n
```

3. Ejemplo de un árbol 2D.

3.2. Implementación en Godot..

Ventana, instancia puerta e instancia ventana

```
func Ventana() -> Node2D :  
    var tra = Transform2D().translated( Vector2( 0.1, 0.1 ) )  
    var esc = Transform2D().scaled( Vector2( 0.4, 0.8 ) )  
    var n = Node2D.new()  
    n.add_child( CrearMeshInstance2D( cuadrado, tr_identidad ) )  
    n.add_child( TransformaNode2D( Marcos(), tra*esc ) )  
    return n  
  
func InstPuerta() -> Node2D :  
    var tra = Transform2D().translated( Vector2( 0.56, 0.0 ) )  
    var esc = Transform2D().scaled( Vector2( 0.3, 0.43 ) )  
    var n = Node2D.new()  
    n.add_child( TransformaNode2D( Puerta(), tra*esc ) )  
    return n  
  
func InstVentana() -> Node2D :  
    var esc = Transform2D().scaled( Vector2( 0.3, 0.3 ) )  
    var n = Node2D.new()  
    n.add_child( TransformaNode2D( Ventana(), esc ) )  
    return n
```

Fachada

```
func Fachada() -> Node2D :  
  
    var tra1 = Transform2D().translated( Vector2( 0.13, 0.13 ) )  
    var tra2 = Transform2D().translated( Vector2( 0.00, 0.43 ) )  
    var tra3 = Transform2D().translated( Vector2( 0.43, 0.00 ) )  
  
    var n = Node2D.new()  
    n.add_child( CrearMeshInstance2D( casa, tr_identidad ) )  
    n.add_child( TransformaNode2D( InstPuerta(), tr_identidad ) )  
    n.add_child( TransformaNode2D( InstVentana(), tra1 ) )  
    n.add_child( TransformaNode2D( InstVentana(), tra1*tra2 ) )  
    n.add_child( TransformaNode2D( InstVentana(), tra1*tra2*tra3 ) )  
  
    return n
```

Subsección 3.3.

Implementación de Grafos parametrizados (y animados).

Diseño de grafos parametrizados

A veces los grafos de escena pueden depender de uno o varios **parámetros o grados de libertad**.

Son uno o varios valores reales (o vectores), de forma que cada uno de ellos **determina una o varias transformaciones de un modelo jerárquico**. Esto permite:

Animar de forma sencilla un modelo jerárquico: haciendo depender los valores de los parámetros del tiempo real transcurrido durante la ejecución.

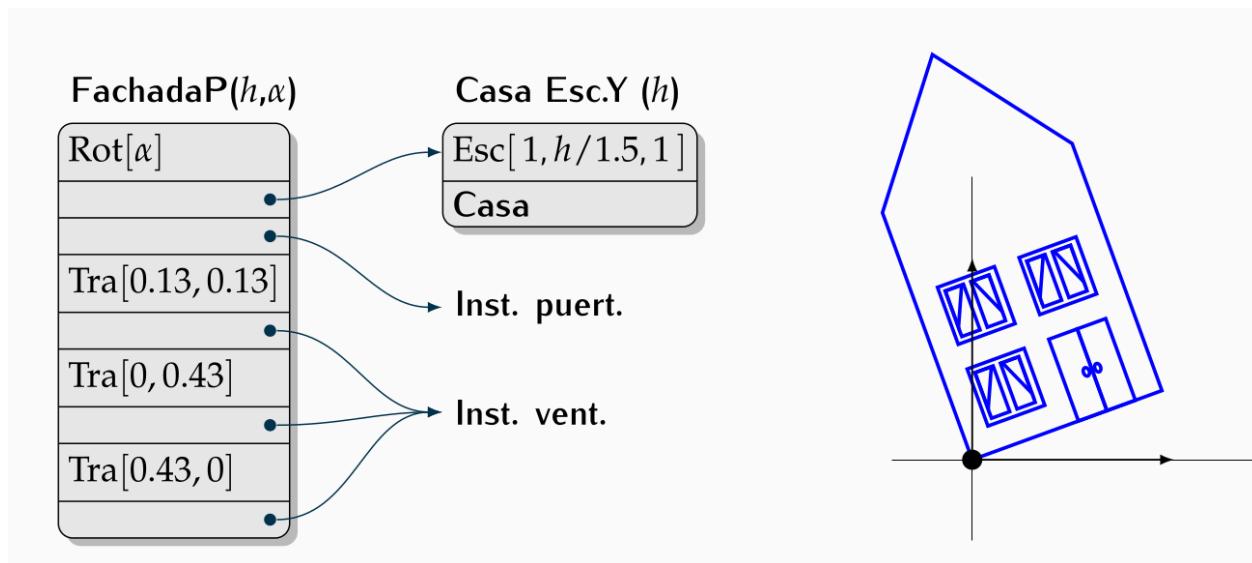
Generar múltiples variantes de un diseño: ya que podemos construir nodos o sub-árboles similares, que solo difieren en los valores de los parámetros.

Modificar interactivamente un modelo en tiempo de ejecución: ya que se puede permitir al usuario usar controles o eventos de entrada para modificar los parámetros.

A modo de ejemplo, en un modelo de un coche, puede existir un parámetro que sea un ángulo de giro de las cuatro ruedas. Es un real en radianes que determina la transformación de rotación.

Ejemplo de diseño de un grafo parametrizado

En el ejemplo anterior, podemos hacer que el grafo dependa de dos parámetros, α (rotación de la figura completa, en radianes) y h (altura de la fachada, real positivo):



Variación lineal u oscilante

Para animar un modelo jerárquico, los valores de los parámetros se pueden variar en cada frame y recalculando las transformaciones de los nodos. Así se consigue animación o interacción. Hay muchas opciones, dos bastante sencillas y útiles son:

- Hacer que un parámetro **varíe linealmente con el tiempo**, por ejemplo, un ángulo de rotación r que aumenta a ritmo constante por cada segundo:

$$r = r_0 + 2\pi \cdot w \cdot t,$$

donde t es el tiempo transcurrido de animación en segundos, w es el número de vueltas por segundo, y r_0 el valor de r al inicio (cuando $t = 0$).

- Para escalados o desplazamientos queremos evitar que crezcan indefinidamente (no suele tener sentido), así que hacemos que v **oscile entre otros dos valores a y b** , y lo haga w veces por segundo:

$$v = a + (b - a) \cdot \frac{1 + \sin(2\pi \cdot w \cdot t)}{2}$$

Implementación de grafos parametrizados en Godot

En este ejemplo modificamos las transformaciones de dos nodos **n1** y **n2**

```
var a      := .... ## valor mínimo (e inicial) del parámetro 2 (oscilante)
var b      := .... ## valor máximo del parámetro 2 (oscilante)
var w1     := .... ## ciclos o vueltas por segundo (parámetro 1)
var w2     := .... ## ciclos o vueltas por segundo (parámetro 2)
var ang2   := 0.0  ## valor acumulado de 2*PI*w2*t (argumento de 'sin')

func _process( delta : float ) :
    if animacion_activada :

        ## actualizar transformación del nodo 'n1':
        n1.rotate( 2*PI*w1*delta ) ## simplemente acumulamos rotación

        ## actualizar transformación del nodo 'n2':
        ## (se calcula un factor de escala 'fe' oscilante)
        ang2 += 2.0 * PI * w2 * delta ## acumulamos en 'ang2'
        var fe : float = a + (b-a)*(1.0 + sin( ang2 ))/2.0
        n2.transform = Transform2D().scaled( Vector2( fe, 1.0 ) )
```

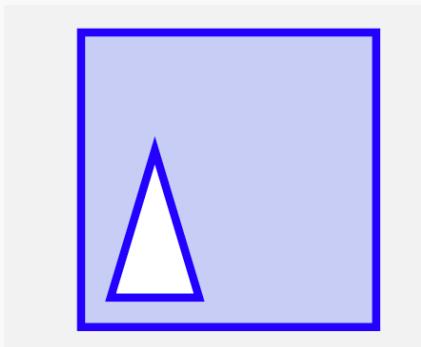
Sección 4.

Problemas

Escena simple

Problema 5.1:

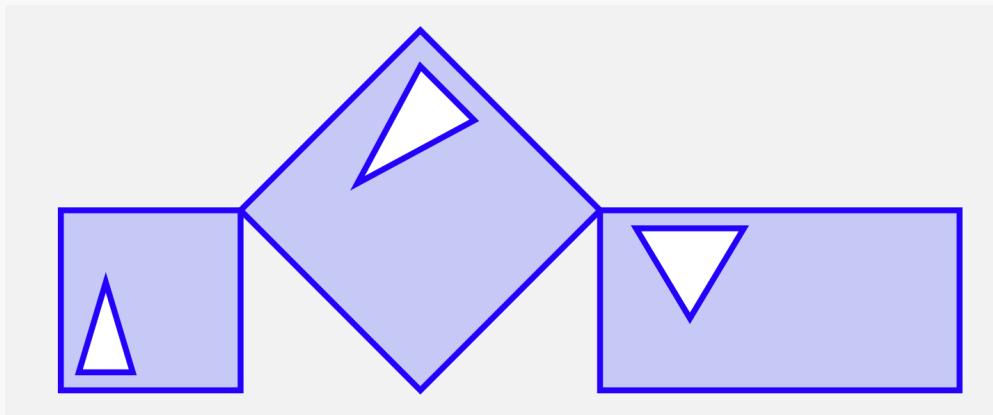
Implementa un proyecto cuya escena principal tenga un de tipo **Node2D** con varios nodos hijos, que formen la figura con un cuadrado de lado 2, centrado en el origen, y con un triángulo inscrito. El cuadrado debe estar relleno de azul claro, el triángulo de blanco, y las aristas deben verse de color azul oscuro.



Proyecto con dos escenas.

Problema 5.2:

Crea un proyecto Godot con una escena principal con un nodo raíz compuesto. Ese nodo tendrá tres hijos, cada uno es una instancia de la escena del problema anterior, pero con una transformación distinta.



Escena simple

Problema 5.3:

Implementa un proyecto Godot con una función **Tronco** que crea y devuelve un **Node2D** con dos nodos hijos que forman la figura de aquí abajo (uno para el relleno y otro para las aristas).

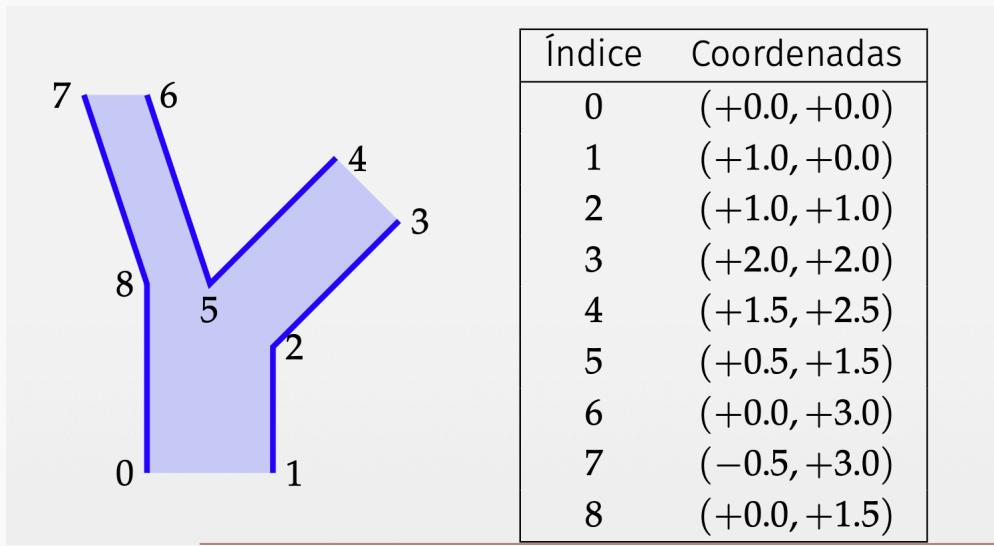
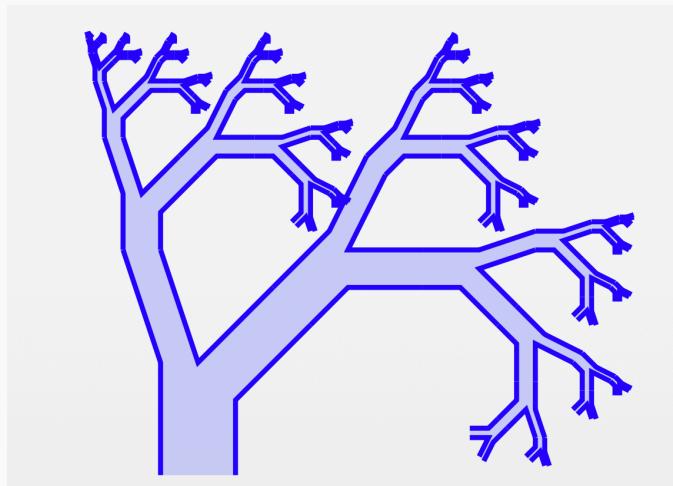


Figura recursiva

Problema 5.4:

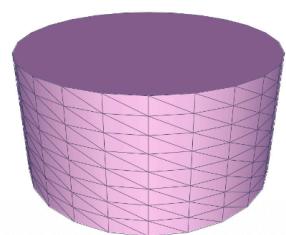
Implementa otro proyecto Godot que use la función del problema anterior para otra función, **Arbol(n)** que genera un árbol de escena con la figura de aquí abajo, que incluye múltiples instancias de **Tronco**, situadas recursivamente unas adyacentes a otras, hasta un nivel de recursividad dado por **n**.



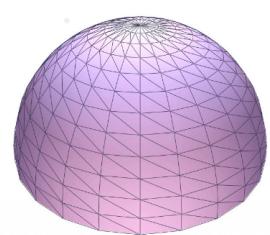
Árbol de escena 3D

Problema 5.5:

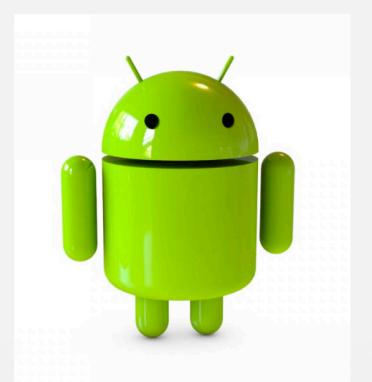
En un proyecto Godot 3D (puedes usar la práctica 2) para crear una figura como el logo de Android, usando únicamente dos objetos **ArrayMesh**, uno con un cilindro y otro con una semiesfera.



Cilindro



Semiesfera



Android

Fin de transparencias.

Informática Gráfica.

Sesión 3: Espacios y transformaciones afines.

Carlos Ureña, Sept 2025.

Dept. Lenguajes y Sistemas Informáticos.

Universidad de Granada.

Índice

Espacios afines y marcos de referencia	3
Transformaciones afines y matrices de transformación.	38
Transformaciones usuales en Informática Gráfica.	57
Transformaciones en Godot.	88

Sección 1.

Espacios afines y marcos de referencia

1. Las estructuras de espacio afín y espacio vectorial.
2. Marcos afines y coordenadas homogéneas
3. Producto escalar y vectorial de vectores.

Subsección 1.1.

Las estructuras de espacio afín y espacio vectorial.

Espacios afines

La noción abstracta de **Espacio Afín** es esencial en Informática Gráfica, ya que permite:

- Razonar sobre los puntos en el espacio,
- Diseñar e implementar representaciones y de esos puntos y estructuras de datos relacionadas en la memoria de los ordenadores, y
- Diseñar e implementar algoritmos que operan sobre esas representaciones y estructuras de datos.

Un espacio afín esencialmente es un modelo de los puntos de un espacio geométrico (de 1,2,3 o más dimensiones) y de las operaciones que podemos hacer con ellos. Está muy relacionado con el concepto de **Espacio Vectorial**

1. Espacios afines y marcos de referencia.

1.1. Las estructuras de espacio afín y espacio vectorial..

Estructura de Espacio Vectorial

Un **espacio vectorial** V_n (con n entero, > 0) es un conjunto de elementos llamados **vectores**, o **vectores libres**, y que notaremos con una flecha: $\vec{u}, \vec{v}, \vec{w}$, etc ..., los cuales interpretamos informalmente como flechas en el espacio, representando un **desplazamiento** desde el origen hasta el destino. El origen es indiferente, lo importante es la distancia y la dirección hasta el destino (por eso se llaman *vectores libres*). En V_n hay definidas estas **dos operaciones**:

Suma de vectores:

Para cualesquiera $\vec{u}, \vec{v} \in V_n$, existe un único $\vec{w} \in V_n$ tal que $\vec{w} = \vec{u} + \vec{v}$. El vector \vec{w} representa un desplazamiento equivalente a aplicar primero el desplazamiento \vec{u} y luego el desplazamiento \vec{v} (o al revés).

Producto por real:

Para cualquier $\vec{u} \in V_n$ y cualquier $a \in \mathbb{R}$, existe un $\vec{v} \in V_n$ tal que $\vec{v} = a\vec{u}$. El vector \vec{v} representa un desplazamiento en la misma dirección que \vec{u} , pero con una distancia multiplicada por a (si a es negativo, el sentido es el opuesto). Los vectores \vec{u} y $\vec{v} = a\vec{u}$ son **vectores paralelos**.

1. Espacios afines y marcos de referencia.

1.1. Las estructuras de espacio afín y espacio vectorial..

Axiomas del espacio vectorial

El espacio vectorial V_n cumple las siguientes propiedades, para cualquier $\vec{u}, \vec{v}, \vec{w} \in V_n$ y $a, b \in \mathbb{R}$:

1. Asociatividad de la suma: $(\vec{u} + \vec{v}) + \vec{w} = \vec{u} + (\vec{v} + \vec{w})$
2. Comutatividad de la suma: $\vec{u} + \vec{v} = \vec{v} + \vec{u}$
3. Elemento identidad de la suma: existe un elemento $\vec{0} \in V_n$ (llamado *vector nulo*) tal que $\vec{u} + \vec{0} = \vec{u}$.
4. Elemento opuesto de la suma: para cada $\vec{u} \in V_n$, existe su *vector opuesto* $-\vec{u} \in V_n$ tal que $\vec{u} + (-\vec{u}) = 0$
5. Elemento identidad del producto: $1\vec{u} = \vec{u}$
6. Asociatividad del producto: $a(b\vec{u}) = (ab)\vec{u}$
7. Distributividad de la suma de vectores y producto: $a(\vec{u} + \vec{v}) = a\vec{u} + a\vec{v}$
8. Distributividad de la suma de reales y producto: $(a + b)\vec{u} = a\vec{u} + b\vec{u}$

(identificar vectores con desplazamientos permite entender estos axiomas).

La estructura de *Espacio Afín*

Un **espacio afín** A_n (con n entero, > 0) sobre un espacio vectorial V_n es un conjunto de elementos llamados **puntos**, que notaremos con un punto sobre ellos $\dot{p}, \dot{q}, \dot{r}$, etc ..., los cuales interpretamos informalmente como posiciones en un espacio de n dimensiones. En A_n definimos esta operación:

Suma punto y vector:

Para cualquier punto $\dot{p} \in A_n$ y cualquier vector $\vec{v} \in V_n$, existe un punto $\dot{q} \in A_n$ tal que $\dot{q} = \dot{p} + \vec{v}$.

Intuitivamente, el punto \dot{q} representa la posición a la que se llega al desplazarse desde la posición \dot{p} siguiendo el desplazamiento representado por el vector \vec{v} .

Axiomas del espacio afín

El espacio afín A_n cumple las siguientes propiedades, para cualquier $\dot{p}, \dot{q}, \dot{r} \in A_n$ y cualquier $\vec{u}, \vec{v} \in V_n$:

1. Identidad de la suma: $\dot{p} + \vec{0} = \dot{p}$
2. Asociatividad: $(\dot{p} + \vec{v}) + \vec{w} = \dot{p} + (\vec{v} + \vec{w})$
3. Dados dos puntos \dot{p} y \dot{q} cualesquiera, existirá un único vector $\vec{v} \in V_n$ tal que $\dot{q} = \dot{p} + \vec{v}$.

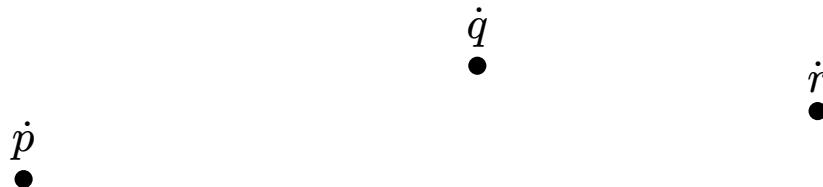
Los axiomas implican que podemos definir la operación:

Resta de puntos: para cualquiera dos puntos \dot{p} y \dot{q} se define el vector $\vec{v} = \dot{q} - \dot{p}$ como el único vector tal que $\dot{q} = \dot{p} + \vec{v}$.

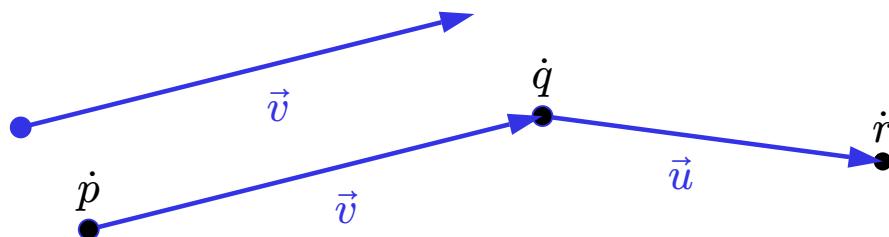
Lógicamente, se cumple $\dot{p} - \dot{p} = \vec{0}$.

Ejemplos de puntos y vectores.

En este ejemplo vemos los puntos \dot{p} , \dot{q} y \dot{r} en el espacio afín 2D de la pantalla:

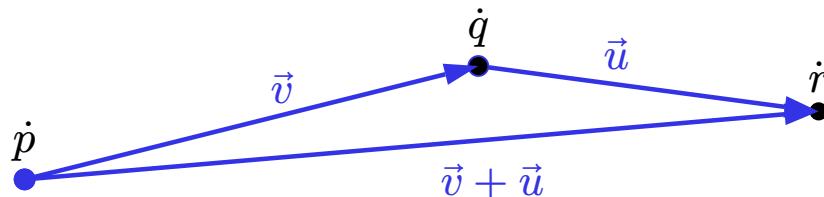


El vector libre $\vec{v} = \dot{q} - \dot{p}$, representa el desplazamiento desde \dot{p} hasta \dot{q} . Como representa un desplazamiento se puede situar con su extremo en cualquier sitio (lo dibujamos dos veces). Igualmente, el vector $\vec{u} = \dot{r} - \dot{q}$ es el desplazamiento desde \dot{q} hasta \dot{r} .

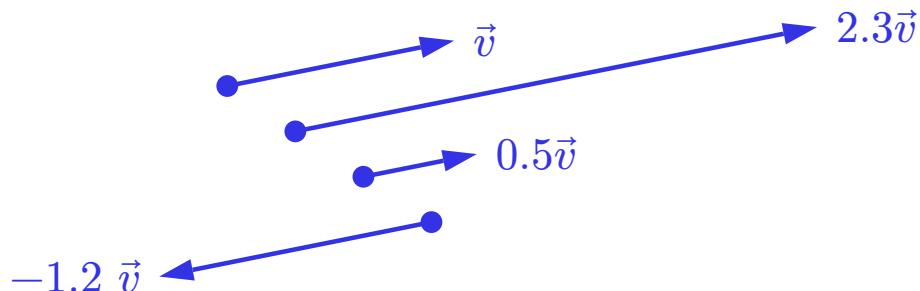


Suma de vectores y multiplicación por escalar

Si \vec{v} traslada de \dot{p} a \dot{q} , y \vec{u} traslada de \dot{q} a \dot{r} , entonces el vector $\vec{v} + \vec{u}$ traslada directamente de \dot{p} a \dot{r} :



Dado un vector \vec{v} y un real a , el vector $a\vec{v}$ representa un desplazamiento en la misma dirección que \vec{v} , pero con una distancia multiplicada por a (si a es negativo, el sentido es el opuesto):



Rectas en el espacio afín.

Una **recta** que pasa por dos puntos distintos \dot{q} y \dot{r} se define como el conjunto de puntos \dot{p} tales que existe un $t \in \mathbb{R}$ tal que:

$$\dot{p} = \dot{q} + t(\dot{r} - \dot{q})$$

- A esta expresión se le denomina **ecuación paramétrica** de la recta, y el punto \dot{p} puede escribirse como $\dot{p}(t)$ para indicar su dependencia del parámetro t .
- Al vector $\vec{v} = \dot{q} - \dot{p}$ se le denomina **vector director** de la recta (se cumple $\|\vec{v}\| > 0$)
- La ecuación paramétrica permite identificar puntos en un recta con valores reales.
- El valor t puede interpretarse (informalmente) como la *distancia* entre \dot{r} y \dot{p} , medida en unidades de la longitud del vector $\dot{q} - \dot{p}$.
- Dos rectas **son paralelas** cuando sus vectores directores \vec{u} y \vec{v} son paralelos, es decir, si existe un real a tal que $\vec{u} = a\vec{v}$.

1. Espacios afines y marcos de referencia.

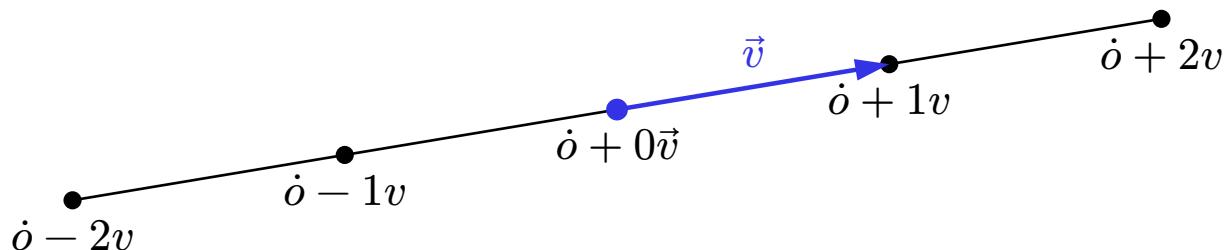
1.1. Las estructuras de espacio afín y espacio vectorial..

Ejemplo de recta en el espacio afín.

La ecuación paramétrica de una recta que pasa por \dot{o} y tiene como vector director \vec{v} es:

$$\dot{p}(t) = \dot{o} + t\vec{v}$$

Se pueden obtener todos los puntos de la recta dando valores a t , aquí vemos algunos puntos en posiciones correspondientes a t entero:



Si los valores de t están equiespaciados, los puntos $\dot{p}(t)$ también lo estarán.

1. Espacios afines y marcos de referencia.

1.1. Las estructuras de espacio afín y espacio vectorial..

Combinaciones afines de puntos

Los puntos **no pueden ser multiplicados por reales**, pero se puede definir la **combinación afín** $a\dot{p} + b\dot{q}$ de dos puntos \dot{p} y \dot{q} (con $a, b \in \mathbb{R}$ y $a + b$ vale 1 o 0):

- Cuando $a + b = 1$ la combinación afín se define como un **punto en la recta** que pasa por \dot{p} y \dot{q} , así:

$$a\dot{p} + b\dot{q} \equiv \dot{p} + b(\dot{q} - \dot{p})$$

- Cuando $a + b = 0$ la combinación se define como un **vector paralelo a la recta** que pasa por \dot{p} y \dot{q} , así:

$$a\dot{p} + b\dot{q} \equiv b(\dot{q} - \dot{p})$$

Puesto que \dot{p} y \dot{q} pueden ser el mismo punto, esto permite definir la **multiplicación de un punto \dot{p} por 1 o por 0**:

$$1\dot{p} \equiv 1\dot{p} + 0\dot{p} = \dot{p} + 0(\dot{p} - \dot{p}) = \dot{p}$$

$$0\dot{p} \equiv 0\dot{p} + 0\dot{p} = 0(\dot{p} - \dot{p}) = \vec{0}$$

Subsección 1.2.

Marcos afines y coordenadas homogéneas

1. Espacios afines y marcos de referencia.

1.2. Marcos afines y coordenadas homogéneas.

Bases de un espacio vectorial

Una **base** de un espacio vectorial V_n es un conjunto ordenado de n vectores $\{\vec{b}_0, \vec{b}_1, \dots, \vec{b}_{n-1}\}$, con $\vec{b}_i \in V_n$, que cumplen estas dos propiedades:

- Cualquier vector $\vec{v} \in V_n$ puede expresarse como *combinación lineal* de todos los vectores de la base, es decir, existe un única tupla $(a_0, a_1, \dots, a_{n-1}) \in \mathbb{R}^n$ tal que:

$$\vec{v} = a_0 \vec{b}_0 + a_1 \vec{b}_1 + \dots + a_{n-1} \vec{b}_{n-1}.$$

- Ningún vector de la base puede expresarse como combinación lineal del resto (**son linealmente independientes**).

Cualquier conjunto de n vectores linealmente independientes en V_n es una base del espacio. Se dice que n es la **dimensión** de V_n .

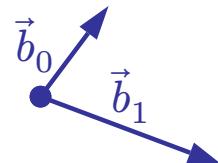
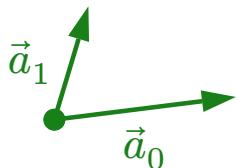
La elección de una base permite representar cualquier vector $\vec{v} \in V_n$ mediante sus **coordenadas** (a_1, a_2, \dots, a_n) , que siempre serán **relativas a dicha base**.

1. Espacios afines y marcos de referencia.

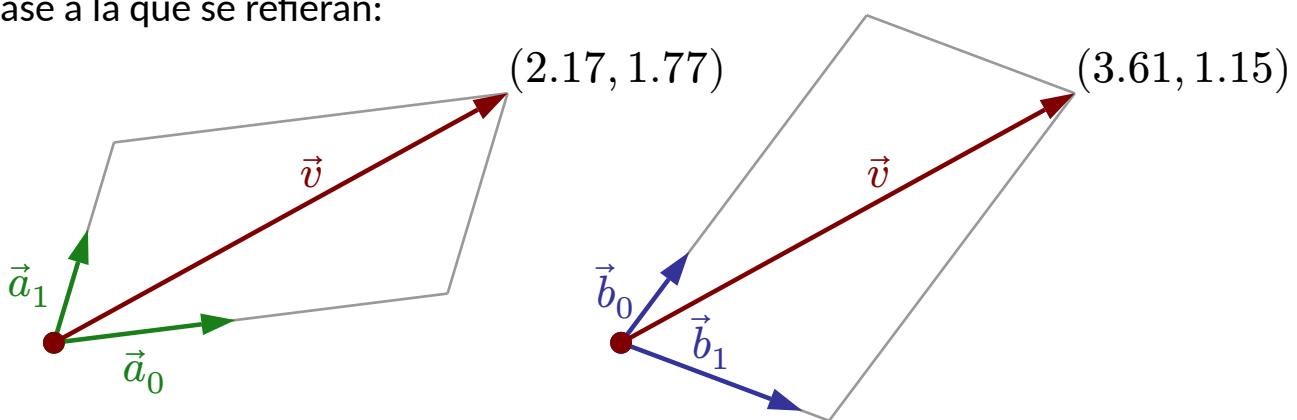
1.2. Marcos afines y coordenadas homogéneas.

Ejemplos de bases y coordenadas en un espacio vectorial.

En 2D, vemos dos bases $A \equiv \{\vec{a}_0, \vec{a}_1\}$ (en verde) y $B \equiv \{\vec{b}_0, \vec{b}_1\}$ (en azul):



Un mismo vector \vec{v} (en rojo) tendrá dos pares de coordenadas distintas, según la base a la que se refieran:



1. Espacios afines y marcos de referencia.

1.2. Marcos afines y coordenadas homogéneas.

Marcos afines

Un **marco afín** (o *marco de referencia*, o *marco de coordenadas*, o *sistema de coordenadas*) \mathcal{R} en un espacio afín A_n es una tupla compuesta por los n vectores de una **base** de V_n y un punto \dot{o} del A_n (llamado **origen** del marco):

$$\mathcal{R} = (\vec{e}_0, \vec{e}_1, \dots, \vec{e}_{n-1}, \dot{o})$$

Fijado el marco afín \mathcal{R} , cualquier punto $\dot{q} \in A_n$ puede expresarse como:

$$\dot{q} = a_0 \vec{e}_0 + a_1 \vec{e}_1 + \dots + a_{n-1} \vec{e}_{n-1} + 1\dot{o}$$

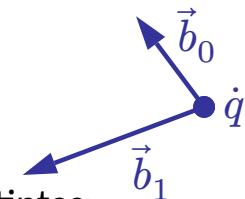
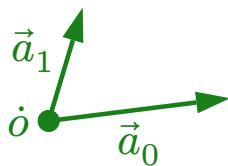
e igualmente cualquier vector \vec{v} puede expresarse como:

$$\vec{v} = a_0 \vec{e}_0 + a_1 \vec{e}_1 + \dots + a_{n-1} \vec{e}_{n-1} + 0\dot{o}$$

donde a_0, a_1, \dots, a_n son n valores reales **únicos**, denominados las **coordenadas** del punto \dot{q} o del vector \vec{v} en el marco \mathcal{R} .

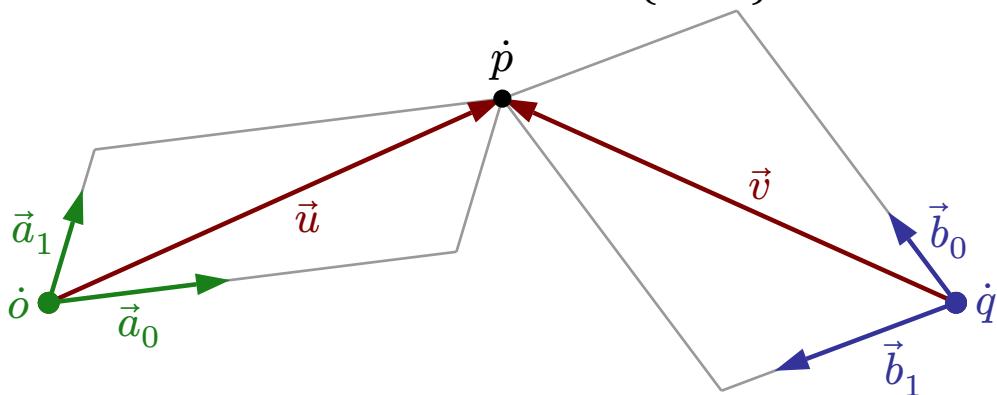
Ejemplos de marcos afines.

Vemos dos marcos afines $\mathcal{R} \equiv \{\vec{a}_0, \vec{a}_1, \dot{o}\}$ (en verde) y $B \equiv \{\vec{b}_0, \vec{b}_1, \dot{q}\}$ (en azul):



Un mismo punto \dot{p} tendrá dos pares de coordenadas distintas:

- (2.25, 1.35) del vector $\vec{u} = \dot{p} - \dot{o}$ (relativas a $\{\vec{a}_0, \vec{a}_1\}$)
- (3.22, 1.29) del vector $\vec{v} = \dot{p} - \dot{q}$ (relativas a $\{\vec{b}_0, \vec{b}_1\}$)



1. Espacios afines y marcos de referencia.

1.2. Marcos afines y coordenadas homogéneas.

Coordenadas homogéneas y su espacio afín.

Las **coordenadas homogéneas** de un punto o un vector (relativos a un marco \mathcal{R}) forman un **vector columna** con $n + 1$ valores reales:

- Los n primeros valores son las coordenadas del punto o vector relativas a \mathcal{R}
- El último valor es 1 para puntos y 0 para vectores, se suele escribir w .

El conjunto de todas las posibles tuplas tiene una estructura de espacio afín:

- El subconjunto de las tuplas con $w = 1$ es un **espacio afín**, que llamamos A_n ya que dos de estas tuplas pueden restarse y se obtiene otra tupla con $w = 0$ (ya fuera de este subconjunto).
- El subconjunto de las tuplas con $w = 0$ es un **espacio vectorial**, y en concreto es el espacio vectorial asociado al espacio afín anterior, ya que estas tuplas pueden sumarse entre ellas y multiplicarse por un real.

Fijado un marco afín \mathcal{R} de A_n , hay una correspondencia biunívoca entre A_n y A_n

1. Espacios afines y marcos de referencia.

1.2. Marcos afines y coordenadas homogéneas.

Correspondencia entre tuplas y puntos y vectores

Decimos que los puntos y vectores se obtienen **interpretando** sus coordenadas homogéneas en un marco $\mathcal{R} = (\vec{e}_0, \dots, \vec{e}_{n-1}, \dot{o})$, en el sentido siguiente:

- Si $\mathbf{u} = (a_0, \dots, a_{n-1}, 1)^T$ son las coordenadas de \dot{p} en \mathcal{R} , entonces podemos escribir la igualdad $\dot{p} = \mathcal{R}\mathbf{u}$, ya que

$$\dot{p} = 1\dot{o} + \sum_0^{n-1} a_i \vec{e}_i = (\vec{e}_0, \vec{e}_1, \dots, \vec{e}_{n-1}, \dot{o}) \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \\ 1 \end{pmatrix} = \mathcal{R}\mathbf{u}.$$

- Si $\mathbf{v} = (a_0, \dots, a_{n-1}, 0)^T$ son las coordenadas de \vec{v} en \mathcal{R} , entonces podemos escribir la igualdad $\vec{v} = \mathcal{R}\mathbf{v}$, ya que:

$$\vec{v} = 0\dot{o} + \sum_0^{n-1} a_i \vec{e}_i = (\vec{e}_0, \vec{e}_1, \dots, \vec{e}_{n-1}, \dot{o}) \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \\ 0 \end{pmatrix} = \mathcal{R}\mathbf{v}$$

1. Espacios afines y marcos de referencia.

1.2. Marcos afines y coordenadas homogéneas.

Operaciones con las tuplas de coordenadas

Un marco afín \mathcal{R} en un espacio afín A_n puede verse como una correspondencia 1 a 1 entre A_n hacia A_n , ya que permite, a partir de una tupla $\mathbf{c} \in A_n$, obtener su correspondiente punto o vector $\mathcal{R}\mathbf{c} \in A_n$ (y al revés).

- Esa aplicación es **lineal** en el sentido de que **conserva las operaciones de los espacios afines**, es decir para cualquiera tuplas, \mathbf{u}, \mathbf{v} (en V_n), y \mathbf{p} (en A_n), y real a se cumple:

$$\mathcal{R}(\mathbf{p} + \mathbf{u}) = \mathcal{R}\mathbf{p} + \mathcal{R}\mathbf{u}$$

$$\mathcal{R}(\mathbf{u} + \mathbf{v}) = \mathcal{R}\mathbf{u} + \mathcal{R}\mathbf{v}$$

$$\mathcal{R}(a\mathbf{u}) = a(\mathcal{R}\mathbf{u})$$

- Esto implica que podemos **usar las operaciones con las tuplas para hacer cálculos con puntos y vectores**, esto permite implementar algoritmos que operan en espacios afines o vectoriales.

1. Espacios afines y marcos de referencia.

1.2. Marcos afines y coordenadas homogéneas.

Ventajas de las coordenadas homogéneas

Las coordenadas homogéneas en A_n se pueden usar para realizar cálculos con puntos y vectores en un ordenador, ya que las usamos como representaciones de los puntos y vectores abstractos de A_n .

El uso de las coordenadas homogéneas **simplifica muchísimo los cálculos con puntos y vectores en Informática Gráfica:**

- Permite representar de forma similar tanto los puntos como los vectores.
- Permite implementar la *transformaciones afines* con matrices.
- Simplifica los cálculos de proyección perspectiva.

Es importante recordar siempre que unas coordenadas no tienen sentido de forma independiente del marco de coordenadas al cual son relativas.

1. Espacios afines y marcos de referencia.

1.2. Marcos afines y coordenadas homogéneas.

Marcos afines en IG

En Informática Gráfica se usan diversos marcos de coordenadas distintos:

- Marco de mundo: global a una escena.
- Marco de objeto: específico de un objeto concreto dentro de la escena.
- Marco de cámara: para coordenadas relativas a una cámara virtual posicionada y orientada de alguna forma concreta dentro de una escena.
- Marco del dispositivo: para coordenadas en una ventana o una imagen, en unidades de pixels.
- Marco normalizado de dispositivo: para coordenadas en una ventana o una imagen, en unidades normalizadas entre -1 y $+1$.

El resultado anterior nos indica que **podemos usar matrices para convertir las coordenadas relativas a un marco en las coordenadas relativas a otro marco**, lo cual es esencial en Informática Gráfica.

Subsección 1.3.

Producto escalar y vectorial de vectores.

La base especial W_n de un espacio afín A_n

En Informática Gráfica, si representamos objetos reales usando un espacio afín abstracto A_n , entonces necesitamos trasladar a A_n los conceptos de *distancia* y *perpendicularidad* que usamos en el espacio físico real. Para ello, designamos una base de V_n llamada la **base especial**

$$W_n = (\hat{e}_0, \hat{e}_1, \dots, \hat{e}_{n-1})$$

cuyos vectores tienen **longitud unidad** y son perpendiculares dos a dos **por definición**. Usamos $\hat{\cdot}$ en lugar de $\vec{\cdot}$ para estos vectores, y se les llama **versores**. En 2D escribiremos $W_2 = (\hat{x}, \hat{y})$, y en 3D $W_3 = (\hat{x}, \hat{y}, \hat{z})$. Una vez seleccionada W_n

- podemos definir la *distancia* en A_n , (ya que W_n define las unidades de distancia en todas las direcciones posibles),
- podemos decir si dos vectores son perpendiculares o no,
- definimos el *ángulo* entre dos vectores no nulos, y
- podemos decir cuando un marco afín es un *marco cartesiano* o no lo es.

Producto escalar de vectores

El **producto escalar (dot product)** es una función que se aplica dos vectores $\vec{u}, \vec{v} \in V_n$ y produce un valor real que se nota como $\vec{u} \cdot \vec{v}$.

El producto escalar cumple estas propiedades:

- Comutativa: $\vec{u} \cdot \vec{v} = \vec{v} \cdot \vec{u}$
- Distributiva respecto a la suma: $\vec{u} \cdot (\vec{v} + \vec{w}) = \vec{u} \cdot \vec{v} + \vec{u} \cdot \vec{w}$
- Asociativa respecto al producto por reales: $(a\vec{u}) \cdot \vec{v} = a(\vec{u} \cdot \vec{v})$
- Positivo: $\vec{u} \cdot \vec{u} \geq 0$

Muchas posibles definiciones distintas del producto escalar pueden cumplir estas propiedades, así que necesitamos también exigir que para dos vectores \vec{e}_i y \vec{e}_j de la base W_n se cumpla:

$$\hat{e}_i \cdot \hat{e}_j \equiv \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

Longitud de un vectores. Versores.

La **norma** (o **módulo**, o **longitud**) de un vector $\vec{u} \in V_n$ es el número real no negativo que se define como:

$$\|\vec{u}\| \equiv \sqrt{\vec{u} \cdot \vec{u}}$$

- La norma de un vector es siempre un valor real no negativo, y representa la longitud del desplazamiento que representa el vector.
- La **distancia** entre dos puntos \dot{p} y \dot{q} se define como el valor real $\|\dot{q} - \dot{p}\|$ (es decir, la longitud del vector que los une).
- Si sabemos que un vector tiene longitud 1, se le denomina **versor** (o **vector unitario**)., y se escribe con un gorro: $\hat{x}, \hat{y}, \hat{z}, \hat{e}, \hat{a}$ etc...
- Los vectores de la base W_n son unitarios, y por eso los hemos escrito con un gorro.

Perpendicularidad y ángulo entre vectores

Una vez bien definido el producto escalar, podemos definir la noción de **perpendicularidad** y el **ángulo** entre dos vectores \vec{u} y \vec{v} (ninguno nulo):

- Si se cumple que $\vec{u} \cdot \vec{v} = 0$, entonces se dice que los vectores \vec{u} y \vec{v} son **perpendiculares** (u **ortogonales**).
- El **ángulo** θ entre \vec{u} y \vec{v} es un valor real (en radianes), en el rango $[0, \pi]$, se define como:

$$\theta \equiv \arccos\left(\frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\| \|\vec{v}\|}\right),$$

y por tanto, para dos versores \hat{a} y \hat{b} con ángulo θ entre ellos se cumple:

$$\cos(\theta) = \hat{a} \cdot \hat{b}.$$

- Los versores de W_n son **perpendiculares** dos a dos por la definición de W_n .

Producto vectorial en 3D

Consideramos ahora el espacio vectorial de las tuplas de 4 componentes con $w = 0$ (son coordenadas de vectores en 3D). En ese espacio se puede definir la operación de **producto vectorial (cross product)** entre dos vectores \vec{u}, \vec{v} , que produce un otro vector $\vec{w} = \vec{u} \times \vec{v}$. Esta operación cumple estas propiedades:

- **Anticonmutativa** : $\vec{u} \times \vec{v} = -(\vec{v} \times \vec{u})$
- **Distributiva** respecto a la suma: $\vec{u} \times (\vec{v} + \vec{w}) = \vec{u} \times \vec{v} + \vec{u} \times \vec{w}$
- **Asociativa** respecto al producto por reales: $(a\vec{u}) \times \vec{v} = a(\vec{u} \times \vec{v})$
- **Perpendicularidad**: se cumple $\vec{u} \cdot (\vec{u} \times \vec{v}) = 0$, implica que $\vec{u} \times \vec{v}$ es perpendicular a \vec{u} y a \vec{v} .

Hay muchas posibles definiciones del producto vectorial que cumplen estas propiedades. Para definirlo correctamente se exige que si $W_3 = (\hat{x}, \hat{y}, \hat{z})$, entonces se cumpla:

$$\hat{x} \times \hat{y} = \hat{z} \quad \hat{y} \times \hat{z} = \hat{x} \quad \hat{z} \times \hat{x} = \hat{y}$$

Marcos ortogonales y ortonormales

Dado un marco $\mathcal{R} = (\vec{b}_0, \dots, \vec{b}_n, o)$, decimos que ese marco es:

Ortogonal: si los vectores de la base son perpendiculares dos a dos, es decir:

$$i \neq j \implies \vec{b}_i \cdot \vec{b}_j = 0$$

Ortonormal: si es ortogonal y además todos los vectores de la base son unitarios (son versores), es decir:

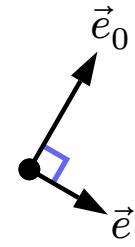
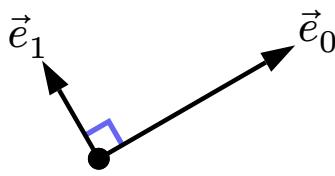
$$\vec{b}_i \cdot \vec{b}_i = 1$$

Definimos **la matriz de productos escalares** M asociada a \mathcal{R} como la matriz con $a_{ij} \equiv \vec{b}_i \cdot \hat{e}_j$, donde cada \hat{e}_i es un versor de W_n . Entonces se cumple:

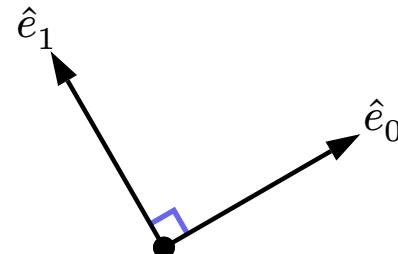
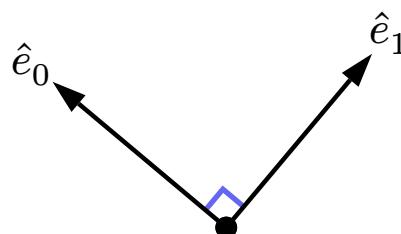
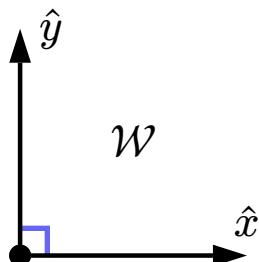
- Si \mathcal{R} es ortogonal, M también lo es.
- Si \mathcal{R} es ortonormal, M también lo es (tendrá determinante $+1$ o -1).

Ejemplos de marcos ortogonales y ortonormales en 2D

Estos marcos son ortogonales pero no ortonormales (los vectores de la base no son unitarios):



Estos marcos son ortonormales (los dos vectores de la base son unitarios y perpendiculares entre ellos). El marco \mathcal{W} (a la izquierda) es ortonormal por definición:



Orientación de un marco. Marcos cartesianos.

Un marco \mathcal{R} cualquiera puede tener orientación a *derechas* o a *izquierdas*, la orientación depende de la base de \mathcal{R} , en concreto del signo del determinante de la matriz M de productos escalares asociada a \mathcal{R}

- Si es positivo, será a **derechas** (W_n es a derechas **por definición**).
- Si es negativo, será a **izquierdas**.

Un marco \mathcal{R} es **cartesiano** si y solo si es ortonormal y además tiene orientación a derechas (es decir: el determinante de M es $+1$).

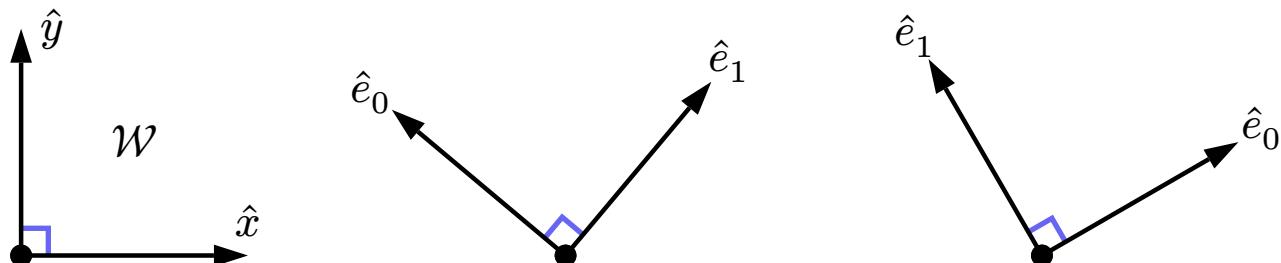
- Eso implica que los vectores en la base de \mathcal{R} se pueden hacer coincidir con los de W_n mediante una misma *rotación* aplicada a todos ellos.
- Con esta definición, un marco cuya base es W_n es **cartesiano** por definición, pero en un espacio afín hay otros muchos otros marcos también cartesianos.

Ejemplos de orientación de marcos en 2D

Dos marcos 2D con las dos orientaciones posibles (a izquierda y a derechas):



El marco \mathcal{W} es **a derechas** (por definición). El marco del centro es **a izquierdas** (\hat{e}_0 a la izquierda de \hat{e}_1) y el de la derecha es **a derechas** (\hat{e}_0 a la derecha de \hat{e}_1)



Cálculo de producto escalar con coordenadas cartesianas

En el espacio vectorial de las tuplas con $w = 0$, el producto escalar de dos tuplas de coordenadas (de vectores) $\mathbf{u} = (a_0, \dots, a_{n-1}, 0)^T$ y $\mathbf{v} = (b_0, \dots, b_{n-1}, 0)^T$ se escribe como $\mathbf{u} \cdot \mathbf{v}$ y es igual a la suma del producto componente a componente:

$$\mathbf{u} \cdot \mathbf{v} = \sum_0^{n-1} a_i b_i$$

Si \mathcal{C} es un marco cartesiano, y \mathbf{u} y \mathbf{v} las coordenadas en \mathcal{C} de dos vectores \vec{u} y \vec{v} , entonces el producto escalar de los vectores se puede calcular fácilmente usando el producto escalar de sus coordenadas, es decir, se cumple:

$$\vec{u} \cdot \vec{v} = (\mathcal{C}\mathbf{u}) \cdot (\mathcal{C}\mathbf{v}) = \mathbf{u} \cdot \mathbf{v}$$

Esto se cumple para cualquier marco cartesiano, es decir, el producto escalar de coordenadas es invariante entre marcos cartesianos.

Cálculo de producto vectorial con coordenadas cartesianas

En el espacio vectorial de las tuplas con $w = 0$ y $n = 3$ (coordenadas de vectores en 3D), el producto vectorial de dos tuplas de coordenadas $\mathbf{u} = (x_0, y_0, z_0, 0)$ y $\mathbf{v} = (x_1, y_1, z_1, 0)$ se escribe como $\mathbf{u} \times \mathbf{v}$ y se define como:

$$\mathbf{u} \times \mathbf{v} = \begin{pmatrix} y_0 z_1 - z_0 y_1 \\ z_0 x_1 - x_0 z_1 \\ x_0 y_1 - y_0 x_1 \end{pmatrix}$$

Si \mathcal{C} es un marco cartesiano, y \mathbf{u} y \mathbf{v} las coordenadas en \mathcal{C} de dos vectores \vec{u} y \vec{v} , entonces el producto vectorial de los vectores se puede calcular fácilmente usando el producto escalar de sus coordenadas, es decir, se cumple:

$$\vec{u} \times \vec{v} = (\mathcal{C}\mathbf{u}) \times (\mathcal{C}\mathbf{v}) = \mathcal{C}(\mathbf{u} \times \mathbf{v})$$

Esto se cumple para cualquier marco cartesiano, es decir, **el producto vectorial de coordenadas es invariante entre marcos cartesianos.**

1. Espacios afines y marcos de referencia.
- 1.3. Producto escalar y vectorial de vectores..

Problemas: cálculo del prod. escalar y vectorial.

Problema 3.1:

Demuestra que efectivamente el producto escalar de dos vectores se puede calcular (usando sus coordenadas en cualquier marco cartesiano) como la suma del producto componente a componente. Usa las propiedades que definen dicho producto escalar.

Problema 3.2:

Demuestra que el producto vectorial de dos vectores se puede calcular usando sus coordenadas en cualquier marco cartesiano según se ha indicado.

Problema 3.3:

Demuestra que el producto vectorial de dos vectores es perpendicular a cada uno de esos dos vectores.

Sección 2.

Transformaciones afines y matrices de transformación.

1. Transformaciones afines
2. Transformación afín de coordenadas. Matrices.
3. Tipos de transformaciones.

Transformaciones geométricas

Una **transformación geométrica** T es una aplicación desde un espacio afín A_n en otro B_n con la misma dimensión, donde A_n y B_n pueden ser distintos o **el mismo espacio**.

La transformación T se aplica a los puntos de A_n y produce puntos en B_n . Si \dot{q} es la imagen por T de \dot{p} , escribimos:

$$\dot{q} = T(\dot{p})$$

En general, una transformación geométrica

- aplicada a una recta puede producir otra recta, o bien una curva,
- puede ser continua o no serlo,
- puede no tener inversa si mas de un punto de A_n tiene como imagen un mismo punto en B_n .

Como consecuencia de todo esto, una transformación geométrica en general **no puede aplicarse a los vectores libres**, únicamente a los puntos.

Función asociada a una transformación geométrica

Si fijamos un marco \mathcal{R} del espacio A_n y una transformación geométrica T , entonces existirá una función F que asocia cada tupla \mathbf{c} de coordenadas de un punto \dot{p} con la correspondiente tupla $\mathbf{c}' = F(\mathbf{c})$ de coordenadas del punto transformado por T , es decir, se cumple:

$$T(\dot{p}) = T(\mathcal{R}\mathbf{c}) = \mathcal{R}F(\mathbf{c}) = \mathcal{R}\mathbf{c}'$$

- La llamamos la **funciones asociadas a la transformación T en el marco \mathcal{R}** .
- Esa función es en realidad una transformación geométrica en el espacio de las tuplas de coordenadas de puntos, es decir, en A_n .
- La función F se puede usar para implementar la transformación en un programa.
- Por supuesto, esta función depende del marco \mathcal{R} que hemos seleccionado, para otro marco sería distinta.

Subsección 2.1.

Transformaciones afines

Transformaciones afines

Una transformación geométrica T es una **transformación geométrica afín** si T es continua y transforma cada paralelogramo en otro paralelogramo. Un paralelogramo está definido por 4 puntos, con lados iguales dos a dos. Formalmente:

- Para cualquiera cuatro puntos $\dot{p}, \dot{q}, \dot{r}, \dot{s}$, si forman un paralelogramo, entonces sus imágenes forman otro, es decir:

$$\dot{q} - \dot{p} = \dot{s} - \dot{r} \implies T(\dot{q}) - T(\dot{p}) = T(\dot{s}) - T(\dot{r})$$

- Esa propiedad permite **aplicar T a vectores y producir vectores**. Si $\vec{v} = \dot{q} - \dot{p}$, entonces se cumple que:

$$T(\vec{v}) = T(\dot{q} - \dot{p}) \equiv T(\dot{q}) - T(\dot{p})$$

el vector resultado está bien definido pues no importa que para \dot{p} y \dot{q} seleccionemos: siempre que $\vec{v} = \dot{q} - \dot{p}$, se obtendrá el mismo vector $T(\vec{v})$.

Linealidad de las transformaciones afines

Se puede demostrar fácilmente que si T es afín, entonces para cualquiera dos vectores \vec{u} y \vec{v} se cumplirá que:

$$T(\vec{u} + \vec{v}) = T(\vec{u}) + T(\vec{v}).$$

También sabemos que T es **lineal**: para un punto \dot{p} , un vector \vec{v} , y un real a :

$$T(\dot{p} + a\vec{v}) = T(\dot{p}) + aT(\vec{v}).$$

Esto implica que T :

- **transforma líneas rectas en líneas rectas**, conservando la ecuación paramétrica de dichas rectas, y por tanto,
- **conserva las proporciones** entre las longitudes de vectores paralelos, y
- **conserva el paralelismo**: transforma dos líneas paralelas en otras dos líneas paralelas.

Transformaciones afines singulares y no singulares

Podemos distinguir las siguientes dos clases de transformaciones afines, en función de si son una biyección o no:

- Una transformación afín T será **no singular** si siempre aplica puntos distintos en puntos distintos, es decir, se cumple:

$$\dot{q} - \dot{p} \neq \vec{0} \quad \Rightarrow \quad T(\dot{q}) \neq T(\dot{p})$$

es lo mismo que decir que la imagen de cualquier vector no nulo es otro vector no nulo. Una transformación no singular es biyectiva y **tiene inversa**, T^{-1} .

- Una transformación afín será **singular** si hay al menos dos puntos con la misma imagen, o equivalentemente, hay al menos un vector no nulo que se transforma en el vector nulo. Una transformación singular **no tiene inversa**.
- Un ejemplo de transformación afín singular es una que pone a cero una componente de los vectores (por ejemplo la componente X): anula cualquier vector paralelo a ese eje.

Subsección 2.2.

Transformación afín de coordenadas. Matrices.

La matriz asociada a una transformación

Fijado un marco $\mathcal{R} = (\vec{b}_0, \dots, \vec{b}_{n-1}, \dot{o})$ y una transformación T (con función F en \mathcal{R}) consideramos los vectores de la base y el origen pero transformados por T , es decir los vectores $T(\vec{b}_j)$ y el punto $T(\dot{o})$. Sus coordenadas en \mathcal{R} son \mathbf{c}_j y \mathbf{p} , respectivamente, es decir:

$$T(\vec{b}_j) = \mathcal{R}\mathbf{c}_j \quad \text{y} \quad T(\dot{o}) = \mathcal{R}\mathbf{p} \quad \text{donde: } \begin{cases} \mathbf{c}_j = (c_{0,j}, \dots, c_{n-1,j}, 0)^T \\ \mathbf{p} = (p_0, \dots, p_{n-1}, 1)^T \end{cases}$$

Entonces podemos construir la **matriz asociada** a T (en coordenadas de \mathcal{R}), con $n + 1$ filas y columnas, y que tiene las tuplas \mathbf{c}_j y \mathbf{p} dispuestas **por columnas**, y que escribimos como M_T :

$$M_T \equiv \begin{pmatrix} c_{0,0} & \dots & c_{0,n-1} & p_0 \\ \vdots & & \vdots & \vdots \\ c_{n-1,0} & \dots & c_{n-1,n-1} & p_{n-1} \\ 0 & \dots & 0 & 1 \end{pmatrix}$$

La función asociada a una transformación afín

Consideramos la función F la asociada a una transformación afín T en el marco \mathcal{R} anterior. Sean $\mathbf{c} = (c_0, \dots, c_{n-1}, w)$ las coordenadas en (en \mathcal{R}) de un punto o vector $\mathcal{R}\mathbf{c}$ cualquiera:

Sabemos que F implementa T , es decir $T(\mathcal{R}\mathbf{c}) = \mathcal{R}F(\mathbf{c})$. Puesto que T y \mathcal{R} son lineales, para un punto $\dot{\mathbf{p}} = \mathcal{R}\mathbf{p}$ y un vector $\vec{v} = \mathcal{R}\mathbf{v}$, y un real a se cumplirá:

$$T(\dot{\mathbf{p}} + a\vec{v}) = \mathcal{R}(F(\mathbf{p}) + aF(\mathbf{v}))$$

Como consecuencia, las coordenadas transformadas $F(\mathbf{c})$ se pueden escribir usando las coordenadas transformadas de \mathcal{R} y la tupla \mathbf{c} :

$$\mathcal{R}\mathbf{c} = w\dot{\mathbf{o}} + \sum_{j=0}^{n-1} c_j \vec{b}_j \quad \Rightarrow \quad F(\mathbf{c}) = w\mathbf{o} + \sum_{j=0}^{n-1} c_j \mathbf{b}_j$$

Lo cual implica que la función F se puede expresar usando la matriz M_T :

$$F(\mathbf{c}) = M_T \mathbf{c}$$

Implementación de transformaciones afines con matrices

El resultado anterior nos dice que **podemos implementar una transformación afín usando su matriz asociada:**

- Eso permite implementar transformaciones afines en un programa, simplemente multiplicando la matriz (a la izquierda) por las coordenadas homogéneas del punto o vector a transformar (a la derecha).
- Si T es no singular, entonces M_T será invertible, y podremos implementar la transformación inversa T^{-1} usando la inversa de la matriz M_T^{-1} .
- La matriz M_T se puede descomponer como el producto de otras matrices cuadradas

$$M_T = D_T R_T$$

es decir, aplicar M_T equivale a aplicar primero R_T y luego D_T , donde:

- ▶ R_T es la matriz que transforma los vectores, y
- ▶ D_T es la matriz que desplaza el origen.

Descomposición de matrices en 3D.

A modo de ejemplo, en 3D, consideramos un marco $\mathcal{R} = (\vec{e}_x, \vec{e}_y, \vec{e}_z, \dot{o})$ y una transformación afín T , consideramos las coordenadas (en \mathcal{R}) de los vectores de la base y el origen transformados:

$$\begin{aligned} T(\vec{e}_x) &= \mathcal{R}(x_0, y_0, z_0, 0)^T & T(\vec{e}_y) &= \mathcal{R}(x_1, y_1, z_1, 0)^T \\ T(\vec{e}_z) &= \mathcal{R}(x_2, y_2, z_2, 0)^T & T(\dot{o}) &= \mathcal{R}(d_x, d_y, d_z, 1)^T \end{aligned}$$

Entonces se cumple:

$$M_T = \begin{pmatrix} x_0 & x_1 & x_2 & d_x \\ y_0 & y_1 & y_2 & d_y \\ z_0 & z_1 & z_2 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 & x_1 & x_2 & 0 \\ y_0 & y_1 & y_2 & 0 \\ z_0 & z_1 & z_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = D_T R_T$$

Composición de transformaciones y matrices. Inversas.

Las transformaciones afines se pueden componer: si T_1 y T_2 son dos transformaciones afines:

- La composición de ambas es otra transformación afín S , que se escribe como $T_2 \circ T_1$ y que supone transformar primero por T_1 y luego por T_2 , es decir, para un punto \dot{p} se cumple:

$$S(\dot{p}) = T_2(T_1(\dot{p}))$$

- En un marco fijo, la matriz M_S asociada a $T_2 \circ T_1$ se obtiene multiplicando las matrices M_2 y M_1 correspondientes a T_2 y T_1 , respectivamente:

$$M_S = M_2 M_1$$

- Puesto que $M_T = D_T R_T$, la inversa de M_T se puede escribir como:

$$M_T^{-1} = (R_T D_T)^{-1} = R_T^{-1} D_T^{-1}$$

Transformación de marcos afines

Dada una transformación T **no singular** y el marco afín $\mathcal{R} = (\vec{b}_0, \dots, \vec{b}_{n-1}, \dot{o})$ podemos definir el marco afín *transformado* \mathcal{S} así:

$$\mathcal{S} := (T(\vec{b}_0), \dots, T(\vec{b}_{n-1}), T(\dot{o})) \quad \text{donde:} \quad \begin{cases} T(\vec{b}_j) = \mathcal{R}(c_{0,j}, \dots, c_{n-1,j}, 0)^T \\ T(\dot{o}) = \mathcal{R}(p_0, \dots, p_{n-1}, 1)^T \end{cases}$$

Entonces se pueden escribir los vectores de la base y el origen transformados como combinaciones lineales de los vectores de la base y el origen originales:

$$T(\vec{b}_j) = 0\dot{o} + \sum_{i=0}^{n-1} c_{i,j} \vec{b}_i \quad \text{y} \quad T(\dot{o}) = 1\dot{o} + \sum_{i=0}^{n-1} p_i \vec{b}_i$$

Lo cual significa que podemos escribir el marco \mathcal{S} **multiplicando el marco \mathcal{R} por la matriz M_T** (por la derecha):

$$\mathcal{S} = \mathcal{R} M_T$$

donde M_T es la matriz asociada a T (en coordenadas de \mathcal{R}).

Cambio de coordenadas entre marcos afines

Dados dos marcos afines \mathcal{A} y \mathcal{B} , siempre existe una transformación afín T que transforma \mathcal{A} en \mathcal{B} . Dicha transformación tendrá asociada una matriz M (en \mathcal{A}) tal que $\mathcal{A}M = \mathcal{B}$. Podemos nombrar esa matriz como $M_{\mathcal{AB}}$, tiene las coordenadas de \mathcal{B} en \mathcal{A} .

Si un punto o vector tiene coordenadas $\mathbf{c}_{\mathcal{A}}$ en \mathcal{A} y $\mathbf{c}_{\mathcal{B}}$ en \mathcal{B} , entonces se cumple:

$$\mathcal{A}\mathbf{c}_{\mathcal{A}} = \mathcal{B}\mathbf{c}_{\mathcal{B}}$$

Pero podemos sustituir \mathcal{B} por $\mathcal{A}M_{\mathcal{AB}}$ y usando la asociatividad, obtenemos:

$$\mathcal{A}\mathbf{c}_{\mathcal{A}} = \mathcal{B}\mathbf{c}_{\mathcal{B}} = (\mathcal{A}M_{\mathcal{AB}})\mathbf{c}_{\mathcal{B}} = \mathcal{A}M_{\mathcal{AB}}\mathbf{c}_{\mathcal{B}} = \mathcal{A}(M_{\mathcal{AB}}\mathbf{c}_{\mathcal{B}})$$

Puesto que las coordenadas en \mathcal{A} son únicas, se deduce que:

$$\mathbf{c}_{\mathcal{A}} = M_{\mathcal{AB}}\mathbf{c}_{\mathcal{B}}$$

Es decir, la matriz $M_{\mathcal{AB}}$ permite hacer el cambio de coordenadas desde las coordenadas en \mathcal{B} a las coordenadas en \mathcal{A} .

Interpretaciones de la acción de una matriz

Si tenemos una matriz M cualquiera (con determinante no nulo), y un marco afín \mathcal{A} , siempre existe una transformación afín T con matriz M que transforma \mathcal{A} en otro marco \mathcal{B} . Esto implica que podemos ver la matriz M de varias formas distintas:

- Como algo que implementa la función F de T : dadas las coordenadas \mathbf{c} de un punto o vector relativos a \mathcal{A} , nos da las coordenadas $F(\mathbf{c})$ del punto o vector transformado, también relativas a \mathcal{A} :

$$F(\mathbf{c}) = M\mathbf{c}$$

- Permite hacer el cambio de coordenadas desde las coordenadas en \mathcal{B} a las coordenadas en \mathcal{A} . Si $\mathbf{c}_{\mathcal{B}}$ son coordenadas en \mathcal{B} y $\mathbf{c}_{\mathcal{A}}$ las correspondientes en \mathcal{A} (ambas de un mismo punto o vector), entonces:

$$\mathbf{c}_{\mathcal{A}} = M\mathbf{c}_{\mathcal{B}}$$

- Como algo que transforma el marco \mathcal{A} en el marco \mathcal{B} ya que

$$\mathcal{B} = \mathcal{A}M$$

Subsección 2.3.

Tipos de transformaciones.

Transformaciones isométricas

Una transformación T es una transformación **isométrica** si conserva el valor absoluto del producto escalar, es decir, para dos vectores cualquiera \vec{u} y \vec{v} se cumple:

$$\| T(\vec{u}) \cdot T(\vec{v}) \| = \| \vec{u} \cdot \vec{v} \|$$

Como consecuencia T

- es una transformación afín, pero no conserva *la orientación*,
- conserva las distancias entre puntos, es decir conserva la longitud de los vectores: se cumple $\|T(\vec{v})\| = \|\vec{v}\|$,
- conserva el valor absoluto del ángulo entre dos líneas,
- conserva las áreas y volúmenes de los objetos, y
- tendrá asociada una matriz R_T (en un marco cartesiano cualquiera) que será ortonormal, y con determinante igual a $+1$ o a -1 .

Las **traslaciones, las rotaciones y las reflexiones** son ejemplos de isometrías.

Transformaciones rígidas.

Una transformación T que conserva el producto escalar (incluyendo el signo) es una transformación **rígida** u **ortogonal**. Para dos vectores cualquiera \vec{u} y \vec{v} se cumple:

$$T(\vec{u}) \cdot T(\vec{v}) = \vec{u} \cdot \vec{v}$$

Por tanto T :

- es afín y conserva la *orientación*,
- es isométrica,
- conserva las distancias, áreas y volúmenes,
- conserva los ángulos (en magnitud y signo), y
- tendrá asociada una matriz R_T (en un marco cartesiano cualquiera) que será ortonormal, y con determinante igual a +1.

Las **traslaciones y rotaciones** son rígidas (no las reflexiones).

Sección 3.

Transformaciones usuales en Informática Gráfica.

1. Traslaciones
2. Escalados y reflexiones.
3. Cizallas
4. Rotaciones

Introducción

En esta sección estudiaremos las transformaciones usuales en Informática Gráfica, tanto en el espacio 2D como 3D.

- Traslaciones.
- Escalados: uniformes, no uniformes, reflexiones.
- Cizallas (*shearings*).
- Rotaciones: entorno a al origen (en 2D), entorno a los ejes de coordenadas o entorno a un eje arbitrario (en 3D).

En cada caso vemos las propiedades y las correspondientes matrices, relativas siempre a un marco \mathcal{R} en 2D o 3D (que en algunos casos será cartesiano). Para introducir las transformaciones se indica como se transforma el origen y los versores de dicho marco cartesiano, y eso nos da directamente la matriz.

Subsección 3.1.

Traslaciones

Traslaciones en 2D y 3D

Una **traslación** T por un vector \vec{d} es una transformación afín rígida que a cada punto \dot{p} le asocia el punto $\dot{p} + \vec{d}$, y lógicamente no afecta a los vectores.

En el espacio 2D, para un marco $\mathcal{R} = (\vec{x}, \vec{y}, \dot{o})$, la traslación T por el vector \vec{d} de coordenadas $\mathbf{d} = (d_x, d_y, 0)^T$ en \mathcal{R} viene dada por:

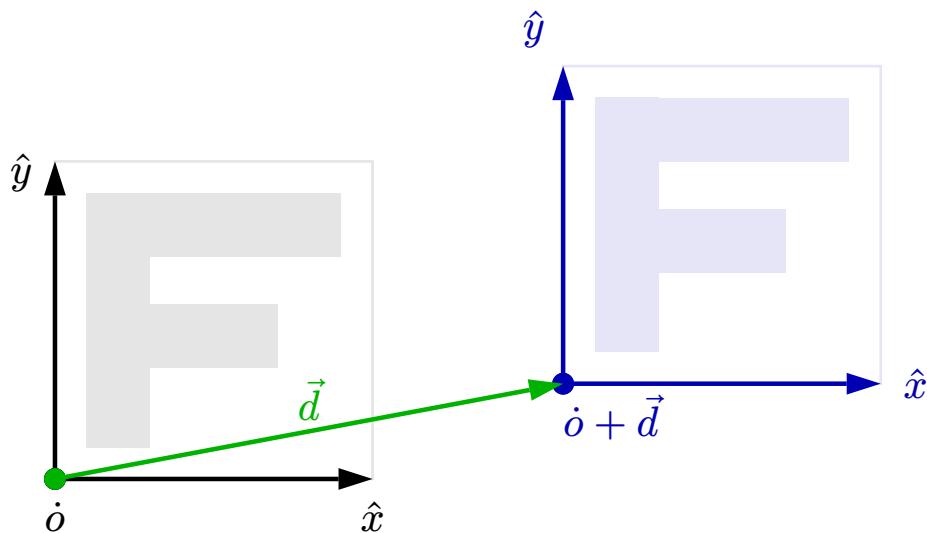
$$\begin{aligned} T_{\mathbf{d}}(\vec{x}) &= \vec{x} \\ T_{\mathbf{d}}(\vec{y}) &= \vec{y} \\ T_{\mathbf{d}}(\dot{o}) &= \dot{o} + \vec{d} \end{aligned} \quad M_T = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix}$$

En 3D, para un marco $\mathcal{R} = (\vec{x}, \vec{y}, \vec{z}, \dot{o})$, la traslación T por el vector \vec{d} de coordenadas $\mathbf{d} = (d_x, d_y, d_z, 0)^T$ en \mathcal{R} viene dada por:

$$\begin{aligned} T_{\mathbf{d}}(\vec{x}) &= \vec{x} \\ T_{\mathbf{d}}(\vec{y}) &= \vec{y} \\ T_{\mathbf{d}}(\vec{z}) &= \vec{z} \\ T_{\mathbf{d}}(\dot{o}) &= \dot{o} + \vec{d} \end{aligned} \quad M_T = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Esquema de la traslación en 2D

Vemos una traslación en 2D por un vector $\mathbf{d} = (1.6, 0.3)$, en negro el marco original y en azul el desplazado. En verde vemos el vector de desplazamiento. Los ejes no cambian.



Subsección 3.2.

Escalados y reflexiones.

Escalados.

Una **escalado** E_s es una transformación afín que multiplica las coordenadas de un punto o vector en un marco \mathcal{R} cualquiera por n factores escalares que forman la tupla $s = (s_0, \dots, s_{n-1})$. Las distancias al origen se ven por tanto multiplicadas por esos factores. El origen de \mathcal{R} no cambia, y se llama **foco de escalado**.

En 2D, la tupla de los dos factores es: $s = (s_x, s_y)$

$$\begin{aligned} E_s(\vec{x}) &= s_x \vec{x} \\ E_s(\vec{y}) &= s_y \vec{y} \\ E_s(\vec{o}) &= \vec{o} \end{aligned} \qquad M_T = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

En 3D, la tupla es: $s = (s_x, s_y, s_z)$

$$\begin{aligned} E_s(\vec{x}) &= s_x \vec{x} \\ E_s(\vec{y}) &= s_y \vec{y} \\ E_s(\vec{z}) &= s_z \vec{z} \\ E_s(\vec{o}) &= \vec{o} \end{aligned} \qquad M_T = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Tipos de escalados

Las transformaciones de escalado pueden clasificarse en base a los factores de escala, suponiendo que se define en un marco \mathcal{C} cartesiano:

- Si todos los factores de escala son todos iguales a un valor s , se dice que el escalado es **uniforme**, conserva los ángulos y las proporciones de los vectores, ya que la longitud de un vector se multiplica por s al transformar.
- Si los factores son distintos, el escalado es **no uniforme**, y no conserva los ángulos ni las proporciones.

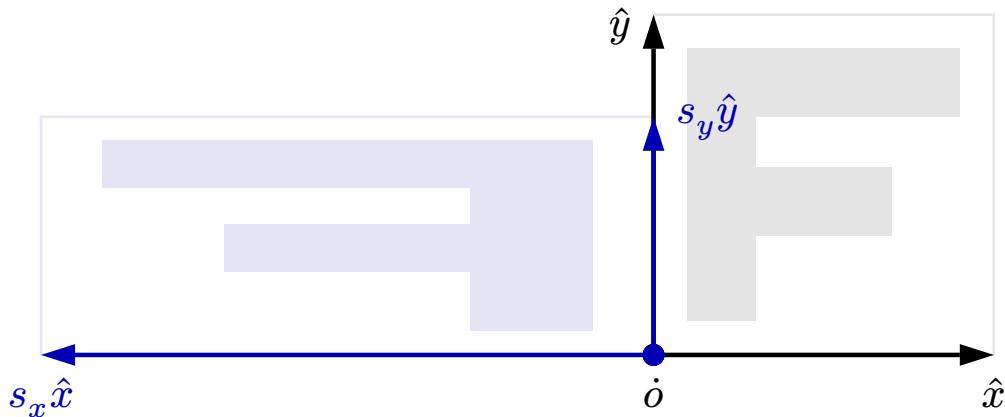
También en un marco cartesiano, podemos analizar el signo del producto de los factores (es el determinante de la matriz R_T)

- Si el producto es **positivo**, el escalado **conserva la orientación** del sistema de referencia.
- Si el producto es **negativo**, el escalado **invierte la orientación** del sistema de referencia.

Esquema de escalados en 2D

Vemos un escalamiento en 2D por un vector, en negro el marco original y en azul el escalado. Se usan los factores de escala $s_x = -1.8$ y $s_y = 0.7$.

El cuadrado unidad con la figura en gris se transforma en el rectángulo con la figura en azul, invirtiendo la orientación del marco (ya que el signo de $s_x s_y$ es negativo).



Reflexiones

Una **reflexión** S_i es un caso particular de escalado. Son escalados definidos en un marco cartesiano \mathcal{C} que cumplen:

- Los factores de escala son todos 1, excepto uno de ellos, el correspondiente al versor \hat{e}_i que es -1 . Por eso, son su propia inversa, es decir $S_i^2 = I$.
- No conservan la orientación: en 2D y 3D convierten un marco a derechas en uno a izquierdas y al revés.
- Conservan las longitudes de los vectores, y la magnitud de los ángulos (no su signo). Así que son **isométricas**, pero no **rígidas**.
- La transformación invierte el signo de una de las coordenadas de cualquier punto o vector, con lo cual es una **reflexión** cuyo *eje o plano de reflexión* (una línea o plano invariante) es el la línea o el plano paralelo a \hat{e}_i que pasa por el origen.
- Las distancias al eje o plano se conservan, pero el punto o vector pasa al otro lado del mismo.

Reflexiones con eje o planos arbitrarios

Una reflexión $S_{\hat{n}}$ puede hacerse teniendo como eje una línea cualquiera por el origen (en 2D) o con un plano de reflexión por el origen (en 3D), en cualquier caso la línea o el plano es perpendicular a un versor \hat{n} , no son necesariamente paralelo a los ejes de coordenadas.

Las coordenadas del versor \hat{n} serán $\mathbf{n} = (n_x, n_y, n_z)$ en 3D o $\mathbf{n} = (n_x, n_y)$ en 2D. En estas condiciones, la matriz de transformación asociada es la **Matriz de Householder** M_S

$$M_S \equiv I - 2(\mathbf{n}\mathbf{n}^T)$$

donde I es la matriz identidad y $\mathbf{n}\mathbf{n}^T$ es esta matriz (en 2D y 3D):

$$\mathbf{n}\mathbf{n}^T \equiv \begin{pmatrix} n_x n_x & n_x n_y & 0 \\ n_x n_y & n_y n_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \mathbf{n}\mathbf{n}^T \equiv \begin{pmatrix} n_x n_x & n_x n_y & n_x n_z & 0 \\ n_x n_y & n_y n_y & n_y n_z & 0 \\ n_x n_z & n_y n_z & n_z n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Subsección 3.3.

Cizallas

Cizallas.

Una **cizalla** (*shear*) C_{ij} es una transformación afín que, en un marco \mathcal{R} cualquiera, incrementa la i -ésima coordenada de una tupla, usando un incremento proporcional a la j -ésima coordenada de esa tupla (con $i \neq j$).

- El valor transformado c'_i de la i -ésima coordenada será $c'_i = c_i + ac_j$, donde a es un parámetro real que define la cizalla, y c_i y c_j las coordenadas originales. Las otras coordenadas c_j (con $i \neq j$) no cambian.
- La matriz asociada es igual a la matriz identidad, excepto que el valor en la fila i y columna j es a en lugar de 0.
- El marco transformado por C_{ij} es igual al original, excepto que en el marco transformado el eje $T(\vec{e}_j)$ se hace igual a $\vec{e}_j + a\vec{e}_i$.
- Las cizallas no conservan las longitudes ni los ángulos en general. Por eso son transformaciones **no rígidas**.
- Las cizallas únicamente conservan longitudes en líneas perpendiculares a \vec{e}_j . En 3D conservan ángulos entre vectores perpendiculares a \vec{e}_j .

Matrices de las cizallas en 2D.

En 2D hay dos tipos de cizallas, las llamamos C_{01} y C_{10} . Ambas dependen del parámetro real a .

- La cizalla C_{01} incrementa la coordenada X según la Y por tanto el marco transformado y la matriz son:

$$\begin{aligned}C_{01,a}(\vec{x}) &= \vec{x} \\C_{01,a}(\vec{y}) &= \vec{y} + a\vec{x} \\C_{01,a}(\dot{o}) &= \dot{o}\end{aligned} \qquad M_T = \begin{pmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- La otra cizalla en 2D es C_{10} , que incrementa la coordenada Y según la X, así que ahora tenemos:

$$\begin{aligned}C_{10,a}(\vec{x}) &= \vec{x} + a\vec{y} \\C_{10,a}(\vec{y}) &= \vec{y} \\C_{10,a}(\dot{o}) &= \dot{o}\end{aligned} \qquad M_T = \begin{pmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Matrices de las cizallas en 3D.

En 3D hay 6 tipos de cizallas, ya que hay 6 posibles pares i, j distintos.

Al igual que en 2D:

- La matriz es igual a la matriz identidad, excepto que en la fila i y columna j aparece el valor a en lugar de 0.
- El eje \hat{e}_j se incrementa por $a\hat{e}_i$.

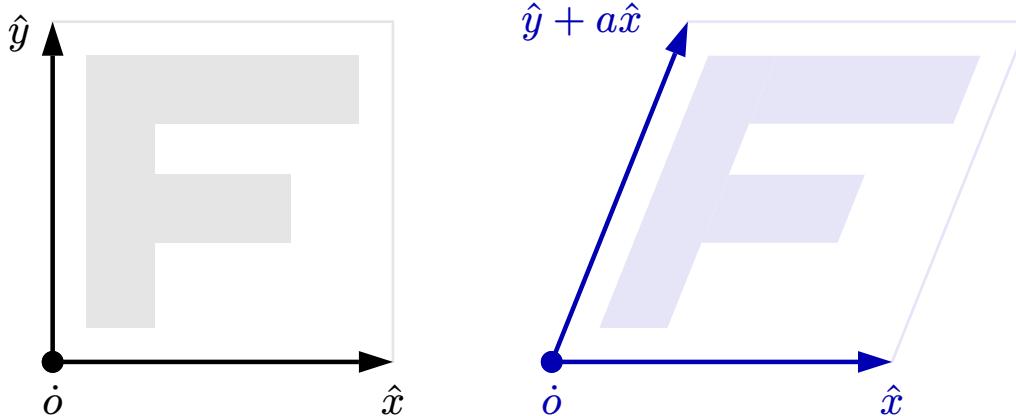
A modo de ejemplo, vemos la cizalla C_{21} que incrementa la coordenada Z ($i = 2$) en base a la Y ($j = 1$)

$$\begin{aligned}C_{21,a}(\vec{x}) &= \vec{x} \\C_{21,a}(\vec{y}) &= \vec{y} + a\vec{z} \\C_{21,a}(\vec{z}) &= \vec{z} \\C_{21,a}(\dot{o}) &= \dot{o}\end{aligned} \quad M_T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & a & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Esquema de una cizalla en 2D

Vemos la cizalla C_{01} en 2D con $a = 0.4$. Los puntos se desplazan en el eje X una distancia proporcional a su coordenada Y.

El cuadrado unidad con la figura en gris (a la izquierda) se transforma en el paralelogramo con la figura en azul a la derecha (la transformación no conlleva traslación, pero se han desplazado las figuras para apreciarlas mejor).



Subsección 3.4.

Rotaciones

Rotaciones en 2D

Una transformación de rotación R en un marco cartesiano $\mathcal{C} = (\hat{x}, \hat{y}, \dot{o})$, es una transformación afín rígida que:

- lleva cada punto \dot{p} a otro a la misma distancia de \dot{o} que el original. Se dice que \dot{o} es el **centro de rotación**.
- depende de un parámetro θ , llamado **ángulo de rotación**, que es el ángulo en radianes (con signo) entre \dot{p} y $R(\dot{p})$. Si $\theta > 0$, la rotación es antihoraria, y si $\theta < 0$ es horaria.

Así que la transformación de los versores y el origen, y la matriz son:

$$R_\theta(\hat{x}) = (\cos \theta)\hat{x} + (\sin \theta)\hat{y}$$

$$R_\theta(\hat{y}) = -(\sin \theta)\hat{x} + (\cos \theta)\hat{y}$$

$$R_\theta(\dot{o}) = \dot{o}$$

$$M_R = \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Componentes de los ejes rotados en 2D

Vemos una rotación en 2D un ángulo $\theta = 35^\circ$, que es mayor que 0. En negro aparece un marco \mathcal{R} y en azul el marco transformado. Ambos tienen el mismo origen.

$$R_\theta(\hat{y}) =$$

$$-(\sin \theta)\hat{x} + (\cos \theta)\hat{y}$$

$$\hat{y}$$

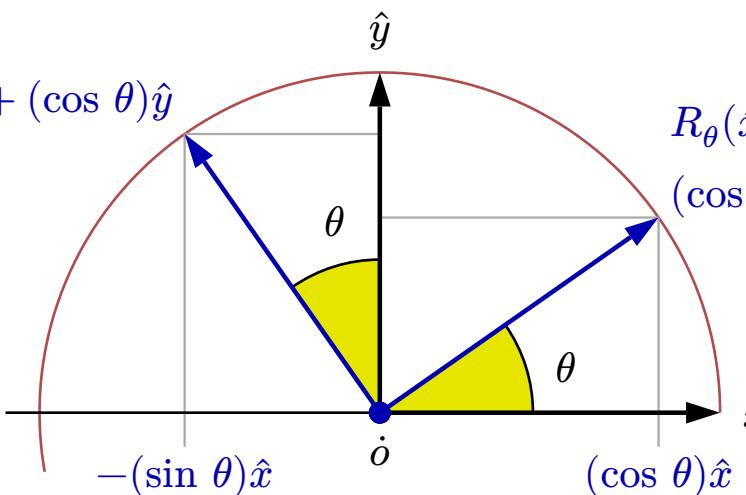
$$R_\theta(\hat{x}) =$$

$$(\cos \theta)\hat{x} + (\sin \theta)\hat{y}$$

$$\hat{o}$$

$$\theta$$

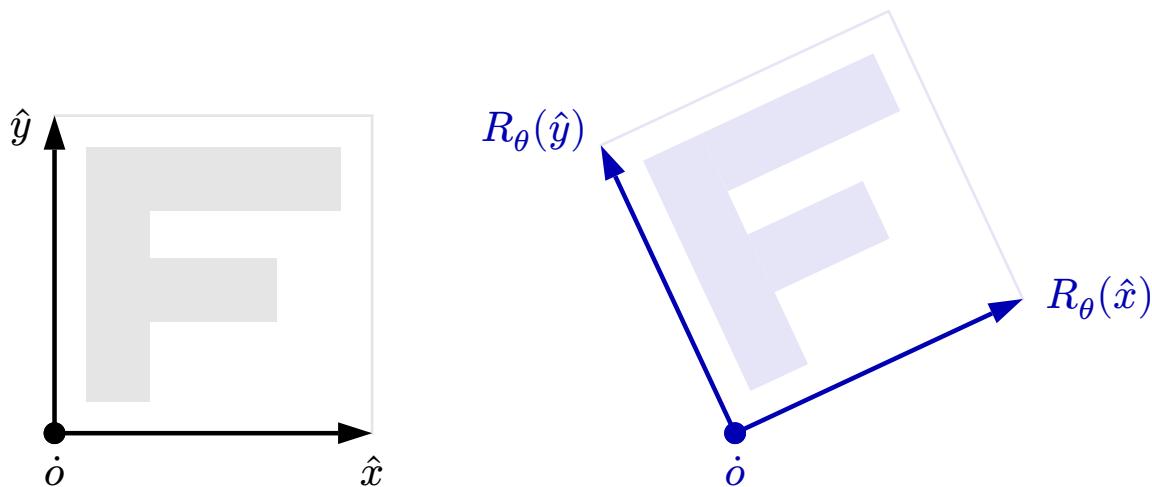
$$(\cos \theta)\hat{x}$$



Esquema de una rotación en 2D

Vemos una rotación en R_θ con $\theta = 25^\circ$. Los puntos se rotan entorno al origen.

El cuadrado de lado 1 con la figura en gris (a la izquierda) se transforma en el cuadrado con la figura en azul a la derecha (la transformación no conlleva traslación, pero se han desplazado las figuras para apreciarlas mejor).



Problemas: rotaciones 2D (1/3)

Problema 3.4:

Demuestra que el producto escalar de vectores en 2D es invariante por rotación, es decir, que para cualquier ángulo θ y vectores \vec{u} y \vec{v} se cumple:

$$R_\theta(\vec{u} \cdot \vec{v}) = R_\theta(\vec{u}) \cdot R_\theta(\vec{v})$$

(para demostrarlo usa las coordenadas de los vectores en un marco cartesiano cualquiera)

Problema 3.5:

Demuestra que en 2D las rotaciones no modifican la longitud de un vector, es decir, que para cualquier ángulo θ y vector \vec{v} , se cumple:

$$\| R_\theta(\vec{v}) \| = \| \vec{v} \|$$

Problemas: rotaciones 2D (2/3)

Problema 3.6:

Demuestra que si rotamos en 2D un vector +90 grados o -90 grados, obtenemos otro vector perpendicular al original, es decir, si $\|\theta\| = \pi/2$ entonces

$$\vec{v} \cdot R_\theta(\vec{v}) = 0.$$

Problema 3.7:

Demuestra que una matriz de rotación en 2D es siempre ortogonal, independientemente del ángulo. Eso quiere decir que sus filas son perpendiculares dos a dos, e igual ocurre con sus columnas, y además cada fila y columna tienen longitud 1.

Problemas: rotaciones 2D (3/3)

Problema 3.8:

Demuestra que, en 2D, el producto de una matriz de rotación y una de escalado no es conmutativo en general, excepto si el escalado es uniforme.

Problema 3.9:

Demuestra que en 2D, el producto de una matriz de rotación y otra de traslación (por un vector no nulo) no es conmutativo.

Rotaciones en 3D

Una transformación de rotación R_θ en un marco cartesiano $\mathcal{C} = (\hat{x}, \hat{y}, \hat{z}, \dot{o})$, es una transformación afín rígida que:

- lleva cada punto \dot{p} a otro a la misma distancia que \dot{p} de una línea que pasa por \dot{o} , y es paralela a un versor \hat{e} (a ese versor o a esa línea se les llama **eje de rotación**).
- también depende de un **ángulo de rotación** θ en radianes entre \dot{p} y $R(\dot{p})$.
- cuando $\theta > 0$, la rotación es antihoraria, vista desde un punto situado en el eje de rotación y mirando hacia el origen. Si $\theta < 0$, la rotación es horaria.
- el eje puede ser paralelo a uno de los tres versores de \mathcal{C} , o puede ser otro cualquiera.

Rotaciones en 3D entorno a los ejes cartesianos. Eje Z

En este caso el eje de rotación \hat{e} es alguno de los versores del marco \mathcal{C} , es decir es \hat{x} , \hat{y} o \hat{z} . Las coordenadas en el eje \hat{e} no cambian, ya que las rotaciones ocurren en un plano perpendicular a \hat{e} . Las otras dos coordenadas cambian igual que en 2D.

Nombramos las rotaciones indicando ángulo y eje. Para una rotación entorno al eje Z tenemos las mismas expresiones que en 2D para X e Y:

$$R_{z,\theta}(\hat{x}) = (\cos \theta)\hat{x} + (\sin \theta)\hat{y}$$

$$R_{z,\theta}(\hat{y}) = -(\sin \theta)\hat{x} + (\cos \theta)\hat{y}$$

$$R_{z,\theta}(\hat{z}) = \hat{z}$$

$$R_{z,\theta}(\dot{o}) = \dot{o}$$

$$M_R = \begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotaciones en 3D entorno a los ejes cartesianos. Ejes Y y X

Para rotaciones entorno al eje Y:

$$R_{y,\theta}(\hat{x}) = (\cos \theta)\hat{x} - (\sin \theta)\hat{z}$$

$$R_{y,\theta}(\hat{y}) = \hat{y}$$

$$R_{y,\theta}(\hat{z}) = (\sin \theta)\hat{x} + (\cos \theta)\hat{z}$$

$$R_{y,\theta}(\dot{o}) = \dot{o}$$

$$M_R = \begin{pmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Para rotaciones entorno al eje X:

$$R_{x,\theta}(\hat{x}) = \hat{x}$$

$$R_{x,\theta}(\hat{y}) = (\cos \theta)\hat{x} + (\sin \theta)\hat{y}$$

$$R_{x,\theta}(\hat{z}) = -(\sin \theta)\hat{x} + (\cos \theta)\hat{y}$$

$$R_{x,\theta}(\dot{o}) = \dot{o}$$

$$M_R = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Problemas: rotaciones 3D entorno a los ejes cartesianos(1/n)

Considera las rotaciones en 3D entorno a \hat{e} , que es uno de los ejes \hat{x} , \hat{y} y \hat{z} de un **marco cartesiano 3D**.

Problema 3.10:

Demuestra que el producto escalar de vectores en 3D es invariante por estas rotaciones. y que tampoco modifican la longitud de un vector. Te puedes basar en los problemas similares en 2D.

Problema 3.11:

Demuestra que el producto vectorial de dos vectores rota igual que lo hacen esos dos vectores, es decir, que para cualquiera dos vectores \vec{u} y \vec{v} y un ángulo θ , se cumple:

$$R_{\theta, \hat{e}}(\vec{u} \times \vec{v}) = R_{\theta, \hat{e}}(\vec{u}) \times R_{\theta, \hat{e}}(\vec{v})$$

Rotaciones en 3D de ejes arbitrarios

En Informática Gráfica se puede necesitar hacer rotaciones entorno a ejes que son distintos de los ejes cartesianos.

Suponiendo que el eje es un versor \hat{e} (unitario), entonces se puede escribir el vector rotado usando la llamada **fórmula de Rodrigues**:

$$R_{\hat{e}, \theta}(\vec{v}) = (\cos \theta)\vec{v} + (1 - \cos \theta)(\vec{v} \cdot \hat{e})\hat{e} + (\sin \theta)\hat{e} \times \vec{v}$$

Si las coordenadas de \hat{e} en \mathcal{C} son $(e_0, e_1, e_2, 0)$, podemos escribir la matriz M_R como una suma:

$$M_R = \begin{pmatrix} a_{00} & a_{01} & a_{02} & 0 \\ a_{10} & a_{11} & a_{12} & 0 \\ a_{20} & a_{21} & a_{22} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} c & -se_2 & se_1 & 0 \\ se_2 & c & -se_0 & 0 \\ -se_1 & se_0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ con } \begin{cases} a_{ij} := (1 - c)e_i e_j \\ c := \cos \theta \\ s := \sin \theta \end{cases}$$

Los ángulos de Euler para rotaciones en 3D

Otra forma alternativa de definir una rotación 3D de eje arbitrario es usando una tupla de tres valores reales (α, β, γ) llamados **ángulos de Euler**, cada uno de ellos en radianes.

Una rotación de eje arbitrario $R_{\hat{e}, \theta}$ será equivalente a la composición de tres rotaciones entorno cada uno de los tres ejes $\hat{x}, \hat{y}, \hat{z}$ de un marco cartesiano, usando los ángulos de Euler:

$$R_{\hat{e}, \theta} = R_{z, \gamma} \circ R_{y, \beta} \circ R_{x, \alpha}$$

Usar esto puede ser útil si conocemos los tres ángulos y queremos obtener la matriz (independientemente de \hat{e} y θ , que no conoceremos), un ejemplo son las *cámaras orbitales*. Sin embargo:

- Si partimos del eje \hat{e} y el ángulo θ no es sencillo calcular los ángulos de Euler que producen la misma rotación.
- Fijado \hat{e} y θ , puede haber más de una tripleta de ángulos de Euler que den lugar a la misma rotación.

Cuaterniones para rotaciones en 3D

Otra alternativa es usar los **cuaterniones** (*quaternions*) que son objetos que se pueden ver como una extensión de los números complejos a 4 dimensiones. Un cuaternion q tiene la forma:

$$q = a + bi + cj + dk$$

donde a, b, c y d son reales, y i, j y k son tres unidades imaginarias distintas. Al igual que los complejos, los cuaterniones se pueden sumar, restar y multiplicar entre ellos. Las propiedades de los cuaterniones permiten:

- Representar rotaciones en 3D de eje y ángulo arbitrarios con un cuaternion (es decir, con una tupla de 4 reales en memoria).
- Multiplicar cuaterniones para componer rotaciones.
- Representar cualquier vector en 3D como un cuaternion.
- **Rotar un vector alrededor de un eje arbitrario de forma muy eficiente**, usando los dos cuaterniones que representan la rotación y el vector y obteniendo el cuaternion del vector rotado.

Rotaciones de ejes arbitrarios: alternativas

Si una rotación debe ser aplicada a muchos vectores en la GPU, entonces es mejor calcular una vez la matriz M_R y usarla para aplicarla a cada vector. Se puede hacer de dos formas:

- Calculando la matriz M_T indicada antes.
- Componiendo: (1) una matriz de rotación R que alinea \hat{e} con \hat{x} , (2) la rotación por θ entorno a \hat{x} y finalmente (3) la rotación inversa R^{-1} de la primera. Es más complejo que calcular la matriz directamente.

Para rotar pocos vectores en la CPU puede ser más eficiente no calcular la matriz, se puede hacer de dos formas:

- Usando la fórmula de Rodrigues directamente.
- Usando cuaterniones.

La fórmula de Rodrigues y los cuaterniones son equivalentes, y para pocos vectores ambas formas son mucho más eficientes que calcular la matriz. Los cuaterniones, sin embargo, son un poco más eficientes que la fórmula de Rodrigues.

Sección 4.

Transformaciones en Godot.

1. Tuplas de valores reales.
2. Matrices de transformación.

Introducción

En esta sección estudiaremos como se representan las tuplas de coordenadas homogéneas en 2D y 3D en Godot (clases `Vector2` y `Vector3`), y las matrices de transformación (clases `Transform2D` y `Transform3D`).

También veremos como se implementan las transformaciones más comunes: traslaciones, rotaciones y escalados, mediante el atributo `transform` de las clases `Node2D` y `Node3D`.

Veremos como la transformación de un nodo en el árbol de escena afecta a ese nodo y a todos sus hijos.

Subsección 4.1.

Tuplas de valores reales.

Tuplas 2D. Clase Vector2.

Un objeto de la clase **Vector2** representa una tupla de dos valores reales, que se puede usar para guardar coordenadas de puntos o vectores en 2D.

- Contienen dos valores reales representados en coma flotante con 32 bits de precisión (equivalente a un **float** de C/C++, pero no a un **float** de GDscript, que tiene 64 bits).
- Los elementos son accesibles como **v.x** y **v.y**, o como **v[e]** donde *e* es una expresión entera que se evalúa a 0 o 1.
- Estos objetos se pueden sumar, restar y multiplicar por un escalar (un **float** de GDscript), usando los operadores binarios infijos **+**, **-** y *****.
- También se pueden usar diversos métodos para otras operaciones:
 - ▶ **v.length()** devuelve la longitud del vector **v** (un **float**)
 - ▶ **v.normalized()** devuelve el vector **v**/ $\|v\|$ (**Vector2**), si $\|v\| > 0$.
 - ▶ **v.dot(u)** devuelve el producto escalar de **v · u** (un **float**).

Hay otros muchos métodos, consultar:

docs.godotengine.org/en/stable/classes/class_vector2.html

Tuplas 3D. Clase Vector3.

Un objeto de la clase **Vector3** representa una tupla de tres valores reales, que se puede usar para guardar coordenadas de puntos o vectores en 3D.

- Contienen tres valores reales representados en coma flotante con 32 bits de precisión
- Los elementos son accesibles como **v.x**, **v.y**, **v.z**, o como **v[e]** donde *e* es una expresión entera que se evalúa a 0, 1 o 2.
- Estos objetos se pueden sumar, restar y multiplicar por un escalar (un **float** de GDscript), usando los operadores binarios infijos **+**, **-** y *****.
- También se pueden usar diversos métodos para otras operaciones:
 - ▶ **v.length()** devuelve la longitud del **v** (**float**)
 - ▶ **v.normalized()** devuelve **v** / $\|v\|$ (**Vector3**), si $\|v\| > 0$.
 - ▶ **v.dot(u)** devuelve el producto escalar de **v · u** (**float**).
 - ▶ **v.cross(u)** devuelve el producto vectorial **v × u** (otro **Vector3**).

Consultar: docs.godotengine.org/en/stable/classes/class_vector3.html

Otros clases relacionadas

Existen otras clases para tuplas:

- Las clases `Vector2i` y `Vector3i` representan tuplas de dos y tres valores enteros (32 bits con signo). Una tupla `Vector3i` se puede usar, por ejemplo, para guardar los tres índices de un triángulo en una malla.
- La clase `Vector4` representa tuplas de cuatro valores reales (32 bits con signo). Se puede usar, por ejemplo, para representar un cuaternión.

Así como otros tipos de clases:

- `Basis`: bases del espacio vectorial en 3D.
- `Line2D`: líneas en 2D.
- `Plane`: planos en 3D.

Ahora veremos las clases para matrices de transformación:

- `Transform2D`: matrices 3x3 en 2D.
- `Transform3D`: matrices 4x4 en 3D.

Subsección 4.2.

Matrices de transformación.

Matrices de transformación 2D. Clase *Transform2D*

Un objeto de la clase **Transform2D** representa una matriz 3×3 que implementa una transformación afín en el espacio afín 2D A_2 .

- Se guardan seis valores reales de 32 bits, las dos primeras filas de la misma. La tercera fila no se guarda y siempre es $(0, 0, 1)$.
- Se pueden multiplicar (componer) entre ellas con el operadores binarios infijo *****.
- Se pueden multiplicar o dividir por un **float** con el operador ***** o **/**.
- Se pueden aplicar a un **Vector2** con el operador ***** por la derecha, devuelve el **Vector2** resultado de aplicar la transformación (añadiendo $w = 1$ al **Vector2**, es decir, considerando el **Vector2** como las coordenadas de un punto).

Consultar: docs.godotengine.org/en/stable/classes/class_transform2d.html

Creación de objetos `Transform2D`. Propiedades

Hay varios métodos para crear objetos `Transform2D`. Supongamos que θ es un ángulo en radianes, a un real, x , y , s son `Vector2` (tuplas que representan vectores), $y o, p$ son `Vector2` que representan puntos. Entonces:

- `Transform2D()` crea la matriz identidad.
- `Transform2D(θ , o)` crea la matriz que rota θ radianes entorno a o .
- `Transform2D(x, y, o)` crea una matriz con x , y y o como 1a, 2a y 3a columnas, es decir, damos la base y el origen del marco transformado. Por tanto esto **permite construir cualquier matriz de transformación**.
- `Transform2D(θ , s, a, o)` crea la matriz componiendo una rotación por θ , un escalado por s , una cizalla por a y una traslación por o .

Se pueden usar las *propiedades* `x`, `y` y `origin` para consultar o actualizar los objetos `Vector2` con los valores en la 1a, 2a y 3a columnas, respectivamente. Permite leer o modificar directamente cualquier real de la matriz.

Métodos para componer una matriz 2D con otra

La clase **Transform2D** tiene varios métodos para crear una matriz B y componerla con otra matriz existente A , sin modificar A .

Hay varios métodos que devuelven BA (componen B por la **izquierda**: la acción es A seguida de B). Son:

- $A.\text{rotated}(\theta)$: $B =$ rotación de θ radianes entorno al origen.
- $A.\text{scaled}(s)$: $B =$ matriz de escalado con factores s .
- $A.\text{translated}(t)$: $B =$ traslación por el vector t .

Muchas veces querremos componer las matrices creadas por la **derecha**, es decir obtener AB (acción de B seguida de A). Los correspondientes métodos son:

- $A.\text{rotated_local}(\theta)$
- $A.\text{scaled_local}(s)$
- $A.\text{translated_local}(t)$

Transformaciones en 3D. La clase `Transform3D`

La clase `Transform3D` es similar a la clase `Transform2D`, solo que se guardan 4 columnas de 3 valores reales cada una (12 valores reales en total)

Los constructores, métodos y propiedades son similares, solo que se usan tuplas de tipo `Vector3` en lugar de `Vector2`. Destacamos estos dos constructores:

- `Transform3D()`: crea la matriz identidad.
- `Transform3D(x, y, z, o)`: crea una matriz cuyas columnas son `x`, `y`, `z` y `o` (tipo `Vector3` todos), es decir, damos la base y el origen del marco transformado.

Al igual que en 2D:

- estas matrices se pueden multiplicar entre ellas con `*`, y
- se pueden aplicar a un `Vector3` con `*`, por la derecha (considerando el `Vector3` como las coordenadas de un punto, es decir, añadiendo $w = 1$).

Métodos para componer una matriz 3D con otra

La clase **Transform3D** tiene varios métodos para crear una matriz B y componerla con otra matriz existente A , sin modificar A .

Varios métodos que devuelven BA (componen B por la **izquierda**: la acción es A seguida de B). Son:

- $A.\text{rotated}(\mathbf{e}, \theta)$: $B =$ rotación de θ radianes entorno al eje e (un **Vector3** que debe de tener longitud unidad).
- $A.\text{scaled}(\mathbf{s})$: $B =$ matriz de escalado con factores s (es un **Vector3**)
- $A.\text{translated}(\mathbf{t})$: $B =$ traslación por el vector t (**Vector3**)

Muchas veces querremos componer las matrices creadas por la **derecha**, es decir obtener AB (acción de B seguida de A). Los correspondientes métodos son:

- $A.\text{rotated_local}(\mathbf{e}, \theta)$
- $A.\text{scaled_local}(\mathbf{s})$
- $A.\text{translated_local}(\mathbf{t})$

La transformación de un nodo

Cada nodo de tipo **Node2D** o **Node3D** tiene una propiedad llamada **transform** que es un objeto de la clase **Transform2D** o **Transform3D**, respectivamente. Inicialmente es la matriz identidad.

- En los scripts podemos asignar valores a esa propiedad (en la función **_ready()** o **_init**), con lo cual podemos cambiar la matriz de transformación que se aplica los vértices de ese nodo en tiempo de ejecución.
- En el editor, se pueden cambiar los valores iniciales de **transform** (posición, rotación y escalado, entre otras) en el inspector de propiedades del nodo.

También existen métodos en esas clases que permiten modificar la matriz de transformación de un nodo, sin necesidad de construir una nueva matriz y asignarla a **transform**. Estos métodos se explican más adelante.

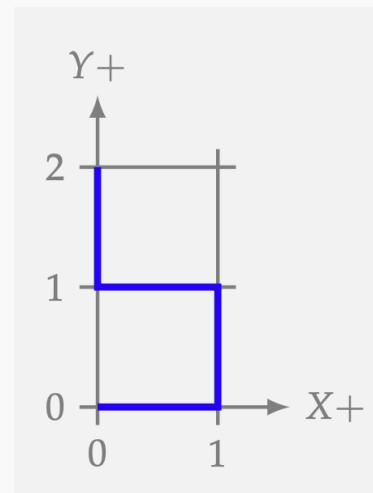
Problemas: implementación de transformaciones 2D en Godot

Para este problema y el siguiente, usa el mismo proyecto de Godot para visualización 2D que ya has usado en problemas previos.

Problema 3.12:

Crea un script global (*autoload*) con una función llamada **gancho** (sin parámetros) que crea y devuelve un objeto de la clase **Mesh** con una polilínea azul como la de la figura (los ejes se han dibujado por claridad).

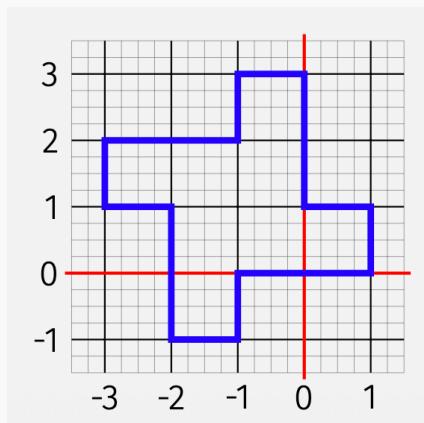
Crea en tu proyecto un nodo 2D de tipo **MeshInstance2D** y en **_ready** asígnale como malla (propiedad **mesh**) el objeto resultado de llamar a **gancho()**, ponle un color azul (propiedad **modulate**) y verifica que el gancho aparece en pantalla al ejecutar el proyecto.



Problemas: implementación de transformaciones 2D en Godot

Problema 3.13:

Crea un nodo 2D de tipo `Node2D` y llámalo `Gancho_x4`. En `_ready`, añádele cuatro nodos hijos de tipo `MeshInstance2D`, cada uno de ellos con un malla creada con la función `gancho` del problema anterior, pero con su `transform` modificada para que el objeto `Gancho_x4` se vea como en la figura (la rejilla y los ejes en rojo se han dibujado por claridad).



Fin de transparencias.

Informática Gráfica.

Sesión 1: Introducción.

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

Índice

Datos de la asignatura.	3
Aplicaciones gráficas interactivas y visualización	26
APIs y motores gráficos.	61

Sección 1.

Datos de la asignatura.

1. La materia.
2. Objetivos, programa, temario.
3. Horarios, datos de contacto, tutorías.
4. Evaluación.
5. Bibliografía y recursos *online*.

Subsección 1.1.
La materia.

Informática Gráfica

La Informática Gráfica es la parte de la Informática que se ocupa del procesamiento de información geométrica y visual. Algunos de los campos más relevantes son:

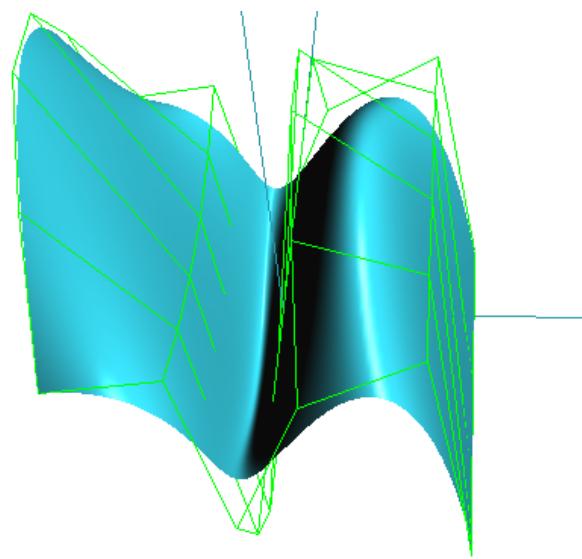
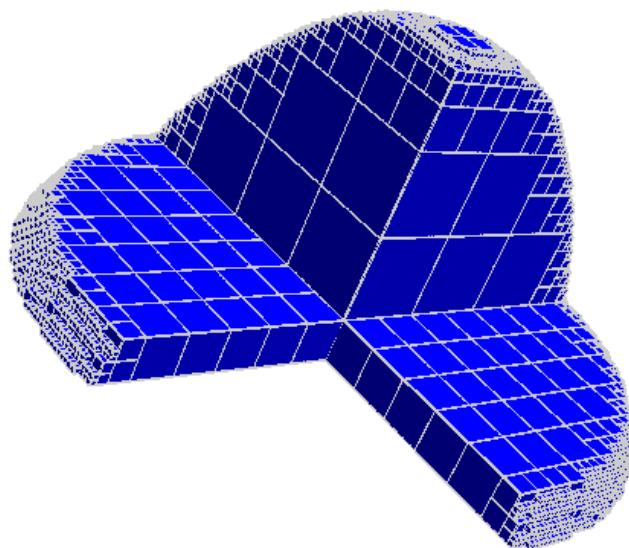
- La representación de información: **modelos geométricos**.
- La generación de imágenes: **visualización (*rendering*)**.
- La entrada de información: **interacción** y **adquisición** de modelos.
- La computación geométrica: **operaciones** y **cálculos** sobre los modelos.

1. Datos de la asignatura..

1.1. La materia..

Modelos geométricos.

Diseño de modelos abstractos de objetos reales, y de las estructuras de datos que se usan para representarlos en la memoria de un ordenador. Creación de los modelos.



1. Datos de la asignatura..

1.1. La materia..

Aplicaciones: Videojuegos, realidad virtual, simuladores



Simulador de Conducción con Editor de Entornos

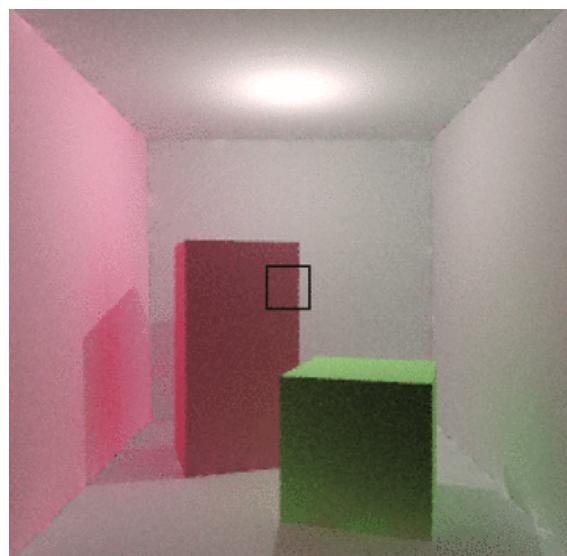
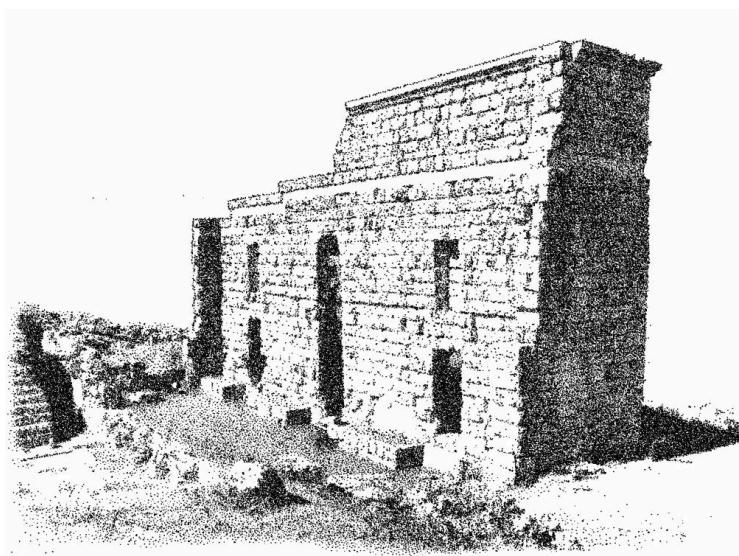
(proyecto fin de carrera de Valerio M. Sevilla, tutor: Carlos Ureña)

1. Datos de la asignatura..

1.1. La materia..

Aplicaciones: visualización (rendering)

Producción de imágenes a partir de modelos geométricos en memoria, no necesariamente de forma interactiva (para cine, anuncios y efectos especiales en general)

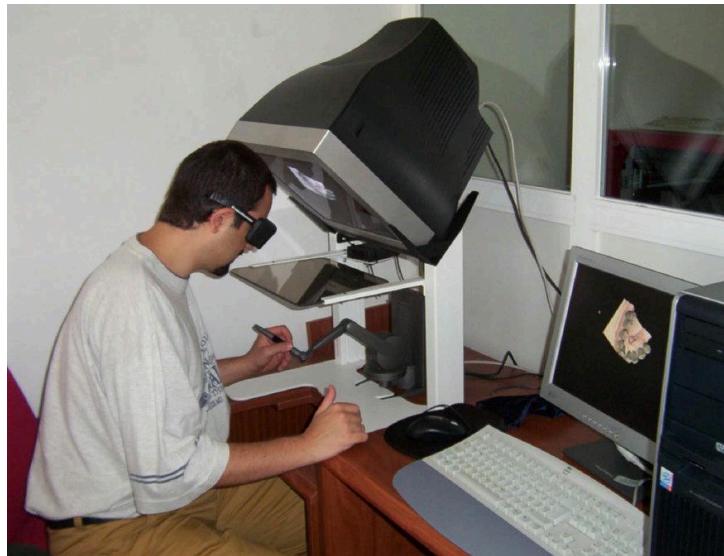


1. Datos de la asignatura..

1.1. La materia..

Aplicaciones: interacción y captura de modelos.

Adquisición de nuevos modelos a partir de objetos reales.

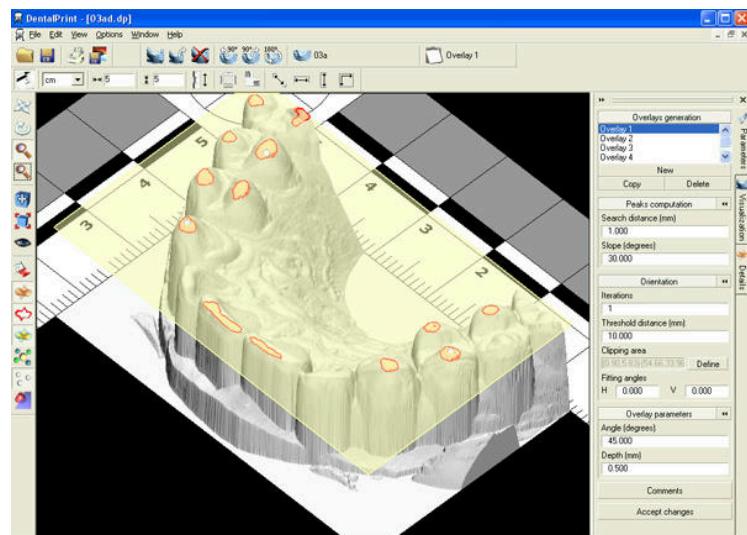
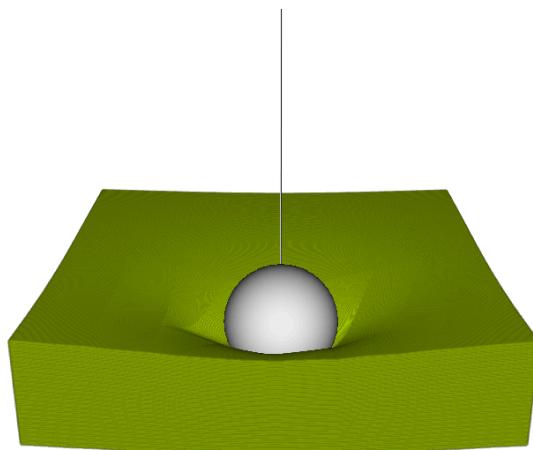


1. Datos de la asignatura..

1.1. La materia..

Aplicaciones: computación geométrica

Algoritmos y metodologías para procesamiento y edición de los modelos.



Subsección 1.2.

Objetivos, programa, temario.

Objetivos de la asignatura

- Conocer los fundamentos del modelado geométrico
- Saber diseñar y utilizar las estructuras de datos más adecuadas para representar un modelo geométrico
- Saber diseñar modelos jerárquicos
- Saber utilizar y representar transformaciones geométricas utilizando coordenadas homogéneas
- Conocer los fundamentos de la visualización 2D y 3D
- Conocer los fundamentos de los modelos de iluminación
- Entender y poder configurar los parámetros de materiales y luces
- Conocer la funcionalidad básica del engine Godot
- Saber diseñar un programa interactivo con eventos, garantizando la accesibilidad y la usabilidad.
- Saber diseñar e implementar programas gráficos interactivos usando Godot
- Conocer los fundamentos de la animación por ordenador

Programa de teoría

El programa de teoría para el curso 2025-26 es el siguiente:

- 1. Introducción:** Introducción a la informática Gráfica. Godot.
- 2. Modelado de objetos:** Fundamentos de modelado. Representación de mallas poligonales. Transformaciones. Instanciación. Modelos jerárquicos. Cálculo de normales. Formatos.
- 3. Visualización:** Cámara y su configuración en Godot. Modelo de iluminación local. Iluminación en Godot. Sombreado para Z-buffer. Visualización de texturas. Texturas en Godot.
- 4. Animación e Interacción:** Sistemas interactivos. Gestión de eventos. Posicionamiento. Selección. Animación. Colisiones.
- 5. Aspectos avanzados de visualización:** Ray-tracing.

Programa de prácticas

El programa de prácticas para el curso 2025-26 es el siguiente:

1. Introducción. Modelado y visualización de objetos 3D sencillos
2. Modelos poligonales: carga de PLYs y generación por revolución.
3. Modelos jerárquicos: creación de un objeto jerárquico.
4. Modelo de aspecto: materiales, fuentes de luz y texturas.
5. Interacción: gestión de cámara y selección.

El código de las prácticas se desarrollará de forma incremental, cada práctica se hace sobre las anteriores.

Subsección 1.3.

Horarios, datos de contacto, tutorías.

Horarios, datos de contacto, tutorías

Clases	Teoría: Martes 9:30-11:30 (aula 1.4) Prácticas: Lunes o Martes 11:30-13-30 (aula 2.1)
Profesor	Carlos Ureña Almagro
Despacho	ETSIIT, planta 3, pasillo izquierdo, desp. 34.
Teléfono	(+34) 958 240 577
E-mail	curena@ugr.es / curena@go.ugr.es
Web	https://www.ugr.es/~curena
Tutorías	Lunes 9:30 - 11:30 — Miércoles 9:30 - 13:30
Info y guía UGR	<u>GIM</u> , <u>GIADE</u> .

Subsección 1.4.
Evaluación.

Evaluación: pruebas de evaluación.

En la convocatoria ordinaria, en evaluación continua se harán las siguientes pruebas:

(E1) Examen de teoría: escrito, en la fecha establecida por el centro, con un peso de 50// en la nota final.

(E2) Examen de prácticas: se establecerán una o varias fechas para la entrega y examen de prácticas, el examen consistirá en resolver problemas de programación usando el código entregado. Suponen el 50// en la nota final (10// por práctica).

En **convocatoria extraordinaria** y en **evaluación única final** se seguirán los mismos criterios excepto que las pruebas de prácticas (E2) se realizarán en una fecha única dentro del período de exámenes correspondiente.

Evaluación: calificación.

- Se podrá sumar hasta un 10% de la nota máxima por trabajos adicionales, re-lización de ejercicios o presentaciones, mejora de las prácticas, etc., siempre que se haga de forma previamente acordada con el profesor y siempre que se supere la asignatura con el resto de items evaluables (E1 y E2).
- Para aprobar la asignatura hay que obtener igual o más del 50% del total, e igual o más del 40% en cada parte (E1 y E2).
- Si un alumno no supera la asignatura en la convocatoria ordinaria del curso 25-26, pero tiene una nota igual o superior al 50% en algunas de las partes (E1 o E2), entonces podrá conservar esa nota para la convocatoria extraordinaria de 25-26.

Subsección 1.5.

Bibliografía y recursos *online*.

Bibliografía: Conceptos de Informática Gráfica (1/2)

J.F. Hughes, A.van Dam, M. McGuire, D.F. Sklar, J.D. Foley, S.K. Feiner, K. Akeley.

Computer Graphics: Principles and Practice (3ed ed.)

Ed. Pearson, 2014. ISBN: 978-0-321-39952-6.

Web autores: <http://cgpp.net>

Web editorial: <https://www.pearson.com/....>

Steve Marschner, Peter Shirley.

Fundamentals of computer graphics. (5th ed.)

Ed. A.K. Peters / CRC Press, 2021. ISBN: 978-1032122861

Web autores: <https://www.cs.cornell.edu/~srm/fcg5/>

Web editorial (DOI): <https://doi.org/10.1201/9781003050339>

Bibliografía: Conceptos de Informática Gráfica (2/2)

Steven J. Gortler.

Foundations of 3D Computer Graphics (1st ed.).

Ed. The MIT Press, 2012. ISBN: 9780262017350.

Web autores: <http://www.3dgraphicsfoundations.com/>

Web editorial: <https://mitpress.mit.edu/.....>

Tomas Akenine-Möller, Eric Haines, Naty Hoffman.

Real-Time Rendering (4th ed.)

Ed. CRC Press, 2018. ISBN: 978-1138627000

Web autores (recursos): <https://www.realtimerendering.com>

Bibliografía: Matemáticas para gráficos

Eric Lengyel.

Mathematics for 3D Game Programming and Computer Graphics (3rd ed.).

Ed. Cengage Learning, 2011. ISBN: 978-1-4354-5886-4

Web autores: <http://mathfor3dgameprogramming.com/>

Michael E. Mortenson.

Mathematics for Computer Graphics Applications (2nd ed.).

Ed. Industrial Press, 1999. ISBN: 0-8311-3111-X.

Web editorial:

<https://books.industrialpress.com/9780831131111/mathematics-for-computer-graphics-applications/>

Bibliografía: Godot / GDScript.

Ivan Korotkin.

Godot 4: Introduction to GDScript. Autoeditado (2023). ISBN: 979-8394581557

Amazon:

https://www.amazon.com/Godot-Introduction-GDScript-practical-tutorials/dp/B0C5241244#detailBullets_feature_div

Sander Vanhove.

Learning GDScript by Developing a Game with Godot 4.

Ed. Packt Publishing 2024. ISBN: 9781801812498.

Web editorial:

<https://www.packtpub.com/en-us/product/learning-gdscript-by-developing-a-game-with-godot-4-9781801812498>

Recursos online: Godot / GDScript

Enlaces a las versiones en español, traducidas (algunas parcialmente) a partir de los originales en inglés:

Web con la documentación oficial de Godot 4:

<https://docs.godotengine.org/es/4.x/>

Introducción a Godot:

https://docs.godotengine.org/es/stable/getting_started/introduction/index.html

Guía para crear un juego 3D con Godot:

https://docs.godotengine.org/es/4.3/getting_started/first_3d_game/index.html

Referencia de las clases de Godot:

<https://docs.godotengine.org/es/4.x/classes/index.html>

Sección 2.

Aplicaciones gráficas interactivas y visualización

1. Aplicaciones gráficas interactivas
2. El proceso de visualización 2D y 3D
3. *Rasterización versus ray-tracing.*
4. El cauce gráfico en rasterización

Resumen de la sección

En esta sección se incluye:

- Una introducción a las aplicaciones gráficas interactivas,
- Los dos métodos básicos de visualización (Rasterización y por Ray-tracing).
- Se da una visión muy general sobre el procesamiento que se hace en el cauce gráfico.

Subsección 2.1.

Aplicaciones gráficas interactivas

Aplicaciones gráficas

Un **programa gráfico** es un programa (o parte de un programa o sistema) que

- Almacena una estructura de datos que constituye un **modelo** computacional de determinada información.
- Produce una salida constituida (principalmente) por una o varias imágenes.
- Las imágenes típicas son **imágenes raster**, constituidas por un array de pixels discretos, cada uno con un color RGB.
- Existen otros tipos de salidas gráficas, la más frecuentes son las **imágenes vectoriales** (p.ej.: archivos **.svg**).
- Los programas gráficos pueden ser: **interactivos** o **no interactivos**

Aplicaciones gráficas interactivas (1/2)

Un programa gráfico **interactivo** es un programa que:

- **Visualiza** en una ventana gráfica una imagen que constituye una representación visual del modelo.
- Procesa acciones del usuario (llamadas **eventos**), que se traducen en modificaciones del modelo.
- Cada vez que el modelo es modificado, se vuelve a visualizar, de forma **interactiva**, lo que significa que desde que el usuario produce el evento hasta que puede observar la imagen actualizada pasan tiempos del orden de decenas de milisegundos como mucho.

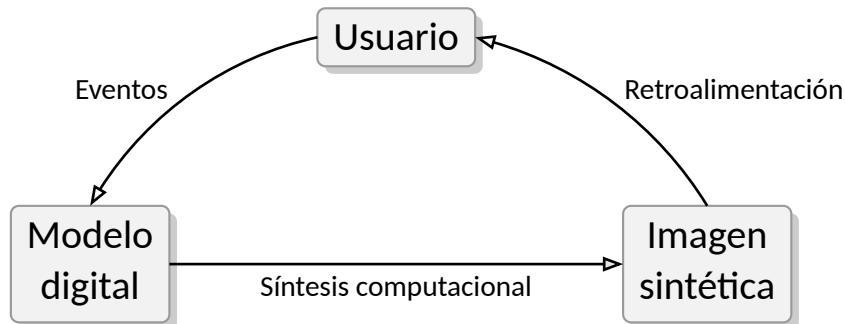
Este esquema es el que se usa típicamente en aplicaciones de simuladores, diseño asistido por computador, videojuegos, realidad virtual y realidad aumentada.

Aplicaciones gráficas interactivas (2/2)

Los elementos esenciales de una aplicación gráfica interactiva son:

- **Modelos digitales** de objetos reales, ficticios o de datos
- **Imágenes o videos digitales** que se usan para visualizar dichos objetos.

En las aplicaciones interactivas, los usuarios modifican los modelos (con eventos) y reciben retroalimentación inmediata:



Eventos de entrada

Los **eventos de entrada** son acciones del usuario que permiten enviar información a la aplicación, información que se usa para modificar el modelo o los parámetros de visualización. Ejemplos:

- Pulsar o levantar una tecla del teclado.
- Pulsar o levantar un botón del ratón.
- Mover el ratón.
- Mover la rueda del ratón.
- Cambiar el tamaño de una ventana de la aplicación.
- Cerrar una ventana de la aplicación.

También se pueden considerar eventos otros tipos de sucesos, no directamente iniciados por el usuario, como la llegada de datos por la red, la finalización de una tarea de cálculo, o el transcurso de un período de tiempo (para animaciones).

El bucle principal de gestión de eventos

Las aplicaciones gráficas interactivas ejecutan (explícitamente o implícitamente) un bucle, el llamado **bucle principal de gestión de eventos**. Es un bucle que se ejecuta continuamente, y que en cada iteración da estos pasos:

1. Espera a que ocurra un evento y entonces recuperar datos del mismo.
2. Procesar el evento: típicamente supone actualizar:
 - el modelo y/o
 - los parámetros de visualización.
3. Visualizar el modelo actualizado con los nuevos parámetros.

Subsección 2.2.

El proceso de visualización 2D y 3D

Visualización 2D y 3D (1/2)

En general, las aplicaciones gráficas pueden, en general, dividirse en dos tipos:

- **Aplicaciones 2D:** usan modelos de objetos visibles o dibujables que se definen en un plano (espacio bidimensional), o en varios planos (unos con más *prioridad* que otros, es decir, por delante de otros).
 - ▶ Pueden incluir también algunos elementos 3D, como sombras.
 - ▶ Ejemplos: aplicaciones de visualización de datos, edición de imágenes, diseño gráfico 2D.

Visualización 2D y 3D (2/2)

En general, las aplicaciones gráficas pueden dividirse en dos tipos:

- **Aplicaciones 3D:** se usan modelos de objetos visibles o dibujables que se definen en el espacio tridimensional. Dichos modelos incluyen información de aspecto como texturas, materiales, fuentes de luz, etc...que son propias de la visualización 3D.
 - ▶ Pueden incluir también elementos 2D, como texto, iconos, etc...
 - ▶ Ejemplos: videojuegos, simuladores, realidad virtual, realidad aumentada.

Visualización 2D. Entradas.

El proceso de visualización 2D produce una imagen a partir de un **modelo** y unos **parámetros**:

- **Modelo:** estructura de datos en memoria que representa los elementos que se van a visualizar en la imagen, suele incluir puntos, líneas, polígonos, textos, imágenes, gradaciones, etc...Estos elementos se construyen a partir de los datos de entrada en función del estilo de visualización que se deseé.
- **Parámetros:** diversos valores que afectan a la visualización: la resolución de la imagen y su posición en pantalla (viewport), la zona de coordenadas de mundo que se quiere visualizar (un rectángulo del espacio de coordenadas de mundo), el rango de valores en Z, etc....

2. Aplicaciones gráficas interactivas y visualización.

2.2. El proceso de visualización 2D y 3D.

Visualización 2D. Visualización de datos.

Las imágenes ayudan a entender la información en tablas numéricas, árboles, DAGs, grafos arbitrarios, relaciones, etc...

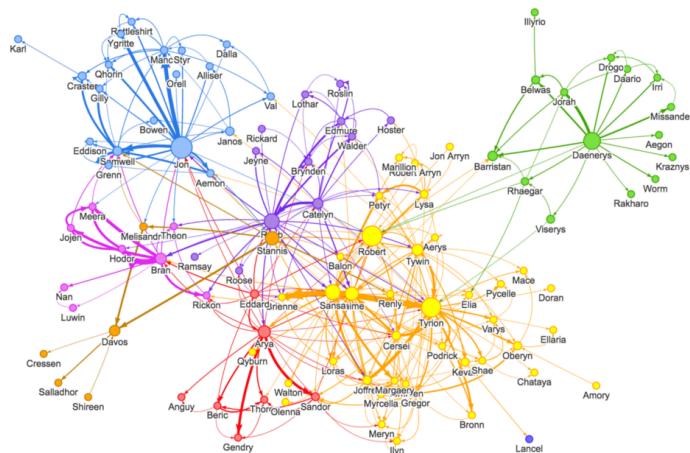


Imagen de *Hands on graph data visualization* (Relaciones entre personajes de *Game of Thrones*)

<https://neo4j.com/developer-blog/hands-on-graph-data-visualization/>

El proceso de visualización 3D: entradas (1/2)

El proceso de visualización 3D produce una imagen a partir de un **modelo de escena** y unos **parámetros**:

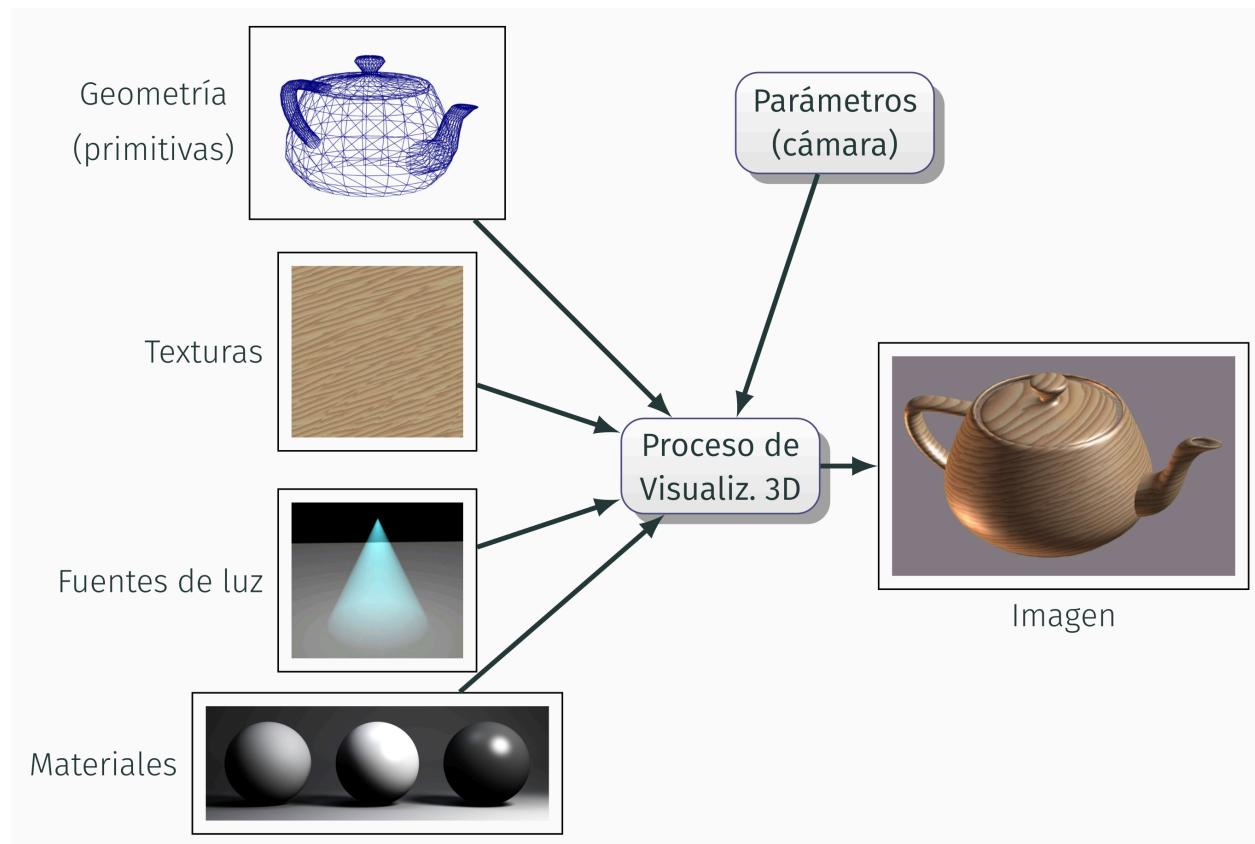
- **Modelo de escena:** estructura de datos en memoria que representa lo que se quiere ver, esta formado por varias partes:
 - ▶ **Modelo geométrico:** conjunto de **primitivas** (típicamente polígonos planos), que definen la forma de los objetos a visualizar
 - ▶ **Modelo de aspecto:** conjunto de parámetros que definen el aspecto de los objetos: tipo de material, color, texturas, fuentes de luz

El proceso de visualización 3D: entradas (2/2)

El proceso de visualización produce una imagen a partir de un **modelo de escena** y unos **parámetros**:

- **Parámetros de visualización:** es un conjunto amplio de valores que determinan como se visualiza la escena en la imagen, algunos elementos esenciales son:
 - ▶ **Cámara virtual:** posición, orientación y ángulo de visión del observador ficticio que vería la escena como aparece en la imagen
 - ▶ **Viewport:** Resolución de la imagen, y, si procede, posición de la misma en la ventana.

El proceso de visualización 3D: esquema



Subsección 2.3.

Rasterización versus ray-tracing.

Visualización basada en rasterización (1/2)

En este curso nos centramos en los algoritmos relacionados con la visualización basada en **rasterización** (*rasterization*). En ese algoritmo, se produce como salida una imagen I a partir de un conjunto de *primitivas* de entrada E . En pseudocódigo:

```
1: Inicializar el color de todos los pixels al color de fondo.  
2: for cada primitiva  $P$  del conjunto  $E$  do  
3:    $S \leftarrow$  conjunto de pixels de la imagen  $I$  cubiertos por  $P$   
4:   for cada pixel  $q$  de  $S$  do  
5:      $c \leftarrow$  color de la primitiva  $P$  en el pixel  $q$   
6:     Asignar el color  $c$  al pixel  $q$  en  $I$   
7:   end  
8: end
```

Visualización basada en rasterización (2/2)

En el código anterior:

- Las **primitivas** son los elementos más pequeños que pueden ser visualizados (típicamente triángulos en 3D, aunque también pueden ser otros: polígonos, puntos, segmentos de recta, círculos, etc...)
- El número de pixels en S es proporcional al número p de pixels en I
- El tiempo de cálculo total es proporcional al número de iteraciones del bucle interno, ese número es a su vez proporcional al número de primitivas (n) por el número de pixels (p).
- Es decir, decimos que el tiempo de cálculo *está en el orden de* $p \cdot n$, o lo que es lo mismo: el tiempo esta en $O(p \cdot n)$.

Si se considera p una constante, el tiempo está en $O(n)$.

Visualización basada en ray-tracing (1/2)

En la técnica de **Ray-Tracing**, los dos bucles de antes se intercambian:

- 1: Inicializar el color de todos los pixels
- 2: **for** cada pixel q de la imagen I a producir **do**
- 3: $T \leftarrow$ subconjunto de primitivas de E que cubren q
- 4: **for** cada primitiva P del conjunto T **do**
- 5: $c \leftarrow$ color de la primitiva P en el pixel q
- 6: Asignar color c al pixel q en I
- 7: **end**
- 8: **end**

- Cuando se trata de visualización 3D, la implementación de esta esquema se conoce como algoritmo de **Ray-tracing**.
- Al igual que en rasterización, el tiempo de cálculo total es proporcional al número de de primitivas (n) por el número de pixels (p), es decir, la complejidad en tiempo está también en $O(n \cdot p)$.

Visualización basada en ray-tracing (2/2)

El algoritmo de Ray-tracing se puede optimizar para lograr que el cálculo de T en la línea 3 sea muy eficiente:

- Esto requiere el uso de **indexación espacial**: las primitivas se organizan en una estructura de datos que permite encontrar rápidamente las primitivas que cubren un pixel dado.
- Al usar esta optimización, la complejidad en tiempo será proporcional al logaritmo natural del número de primitivas ($\log n$), en lugar de a n , es decir, el tiempo de cálculo estará ahora en el orden de $O(p \cdot \log n)$.

Si se considera p una constante el tiempo está en $O(\log n)$

Rasterización

Respecto a la técnica de *rasterización*:

- Las **unidades de procesamiento gráfico** (GPUs) son un hardware diseñado originalmente para ejecutar la rasterización de forma eficiente en tiempo.
- El método de rasterización es preferible para **aplicaciones interactivas**, y es el que se usa principalmente en la actualidad para **videojuegos, realidad virtual y simulación**, asistido por GPUs.
- La totalidad de las aplicaciones gráficas en dispositivos móviles y tabletas usan rasterización.

En este curso nos centramos en los pasos de cálculo necesarios para la rasterización 2D y 3D, y veremos ejemplos prácticos en el contexto del *game engine* Godot.

Ray-tracing

Respecto de la técnica de *Ray-tracing*:

- El método de Ray-tracing y sus variantes suele ser **más lento**, pero consigue resultados **más realistas** cuando se pretende reproducir ciertos efectos visuales.
- Las variantes y extensiones de Ray-tracing son preferibles para síntesis de imágenes *off-line* (no interactivas), y es el que se usa principalmente en la actualidad para **producción de animaciones y efectos especiales** en películas o anuncios.
- En los últimos años (2018 en adelante) han aparecido arquitecturas de GPUs con aceleración por hardware para Ray-Tracing, lo que está llevando a **implementar algunos videojuegos** usando Ray-Tracing (pero requieren GPUs muy potentes).

En este curso veremos algo de Ray-tracing.

Subsección 2.4.

El cauce gráfico en rasterización

El cauce gráfico en rasterización: entradas y salidas

El **cauce gráfico** es el conjunto de etapas de cálculo que permiten la síntesis de imágenes por rasterización:

- Las entradas al cauce gráfico se denominan **primitivas**, una primitiva es una forma visible que no se puede descomponer en otros más simples, en rasterización típicamente las primitivas son: triángulos, segmentos de líneas o puntos (en 2D o en 3D)
- Un **vértice** es un punto del espacio 2D o 3D, extremo de una arista de un triángulo, o de un segmento de recta, o donde se dibuja un punto. Una o varias primitivas se especifican mediante una **lista de coordenadas de vértices**, más alguna información adicional.
- El cauce escribe en el **framebuffer**, que es una zona de memoria donde se guardan uno o varios arrays con los colores RGB de los pixels de la imagen (y alguna información adicional). Está conectado al monitor.

- 2. Aplicaciones gráficas interactivas y visualización.
- 2.4. El cauce gráfico en rasterización.

Transformación y sombreado

Hay (entre otros) dos pasos importantes del cauce gráfico que se ejecutan en la GPU:

1. Transformación:

En esta etapa se parte de las coordenadas de un vértice que se especifican en la aplicación, y se calculan las coordenadas (normalizadas) de su proyección en la ventana.

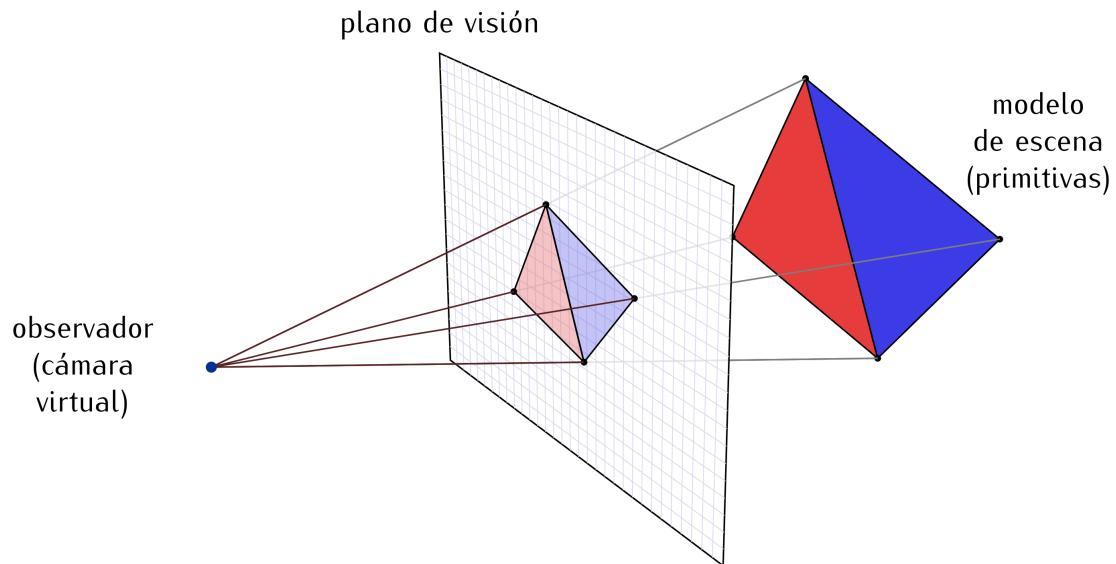
2. Sombreado:

El cálculo del color de un pixel (una vez que se ha determinado que una primitiva se proyecta en dicho pixel). Por lo visto hasta ahora, esto se hace simplemente asignando un color prefijado al pixel (pero es usualmente más complicado).

Además de estas etapas, hay otras como la rasterización y el recortado de polígonos.

Transformación y proyección

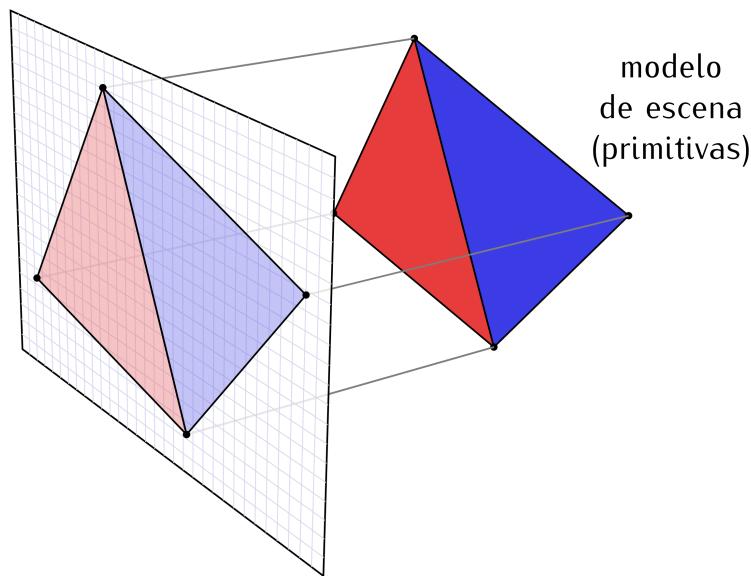
Cada primitiva se sitúan en su lugar en el espacio, y se encuentra su proyección en un plano imaginario (**plano de visión, viewplane**) situado entre el **observador** y la escena (las primitivas):



Proyección paralela

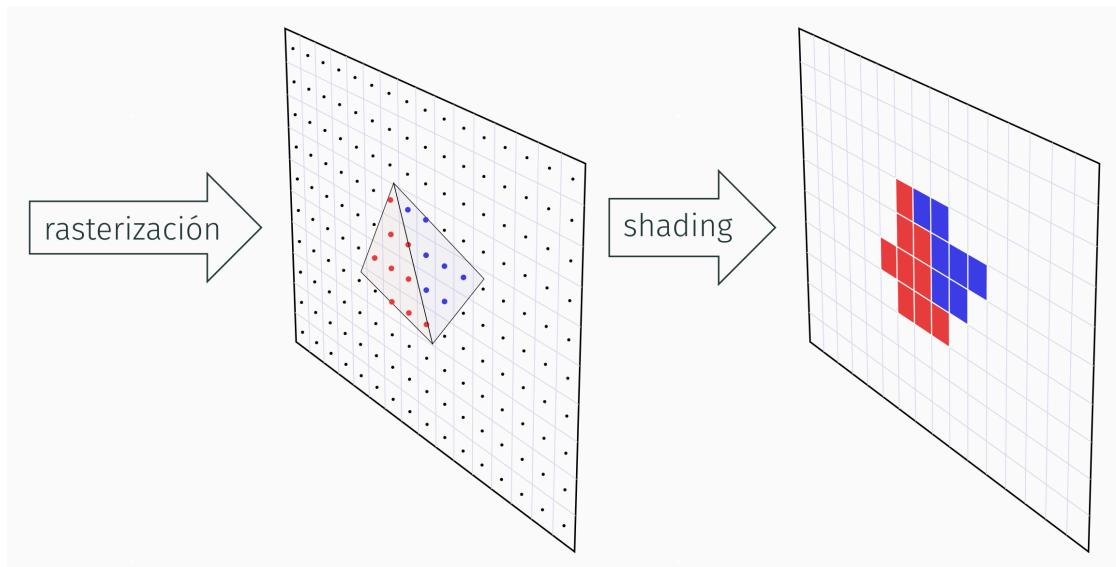
La proyección puede ser **perspectiva** (como en la transparencia anterior), o **paralela**, como aparece aquí:

plano de visión



Rasterización y sombreado

- **Rasterización:** para cada primitiva, se calcula que pixels tienen su centro cubierto por ella.
- **Sombreado:** (*shading*) se usan los atributos de la primitiva para asignar color a cada pixel que cubre.



Sombreado: básico versus avanzado

El proceso de sombreado puede simplemente asignar un color *plano* a cada polígono (izquierda) o bien incluir cálculos avanzados con *iluminación* y *texturas* (derecha)



Etapas del cauce gráfico (1/2)

Con más detalle, el cauce gráfico tiene estas etapas:

1: Procesado de vértices: parte de una secuencia de vértices (puntos del espacio) y produce una secuencia de primitivas (puntos, segmentos o triángulos). Tiene estas sub-etapas:

1.1: Transformación: los vértices de cada **primitiva** son transformados en diversos pasos hasta encontrar su proyección en el plano de la imagen. Es realizado por un sub-programa llamado **Vertex Shader** (modificable por el programador, o *programable*).

1.2: Teselación y nivel de detalle: transformaciones adicionales avanzadas, realizadas por varios programas, entre ellos el **tesselation shader** y el **geometry shader** (programables).

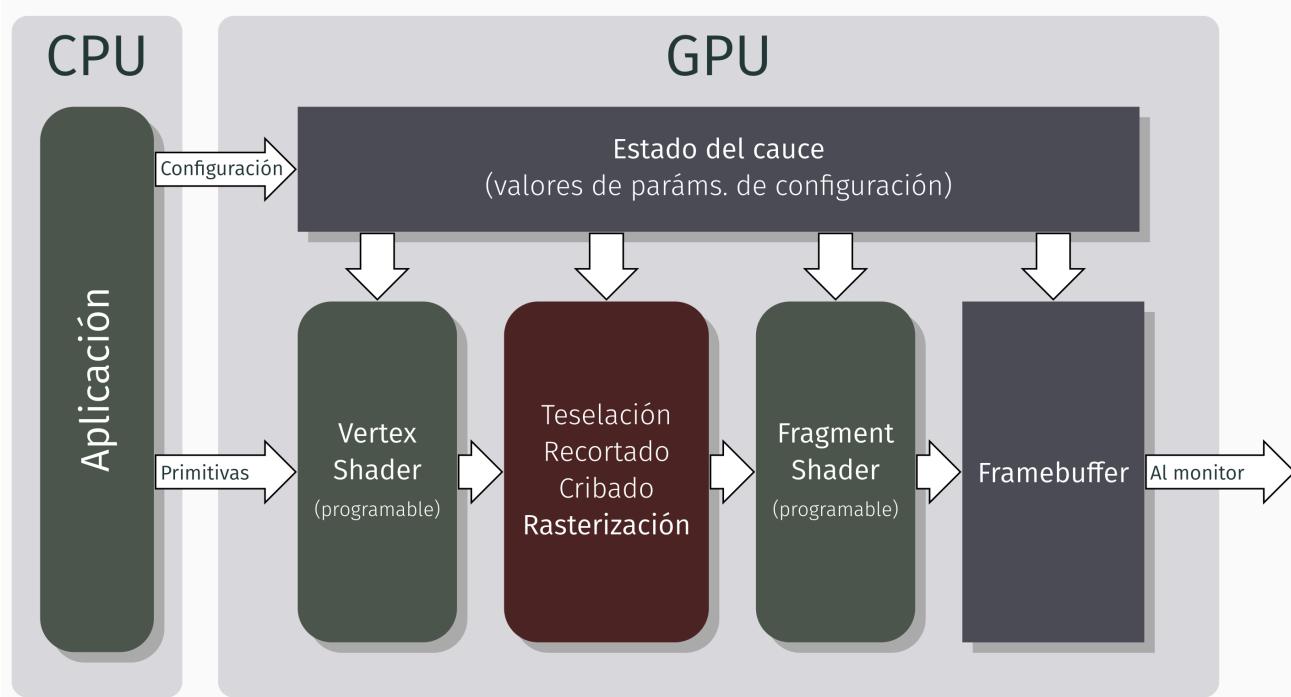
Etapas del cauce gráfico (2/2)

A continuación ocurren estas etapas:

2. **Post-procesado de vértices y montaje de primitivas** incluye varios cálculos como el *recortado* (*clipping*) y el *cribado de caras*} (*face culling*}), ninguno de ellos programable.
3. **Rasterización** (*rasterization*) cada primitiva es *rasterizada* (discretizada), y se encuentran los pixels que cubre en la imagen de salida (no es programable).
4. **Sombreado** (*shading*): en cada pixel cubierto se calcula el color que se le debe asignar. Se realiza por un programa llamado **fragment shader** o **pixel shader** (programable).

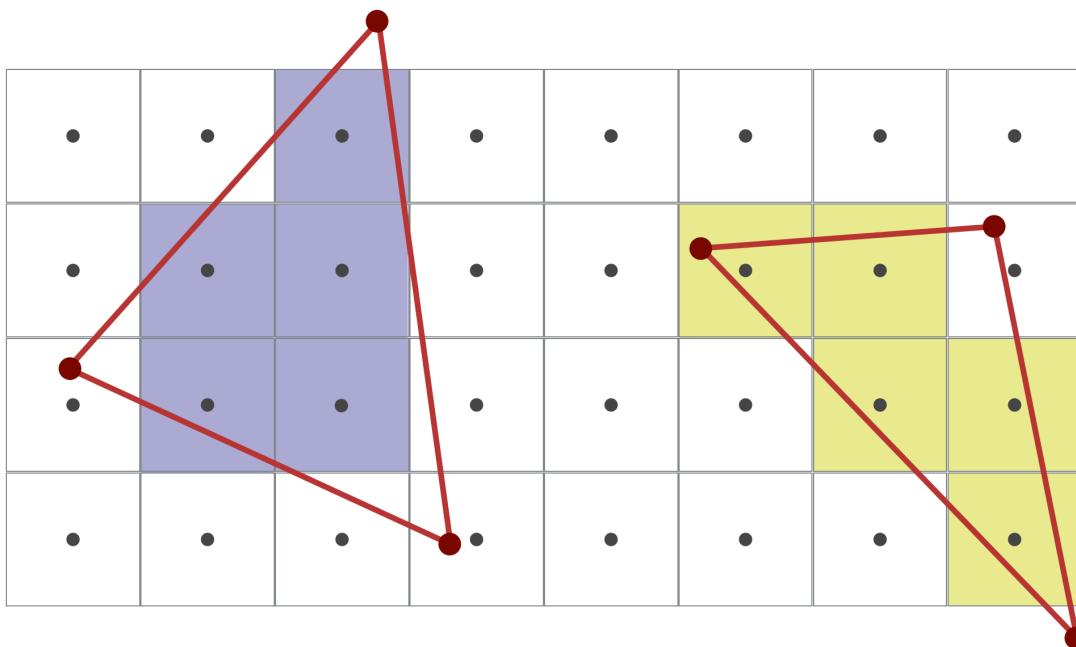
Esquema del cauce gráfico en una GPU

Este diagrama de flujo de datos refleja las etapas:



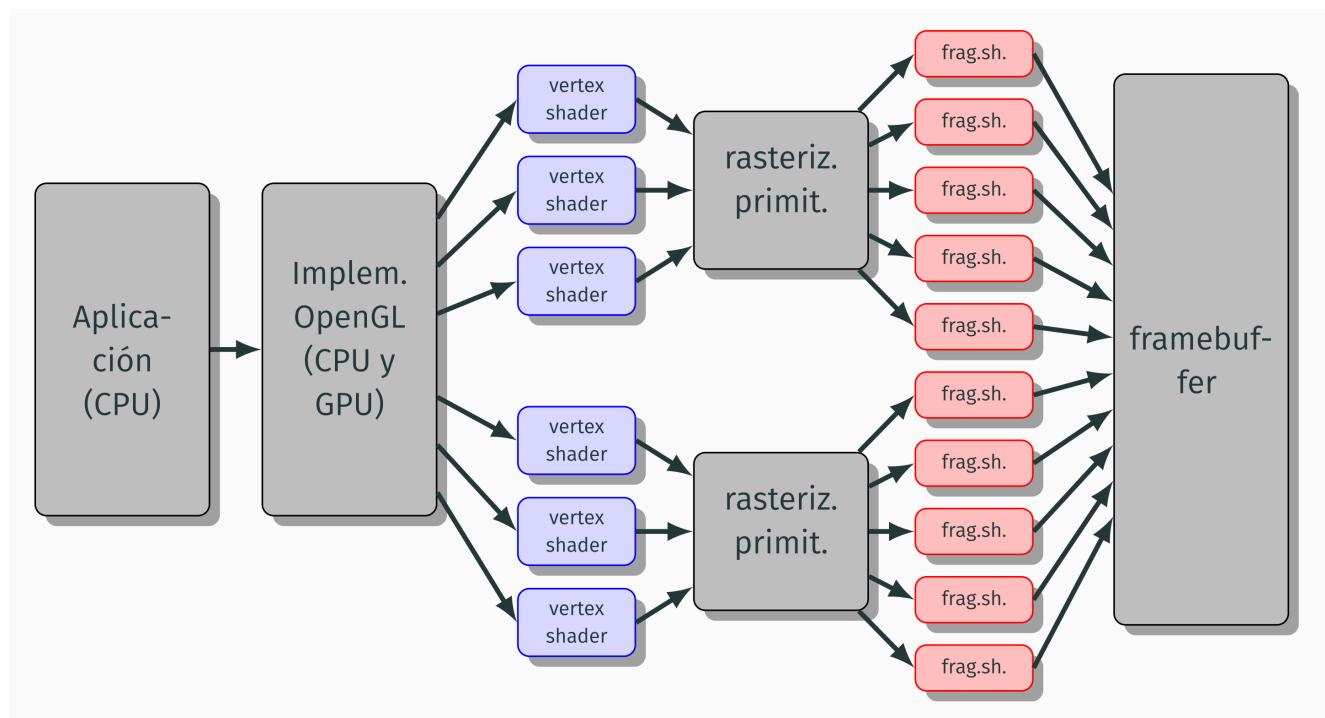
Ejemplo: rasterización de dos triángulos

La rasterización consiste en calcular los pixels cuyo centro está cubierto por la proyección de cada primitiva:



Paralelismo potencial en la GPU

En este grafo de dependencia vemos como se pueden ejecutar concurrentemente las ejecuciones de los distintos shaders para el ejemplo anterior:



Sección 3.

APIs y motores gráficos.

1. APIs para Rasterización, Ray-tracing y GPGPU
2. Motores gráficos

Subsección 3.1.

APIs para Rasterización, Ray-tracing y GPGPU

APIs de rasterización

Una **API de rasterización** es la **especificación** de un conjunto de funciones y/o clases útiles para visualización 2D/3D basada en rasterización, es decir un documento con descripción de funciones (y sus parámetros), clases, interfaces, etc... junto con el comportamiento esperado de estas funciones y clases.

- Estas APIs son definidas por organizaciones sin ánimo de lucro (consorcios) o empresas privadas.
- Permiten la rasterización de primitivas de bajo nivel (polígonos, líneas segmentos), de forma eficiente y portable.
- La API OpenGL fue pionera en la rasterización en GPUs.
- Las APIs usan un *Shading Language*: un lenguaje de programación específico para los programas que se ejecutan en la GPU (*shaders*).

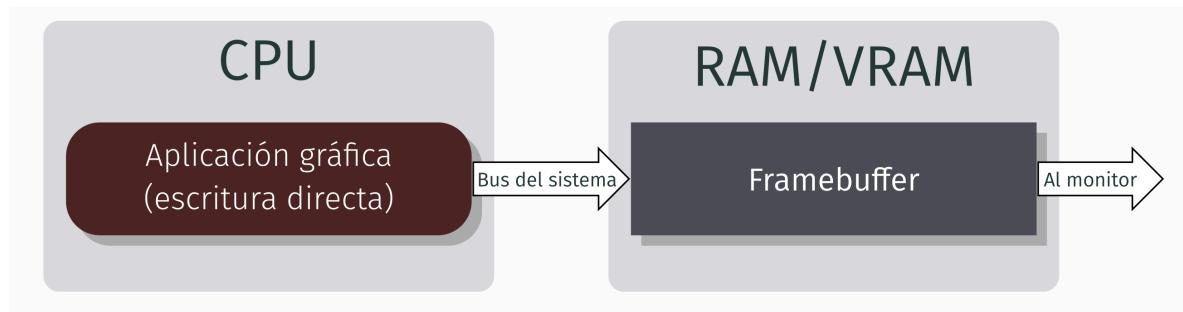
Evolución de la funcionalidad en rasterización

A lo largo de los años y hasta la actualidad, se incrementa la funcionalidad y programabilidad de las GPUs para rasterización, así como las plataformas soportadas:

- Se van incorporando lenguajes de *shading* de alto nivel estandarizados, como GLSL (Khronos), Cg (nVidia), HLSL (Microsoft), MSL (Apple).
- Se definen formatos de código intermedio para lenguajes de shading, como SPIR-V.
- Se pueden programar más etapas del cauce: *tesselation shaders*, *geometry shaders*, *mesh shaders*, etc...
- Partiendo de las *workstations* de finales de los 80, se incorporan GPUs programables en toda clase de dispositivos: ordenadores de sobremesa, ordenadores portátiles, tabletas, móviles y relojes.

Aplicaciones de escritura directa

Inicialmente (años 70-80), las aplicaciones gráficas escribían directamente en la memoria de vídeo (VRAM)

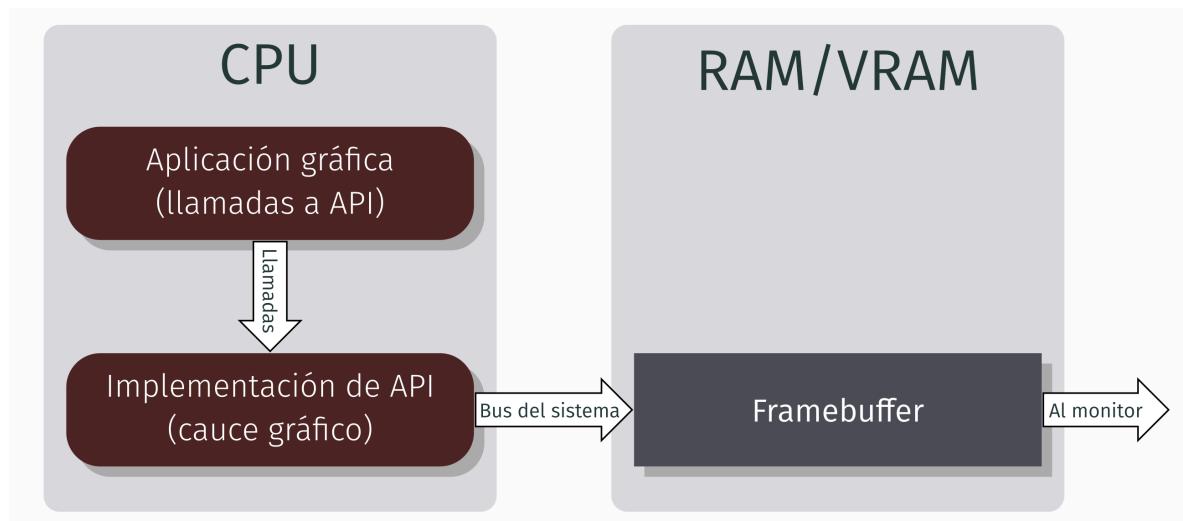


Desventajas

- La escritura en el framebuffer a través del bus del sistema es lenta, y se realiza pixel a pixel.
- Solución no portable entre arquitecturas hardware o software.
- Una aplicación gráfica no puede coexistir con otras

Uso de APIs gráficas

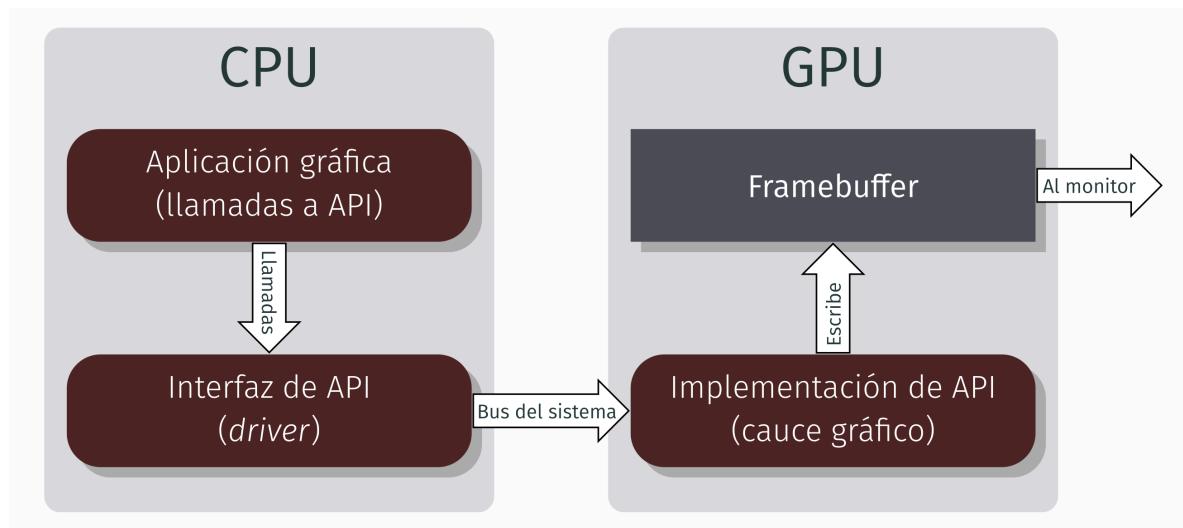
El uso de implementaciones de APIs gráficas portables (y gestores de ventanas) proporciona **portabilidad** y **acceso simultáneo**



- La escritura en el *framebuffer* a través del bus del sistema sigue siendo lenta.

Uso de APIs y hardware gráfico (GPUs)

El uso de **GPUs aumenta la eficiencia** ya que ejecutan el cauce y y reducen el tráfico a través del bus del sistema (se envía menos información de más alto nivel).



Plataformas hard./soft.: OpenGL, OpenGL ES y Vulkan



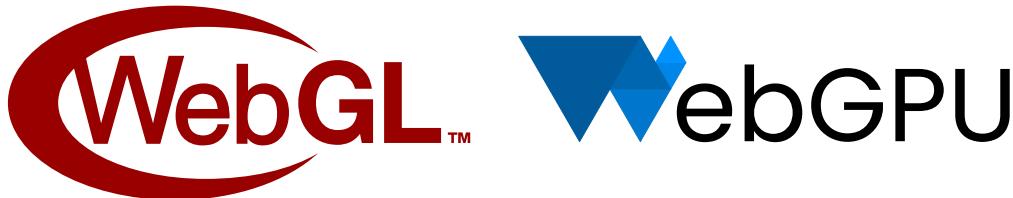
- **OpenGL** (www.opengl.org) se puede usar en aplicaciones que se ejecutan en:
 - ▶ Linux y Windows, usando implementaciones (drivers) de los fabricantes de GPUs y macOS (nVidia, AMD, Intel)
 - ▶ macOS, con una implementación de Apple, que la considerada *obsoleta* (*deprecated*), pero funcional.
- **OpenGL ES** se puede usar en dispositivos Android, PS3 y PS4
- **Vulkan** (www.vulkan.org): se puede usar en aplicaciones que se ejecutan en:
 - ▶ Linux y Windows (con drivers de nVidia, AMD o Intel)
 - ▶ macOS, mediante MoltenVK de Khronos: traduce llamadas de Vulkan 1.2 a llamadas de Metal.
 - ▶ Dispositivos móviles y consolas: Android, Nintendo Switch.

Plataformas hardware y software: DirectX y Metal



- DirectX se ha implementado por Microsoft para diversas plataformas:
 - ▶ Windows para ordenadores.
 - ▶ Consolas de videojuegos XBox.
- Metal se ha implementado por Apple para sus distintos sistemas operativos sobre distintos tipos de dispositivos:
 - ▶ macOS en todos los ordenadores de Apple.
 - ▶ iOS en móviles iPhone.
 - ▶ iPadOS en dispositivos iPad.
 - ▶ tvOS en dispositivos Apple TV.
 - ▶ visionOS para dispositivos Vision PRO.

APIs para programación del cauce en la Web



- **WebGL** (www.khronos.org/webgl/): soportado por todos los navegadores, diseñado por Mozilla foundation. Se han publicado dos versiones:
 - ▶ **WebGL 1:** lanzado en 2011, basado en OpenGL ES 2.0. Disponible en la inmensa mayoría de ordenadores y dispositivos móviles.
 - ▶ **WebGL 2:** lanzado en 2017, basado en OpenGL ES 3.0. Disponible en ordenadores y dispositivos móviles modernos (algunos dispositivos móviles de gama baja o antiguos podrían no soportarlo).
- **WebGPU** (www.w3.org/TR/webgpu/): basado en Vulkan, diseñado por W3C. Soportado (desde mediados de 2023) exclusivamente en las versiones para ordenadores de los navegadores.

APIs modernas frente a tradicionales

Llamamos *APIs tradicionales* a: OpenGL, OpenGL ES, Direct X (hasta versión 11 incluida) y WebGL, y *APIs modernas* a Vulkan, Metal, DirectX (versión 12) y WebGPU.

Las APIs modernas son más eficientes:

- **Bajo nivel:** permiten un control más fino sobre el hardware
- **Multihebra:** aprovechan múltiples núcleos de CPU a la vez.
- **Menor sobrecarga:** reducen la sobrecarga de la CPU y el consumo de memoria.

Aunque también presentan desventajas:

- **Mayor complejidad:** requieren más código (y por tanto mayor tiempo de desarrollo) en comparación con las tradicionales.
- **Menor portabilidad:** requieren más esfuerzo para portar aplicaciones a distintas plataformas, y en algunos casos no se podrá mantener la eficiencia o la funcionalidad sin un esfuerzo importante de desarrollo.

Histórial de versiones de APIs (tradicionales).

Año	OpenGL	DirectX	OpenGL ES	WebGL
1992	1.0			
1995	1.5	1.0		
1996-2000		2.0 al 8.0		
2003		9.0	1.0	
2004	2.0		1.1	
2006	2.1	10.0		
2007			2.0	
2008	3.0			
2009	3.1 - 3.2	11.0		
2010	3.3 - 4.0 - 4.1			
2011	4.2			1.0
2012	4.3	11.1	3.0	
2013	4.4	11.2		
2014	4.5		3.1	
2015			3.2	
2017	4.6			2.0

Histórial de versiones de APIs (modernas).

A partir de 2017 no hay nuevas versiones de OpenGL ni OpenGL ES (se sustituye por Vulkan) ni WebGL (se sustituye por WebGPU).

Las APIs modernas se inician con Metal en 2014, este es el histórial de versiones:

Año	Metal	DirectX 12	Vulkan	WebGPU
2014	1 (iOS)			
2014	1 (macOS)			
2015		12.0		
2016			1.0	
2017	2			
2018		12.1	1.1	
2020		12.2	1.2	
2022	3		1.3	
2023				1.0

Evolución de la funcionalidad: soporte para Ray-Tracing

Además del cauce gráfico en rasterización, las GPUs se usan en la actualidad para acelerar otros tipos de algoritmos, tanto dentro como fuera del ámbito de los gráficos:

- En gráficos: se incorpora soporte hardware y software para acelerar Ray-Tracing (de 2018 en adelante), a través de nuevas APIs o de extensiones de las APIs existentes:
 - ▶ **OptiX** (nVidia, 2009),
 - ▶ **Vulkan Ray-Tracing** (Khronos, 2018)
 - ▶ **Direct X Ray-Tracing** (Microsoft, desde 2018),
 - ▶ **Metal Ray-Tracing** (Apple, desde 2020).

Evolución de la funcionalidad: soporte para GPGPU

El uso de las GPUs se ha extendido a otros ámbitos , fuera del campo de la informática gráfica, en lo que se conoce como **cálculo de propósito general en GPUs** (*General Purpose Computing on GPUs*) abreviado como **GPGPU**:

- Los campos de aplicación son principalmente:
 - ▶ Entrenamiento de modelos de IA.
 - ▶ Ejecución de modelos de IA.
 - ▶ Simulación numérica (meteorología, dinámica de fluidos, resistencia de materiales, cálculo de estructuras, etc...)
 - ▶ Minado de criptomonedas.
- Se definen APIs o herramientas para este tipo de cómputo:
 - ▶ **CUDA** (nVidia, desde 2006).
 - ▶ **OpenCL** (consorcio Khronos, desde 2009),

Subsección 3.2.

Motores gráficos

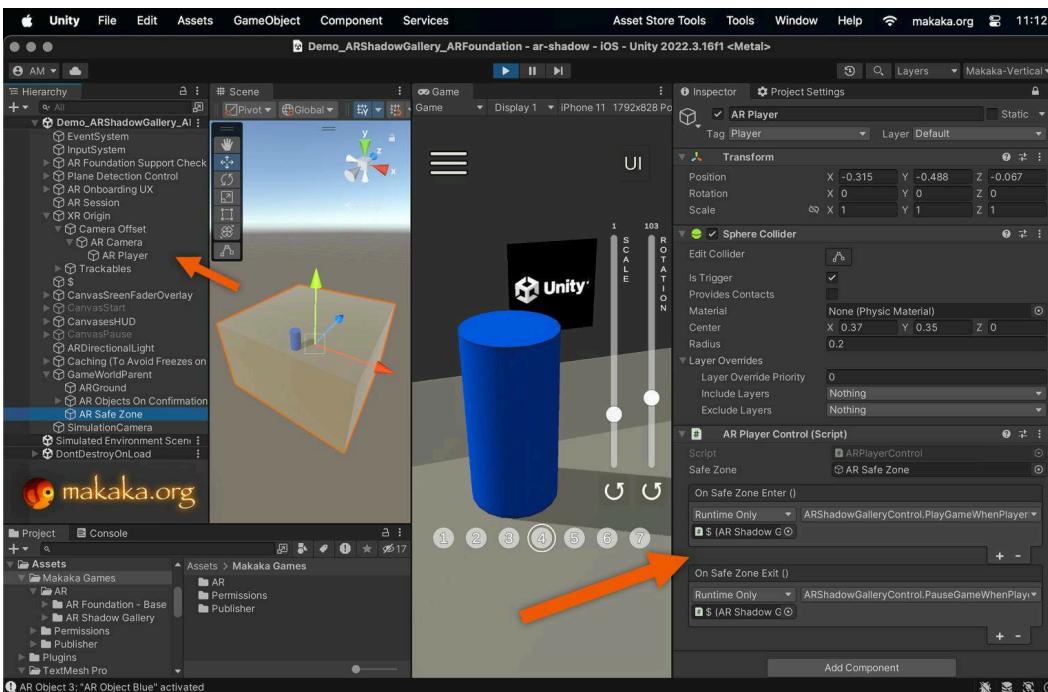
Motores gráficos (*game engines*)

Un **motor gráfico** (*game engine*) es un conjunto de herramientas software que facilita la creación de aplicaciones gráficas interactivas, principalmente videojuegos, aunque también se usan en simulación, visualización científica, realidad virtual, etc...

- Incluyen un motor de renderizado (basado en rasterización y/o ray-tracing), un editor visual, herramientas para animación, físicas, sonido, programación tradicional o visual, entre otras.
- Permiten crear aplicaciones gráficas portables y complejas sin necesidad de considerar detalles de bajo nivel.
- Algunos motores gráficos son de código abierto y gratuitos (Godot), otros son comerciales (Unreal Engine, Unity).
- Los motores gráficos usan APIs gráficas como OpenGL, Vulkan, DirectX o Metal para la rasterización y otras tareas gráficas.

El editor de los motores gráficos

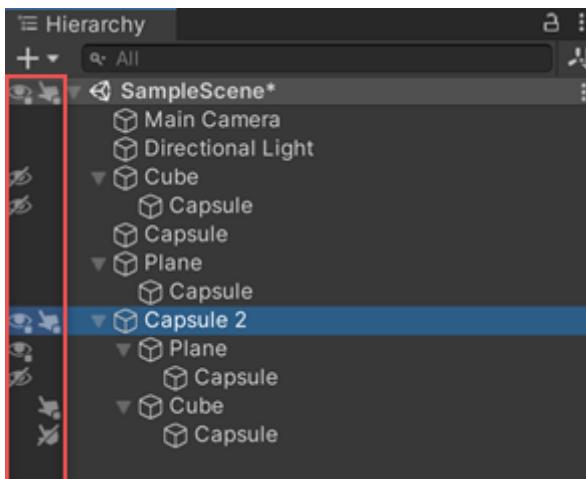
El **editor** de un motor es la aplicación que permite diseñar visual e interactivamente



Captura del editor de Unity, obtenida de: makaka.org/unity-tutorials/ar-testing

Grafo de escena: modelo de la escena y objetos

En un motor, el **grafo de escena** (o la **jerarquía**) es una estructura de datos (un árbol o un grafo dirigido acíclico) que modela las relaciones jerárquicas entre los objetos de la aplicación (escenas, cámaras, luces, objetos geométricos, entre otros)



Captura de una *hierarchy* en Unity, obtenida de:
docs.unity3d.com/6000.2/Documentation/Manual/Hierarchy.html

Programabilidad: visual scripting

El término **visual scripting** se refiere a la posibilidad de programar aspectos de la aplicación gráfica creando un grafo que codifica un **diagrama de flujo de datos**:

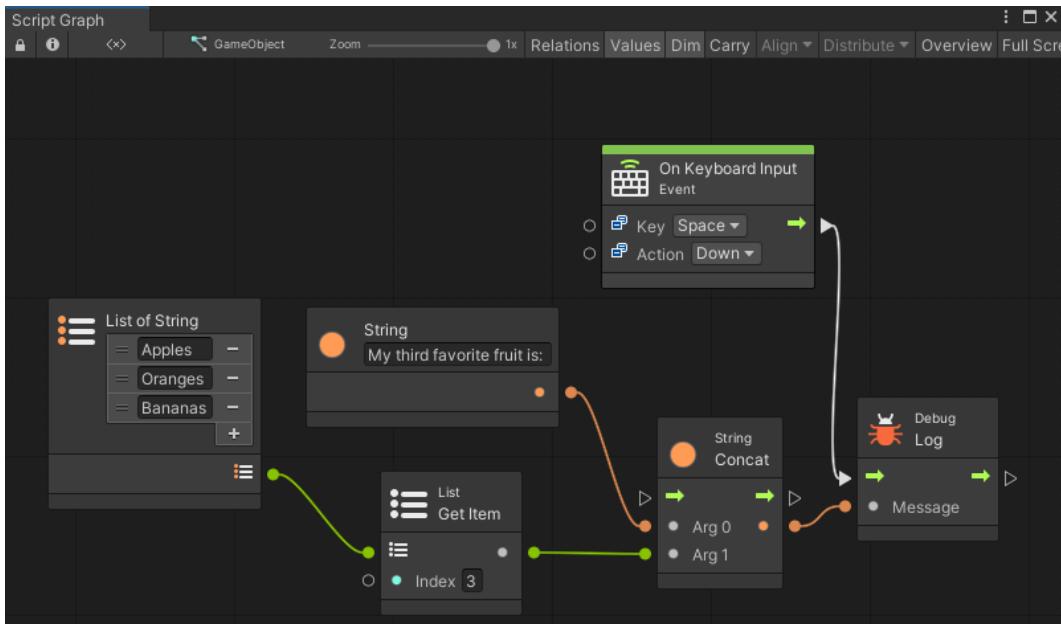
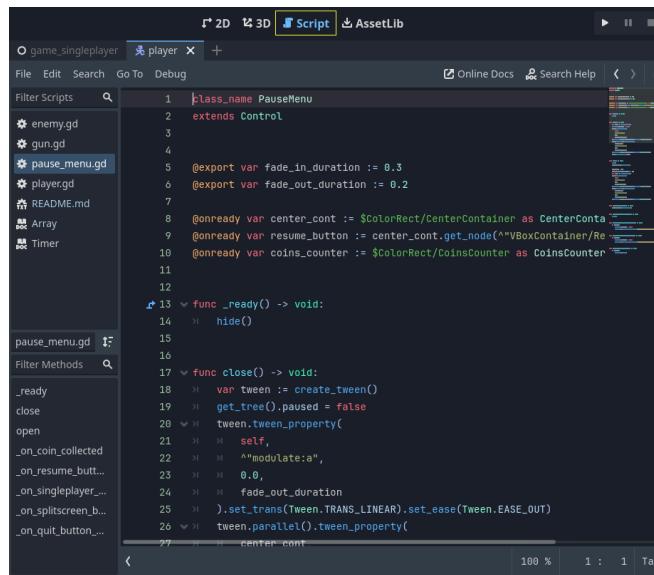


Imagen de un *script graph* de Unity: docs.unity3d.com/Packages/com.unity.visualscripting

Programabilidad: lenguajes de programación

Los motores gráficos también permiten programar partes de la aplicación usando lenguajes de programación tradicionales, como C++, C#, u otros específicos de un motor, como *GDScript* en Godot.



The screenshot shows the Godot Engine's Script Editor interface. The title bar says "Script". The tabs at the top are "2D" and "3D", with "Script" being the active tab. The file path "game_singleplayer/player" is shown, along with a dropdown menu for "player". The main editor area contains the following GDScript code:

```
1 class_name PauseMenu
2 extends Control
3
4
5 @export var fade_in_duration := 0.3
6 @export var fade_out_duration := 0.2
7
8 @onready var center_cont := $ColorRect/CenterContainer as CenterContainer
9 @onready var resume_button := center_cont.get_node("^VBoxContainer/Re")
10 @onready var coins_counter := $ColorRect/CoinsCounter as CoinsCounter
11
12
13 func _ready() -> void:
14     hide()
15
16
17 func close() -> void:
18     var tween := create_tween()
19     get_tree().paused = false
20     tween.tween_property(
21         self,
22         "^modulate:a",
23         0.0,
24         fade_out_duration
25     ).set_trans(Tween.TRANS_LINEAR).set_ease(Tween.EASE_OUT)
26     tween.parallel().tween_property(
27         center_cont
```

Imagen del editor de código fuente *GDScript* de Godot, tomada de:
docs.godotengine.org/en/latest/tutorials/editor/script_editor.html

Programabilidad: shaders

Se pueden programar los shaders, mediante *visual scripting* y/o lenguajes de *shading* bien propios, bien GLSL, HLSL, etc...

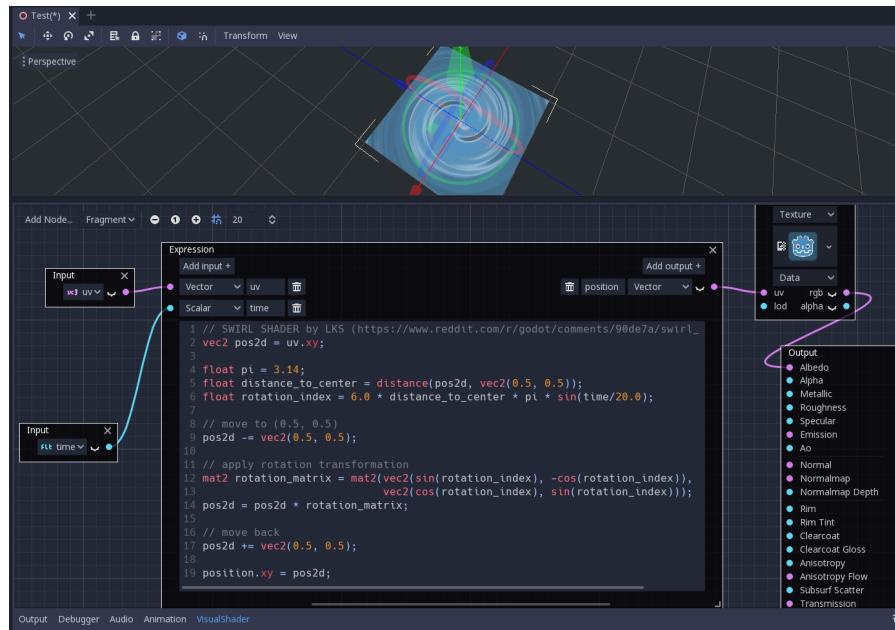


Imagen del editor visual de shaders de Godot, tomada de:

docs.godotengine.org/en/latest/tutorials/shaders/visual_shaders.html

Principales motores gráficos

Los motores gráficos más usados en la actualidad son:

- **Unreal Engine** (Epic Games): de fuentes *accesibles* (no de fuentes abiertas).
- **Unity** (Unity Technologies): privativo.
- **Godot** de fuentes abiertas.

Otros:

- **CryEngine** (Crytek)
- y muchísimos otros, ver: en.wikipedia.org/wiki/List_of_game_engines

Unreal Engine



**UNREAL
ENGINE**

Unreal Engine (www.unrealengine.com) es un motor gráfico de **Epic Games**:

- Desarrollo iniciado en 1995 para el videojuego *Unreal*.
- Programable con **C++** o bien usando *scripting visual* (**Blueprints**).
- Permite desarrollo en Windows, macOS y Linux.
- Fuentes accesibles gratuitamente, bajo registro en GitHub: github.com/EpicGames. Se pueden ver y modificar, pero no redistribuir cambios.
- La licencia permite uso **gratuito** educacional, o bien comercial si se factura menos de 1 millón de dólares, o bien si se comercializa en la [Epic Games Store](https://www.epicgames.com/store)

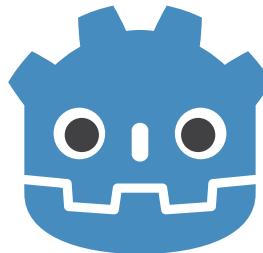
Unity



Unity (unity.com) es un motor gráfico desarrollado por **Unity Technologies**

- La primera versión se publicó en 2005.
- Programable usando el lenguaje **C#** o bien *scripting* visual.
- Permite desarrollo en Windows, macOS y Linux.
- Es software propietario.
- Se ofrece una versión gratuita limitada (sin acceso al código fuente del motor, sin soporte técnico prioritario, sin generación de ejecutables para consolas, entre otras limitaciones)

Godot



Godot (godotengine.org) es un motor gráfico *Open Source* desarrollado inicialmente por Juan Linietsky y Ariel Manzur y luego por una comunidad de desarrolladores:

- La primera versión se publicó en 2014.
- Programable usando **GDScript** , **C#**, **C++** y *scripting* visual.
- Permite desarrollo en Windows, macOS y Linux.
- Es de código abierto, los fuentes están disponible en GitHub:
github.com/godotengine/godot
- Es completamente gratuito y no requiere pago de licencias ni royalties.

Fin de transparencias.

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

Sección 1.

Introducción a Godot

1. El lenguaje de programación GDScript
2. La jerarquía de clases de Godot.
3. El bucle principal de 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.

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** la 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.

Subsección 1.1.

El lenguaje de programación GDScript

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

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
```

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).

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**.

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).

Subsección 1.2.

La jerarquía de clases de Godot.

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**.

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).

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 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.

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, asignandole 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.

Clases para tuplas: coordenadas y colores

Godot define varias clases (fuera de la jerarquía de herencia) para representar tuplas de valores, entre ellas:

Vector2, Vector3, Vector4: tuplas con 2, 3 o 4 elementos de tipo **float**. Se usan para representar posiciones, direcciones, velocidades, normales, tangentes, coordenadas de textura, etc... Se puede operar con ellas mediante suma, resta, producto escalar, producto vectorial (solo **Vector3**), etc...

Vector2i, Vector3i, Vector4i: tuplas con 2, 3 o 4 elementos de enteros (tipo **int**)

Color: representa colores en formato RGBA (rojo, verde, azul, alfa).

Rect2: representa un rectángulo en 2D mediante la posición de su esquina superior izquierda y su tamaño (ancho y alto).

Subsección 1.3.

El bucle principal de Godot.

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.

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(n)** (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.

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

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.

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**.

Tipos de primitivas: puntos y segmentos

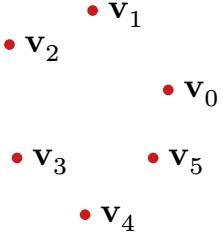
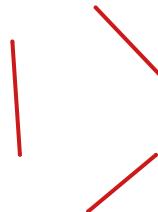
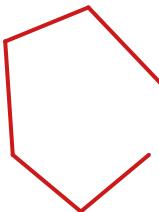
Una lista de n coordenadas de vértices (con $n \neq 1$) puede usarse para codificar puntos o segmentos. Más en concreto, puede codificar:

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

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 $n = 6$:

Puntos PRIMITIVE_POINTS	Segmentos PRIMITIVE_LINES	Polilínea abierta PRIMITIVE_LINE_STRIP
 A group of six red dots representing points. They are labeled v_0 , v_1 , v_2 , v_3 , v_4 , and v_5 . The points are arranged in two columns: v_0 and v_1 are at the top; v_2 and v_3 are in the middle-left; and v_4 and v_5 are in the bottom-left.	 Three red line segments forming a V-shape. The left segment is vertical, the right segment is diagonal pointing down and to the right, and the bottom segment connects them.	 A red line forming an irregular closed polygon with five vertices. It has several sharp angles and some concave features.

Triángulos delanteros y traseros. Cribado.

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

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

Este es el comportamiento por defecto (se puede cambiar).

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

Esta clasificación tiene utilidad especialmente en visualización 3D.

Primitivas de tipos triángulos (rellenos)

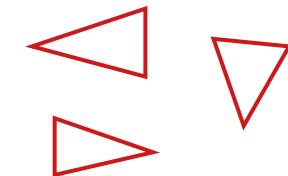
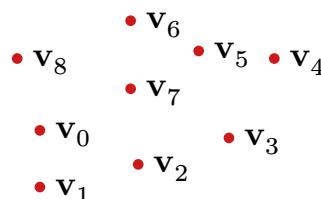
En estos tipos de primitivas la secuencia codifica uno o más triángulos, todos ellos rellenos. Aquí vemos los puntos en las coordenadas y a la derecha un esquema de las aristas de los triángulos que se formarían.

Triángulos

PRIMITIVE_TRIANGLES

Triángulos:

(0, 1, 2), (3, 4, 5), (6, 7, 8), ...

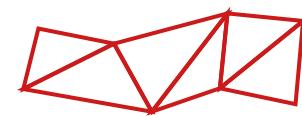
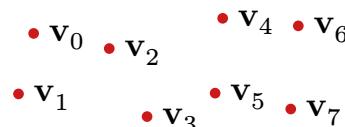


Tira de triángulos

PRIMITIVE_TRIANGLE_STRIP

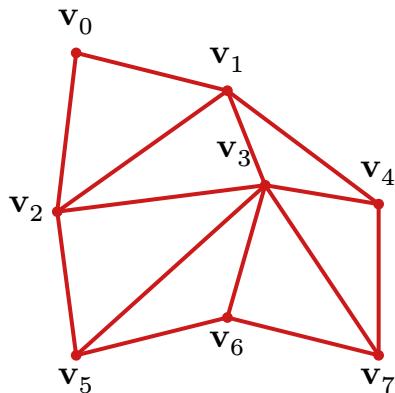
Triángulos:

(0, 1, 2), (2, 1, 3), (2, 3, 4),
(4, 3, 5), (4, 5, 6), (6, 5, 7), ...



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:

v₀, v₂, v₁, v₁, v₂, v₃, v₁, v₃, v₄, v₂, v₅, v₃, v₃, v₅, v₆, v₃, v₆, v₇, v₃, v₇, v₄

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., v₂ aparece 4 veces y v₃ aparece 6 veces).

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:

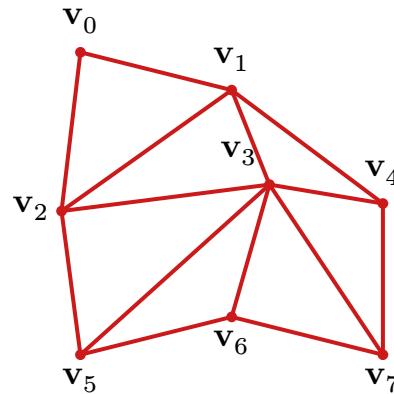
- Se parte de una secuencia V_n de n coordenadas arbitrarias de vértices $V_n = \{\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{n-1}\}$.
- Se usa una secuencia I_m de m **índices** $I_m = \{i_0, i_1, \dots, i_{m-1}\}$ donde cada valor i_j es un entero entre 0 y $n - 1$ (ambos incluidos). Puede haber valores repetidos.
- La secuencia de vértices V_n y la de índices determinan otra secuencia S_m de m vértices:

$$S_m = \{\mathbf{v}_{i_0}, \mathbf{v}_{i_1}, \dots, \mathbf{v}_{i_{m-1}}\}$$

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

Ejemplo de secuencia indexada

En este ejemplo que hemos visto antes



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

$$V_8 = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$

$$I_{21} = \{0, 2, 1, 1, 2, 3, 1, 3, 4, 2, 5, 3, 3, 5, 6, 3, 6, 7, 3, 7, 4\}$$

Subsección 2.2.

Atributos de vértices

Atributos de vértices

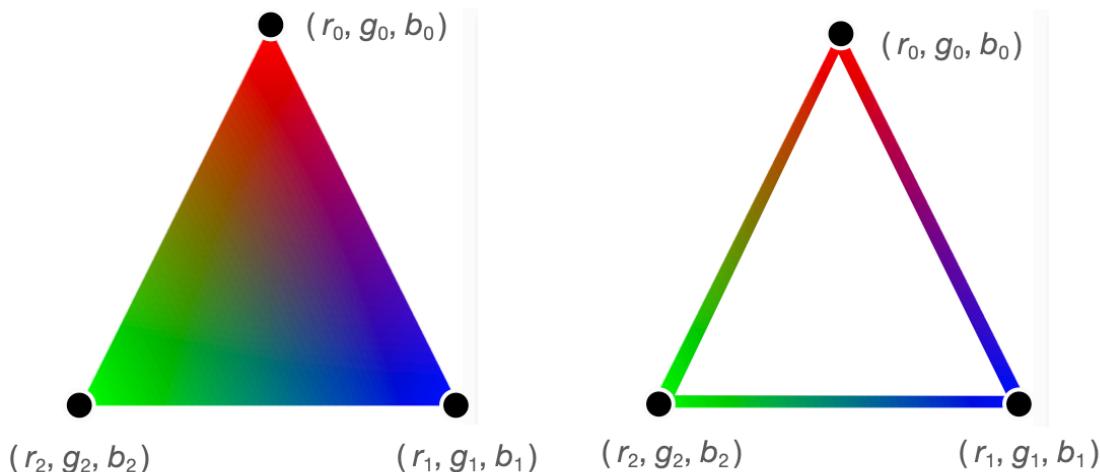
Las coordenadas de su posición se considera un **atributo** de los vértices, es un atributo imprescindible, pero en rasterización se pueden opcionalmente usar otros atributos, por ejemplo:

- 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.

Atributos: colores de vértices

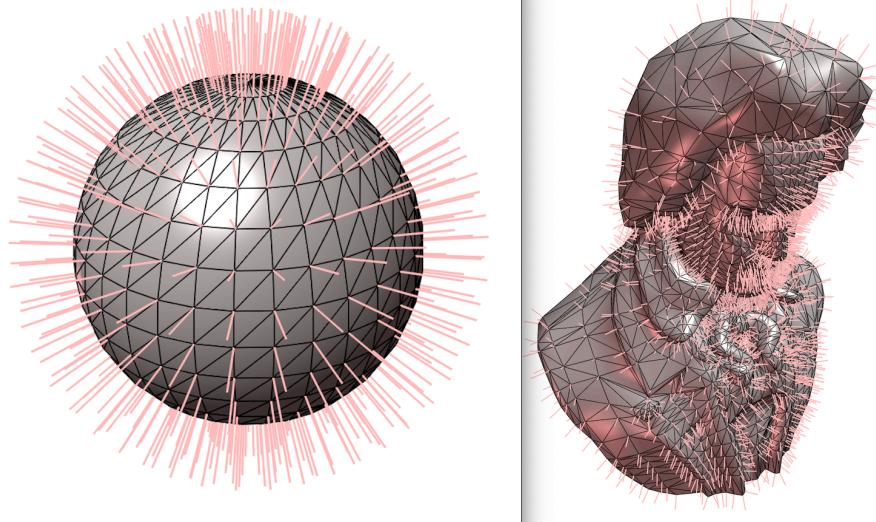
Es posible asignar un color a cada vértice, es una terna RGB con tres reales (r, g, b) , (con valores entre 0 y 1) o bien una cuádrupla RGBA (RGB+transparencia). En el interior (o en las aristas) del polígono se usa interpolación para calcular el color de cada pixel.



- 2. Mallas en Godot..
- 2.2. Atributos de vértices.

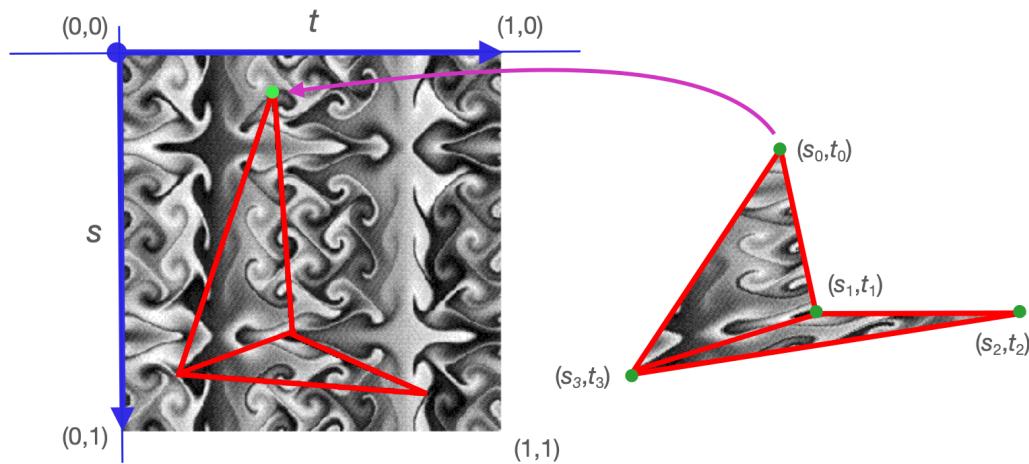
Atributos: normales

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**.

Subsección 2.3.

Modos de envío de datos a la GPU

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.

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.

2. Mallas en Godot..

2.3. Modos de envío de datos a la GPU.

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).

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**.

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

En AOS hay una única secuencia de valores reales:

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

vértice 0

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

$$\text{posiciones} \equiv \left\{ \underbrace{x_0, y_0, z_0}_{\text{posición 0}}, \underbrace{x_1, y_1, z_1}_{\text{posición 1}}, \underbrace{x_2, y_2, z_2}_{\text{posición 2}}, \dots, \underbrace{x_{n_1}, y_{n-1}, z_{n-1}}_{\text{posición } n-1} \right\}$$

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

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

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

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...).

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
Mesh.ARRAY_VERTEX	0	Coordenadas de posición (obligatorias)
Mesh.ARRAY_NORMAL	1	Vector normal
Mesh.ARRAY_TANGENT	2	Vector tangente
Mesh.ARRAY_COLOR	3	Color RGB
Mesh.ARRAY_TEX_UV	4	Coordenadas de textura
Mesh.ARRAY_TEX_UV2	5	Segundas coordenadas de textura
Mesh.ARRAY_CUSTOM0-3	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.

Subsección 2.5.
Mallas en 2D.

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

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 )
```

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**:

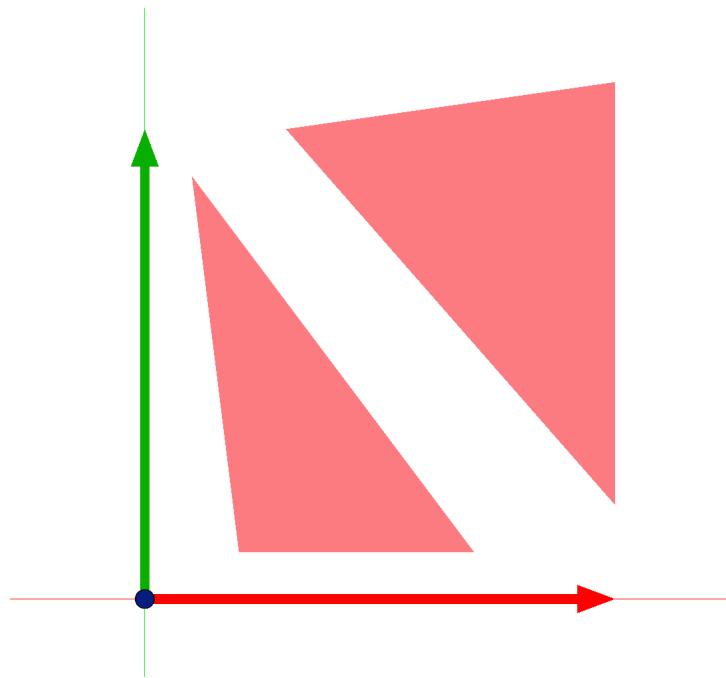
```
## añadir la tabla con las coordenadas de posición de 6 vértices:  
tablas[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([  
    Vector2( 0.2, 0.1 ), Vector2( 0.7, 0.1 ), Vector2( 0.1, 0.9 ),  
    Vector2( 0.3, 1.0 ), Vector2( 1.0, 0.2 ), Vector2( 1.0, 1.1 ),  
])
```

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
```

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):



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 )
```

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

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**:

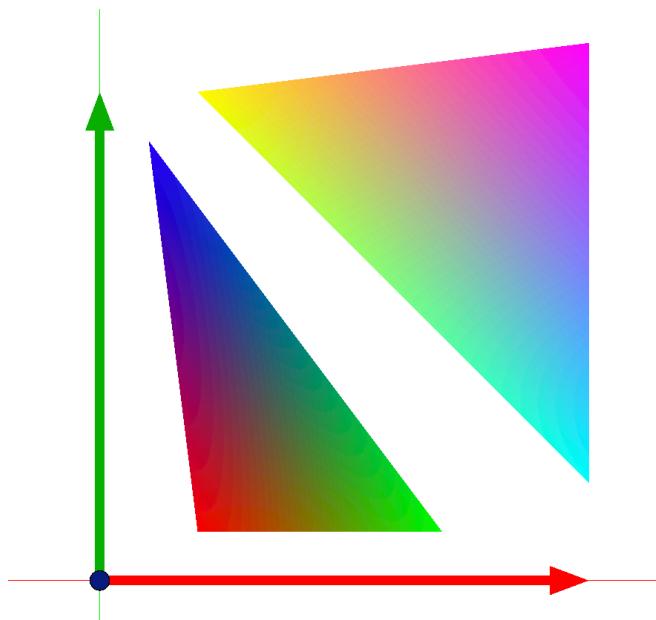
```
var posv := PackedVector2Array([]) # crear array posic. verts. vacío
var colv := PackedColorArray([]) # crear array colores vértices vacío

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
```

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:



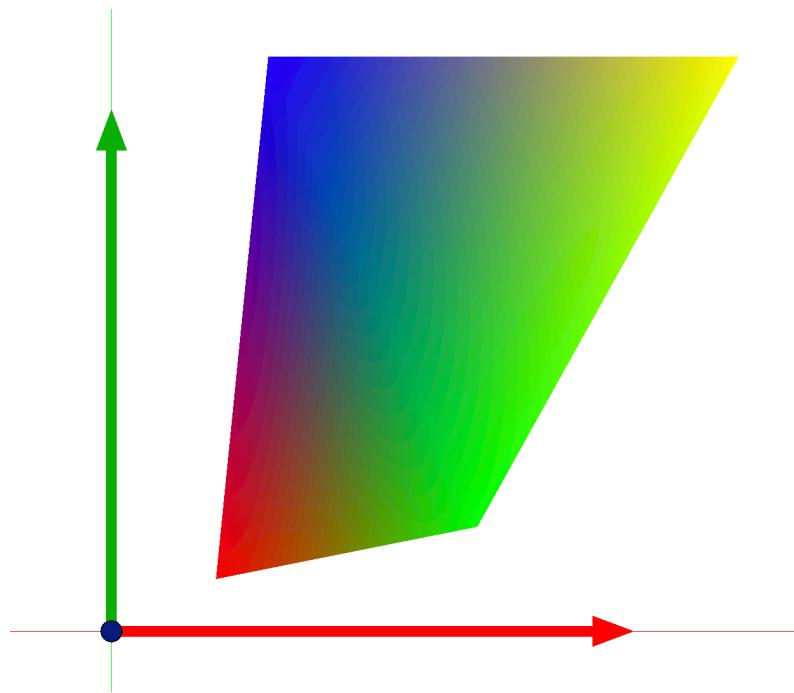
Mallas indexadas en 2D (1/2)

Malla de 4 vértices, que forman dos triángulos adyacentes (3 x 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 )
```

Mallas indexadas en 2D (2/2)

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



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

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 ])

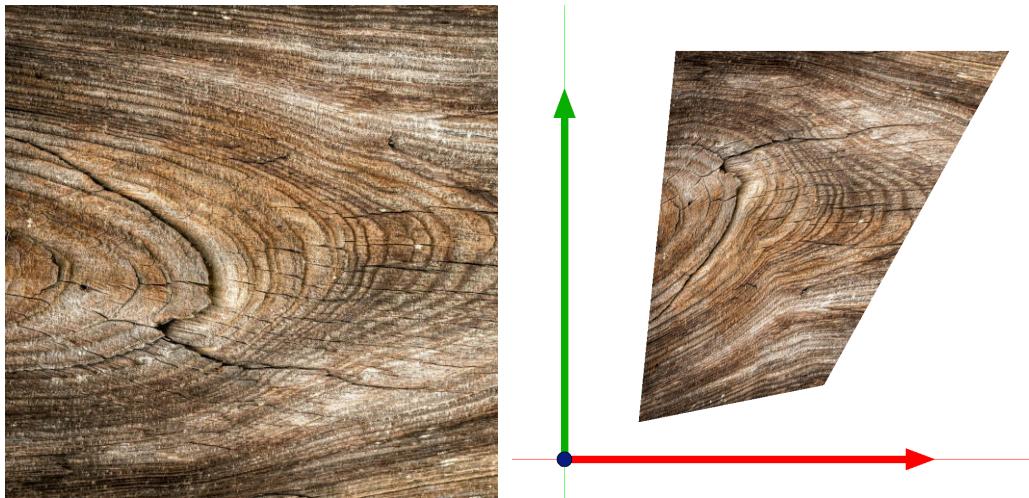
    ## crear el objeto 'mesh'
    mesh = ArrayMesh.new()
    mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )

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

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

Mallas 2D indexada con texturas (2/2)

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



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

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 imagen  
    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
```

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

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) ....
```

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
```

Subsección 2.6.
Mallas en 3D.

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) ....
```

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'

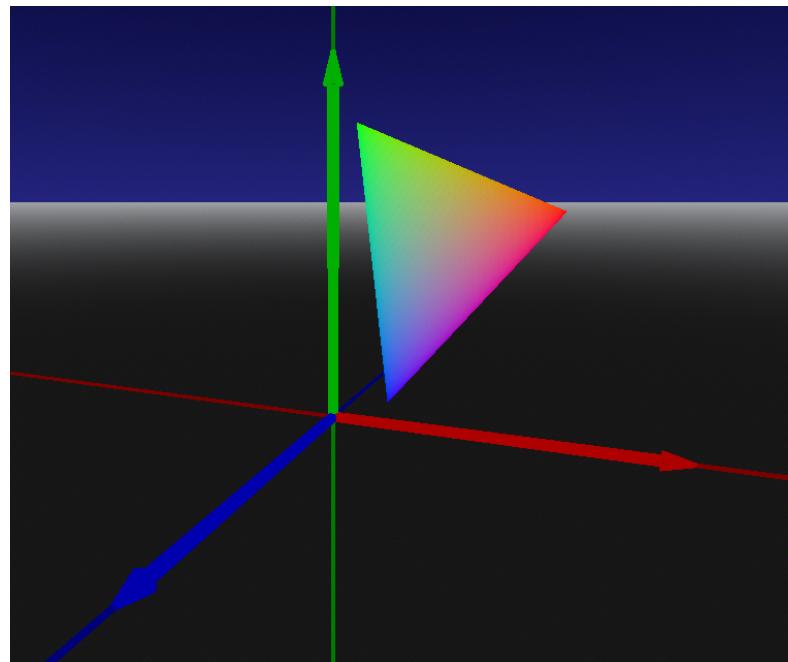
## configurar el objeto con el material ('mat')
mat.vertex_color_use_as_albedo = true # usar colores de vértices
mat.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADED # sin sombr.
mat.cull_mode     = BaseMaterial3D.CULL_DISABLED # sin cribado

## 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.

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

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



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**):

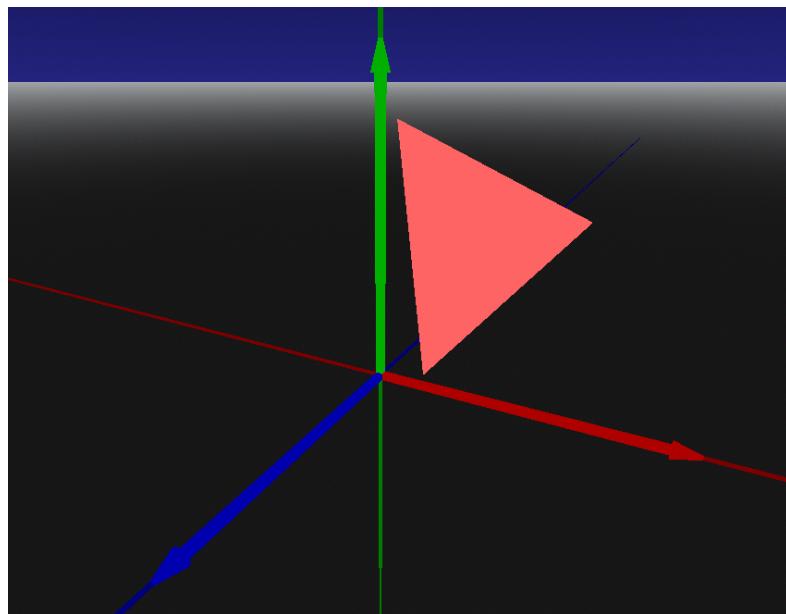
```
# asignarle un material sin iluminación ni sombreado
var mat := StandardMaterial3D.new() # crear objeto 'StandardMaterial3D'

## 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 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 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
])
```

Especificación de un material con iluminación

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

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

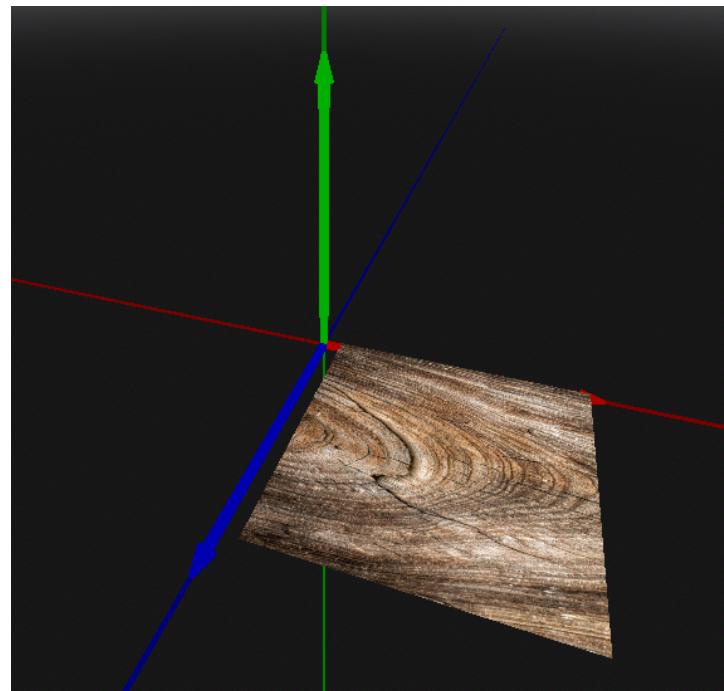
mat.cull_mode      = BaseMaterial3D.CULL_DISABLED # sin cribado
mat.albedo_color  = Color( 1.0, 0.5, 1.0 ) # color base de la malla
mat.metallic       = 0.25
mat.roughness      = 0.15
material_override = mat
```

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" )
```

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 englobantes 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.)

$i_{0,0}$	$i_{0,1}$	$i_{0,2}$
$i_{1,0}$	$i_{1,1}$	$i_{1,2}$
$i_{2,0}$	$i_{2,1}$	$i_{2,2}$
$i_{3,0}$	$i_{3,1}$	$i_{3,2}$
:	:	:
$i_{n-2,0}$	$i_{n-2,1}$	$i_{n-2,2}$
$i_{n-1,0}$	$i_{n-1,1}$	$i_{n-1,2}$

$$0 \leq i_{jk} < m$$

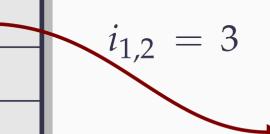
$$i_{1,2} = 3$$


Tabla Vértices (m verts.)

0	x_0	y_0	z_0
1	x_1	y_1	z_1
2	x_2	y_2	z_2
3	x_3	y_3	z_3
4	x_4	y_4	z_4
:	:	:	:
$m-2$	x_{m-2}	y_{m-2}	z_{m-2}
$m-1$	x_{m-1}	y_{m-1}	z_{m-1}

2. Mallas en Godot..

2.7. Mallas indexadas de triángulos en 3D.

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).

2. Mallas en Godot..

2.7. Mallas indexadas de triángulos en 3D.

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

Subsección 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.

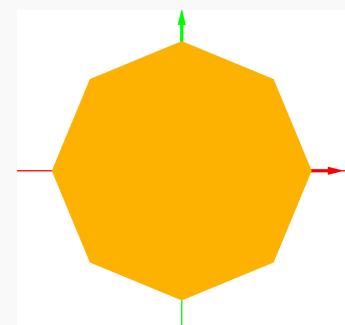
Polígono regular relleno de color plano

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
```



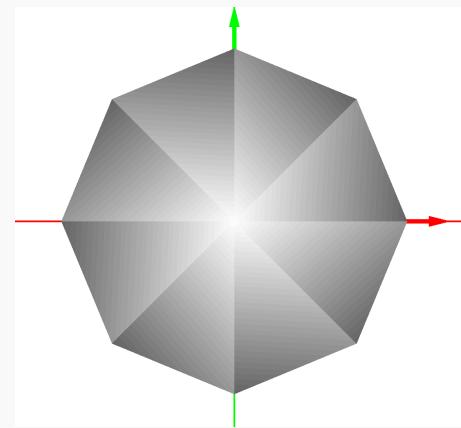
Los valores de estas constantes se podrán cambiar sin tocar nada del resto del script.

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 ?

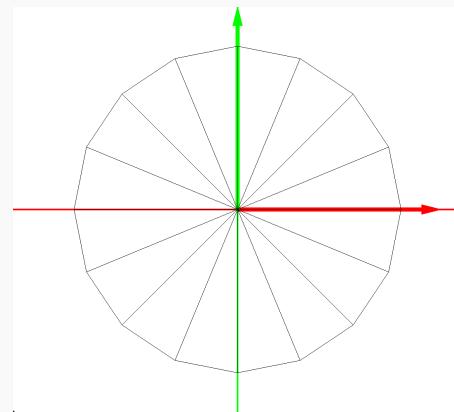
Aristas del polígono regular

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.



Subsección 3.2.

Problemas de creación de mallas 3D

Generación de malla con segmentos de normales

Problema 2.5:

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 asegurate de que a los segmentos no les afecta la iluminación.

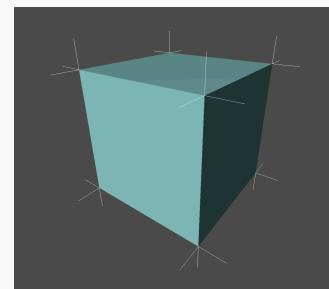
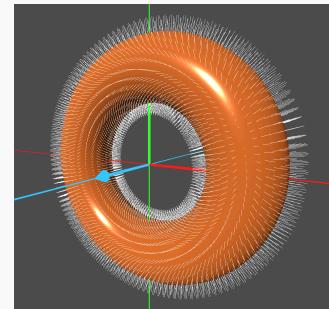
(continua...)

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.



Fin de transparencias.