

*IGY*

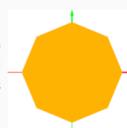
Ejercicios  
por TEMAS

# Tema 2

## 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** de 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.

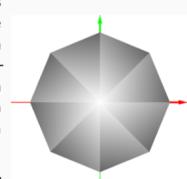
```
1  extends MeshInstance2D  
2  const n : int = 8  
3  const r : float = 0.8  
4  func _ready():  
5  |> var tablas : Array = []  
6  |> tablas.resize(Mesh.ARRAY_MAX)  
7  
8  |> var verts := PackedVector2Array()  
9  |> var cols := PackedColorArray()  
10  
11 |> # centro blanco  
12 |> verts.push_back(Vector2(0, 0))  
13 |> cols.push_back(* Color(1, 1, 1))  
14  
15 |> # contorno alternando grises  
16 |> for i in n:  
17 |> |> var ang := TAU * float(i) / float(n)  
18 |> |> verts.push_back(Vector2(r * cos(ang), r * sin(ang)))  
19  
20 |> |> if i % 2 == 0:  
21 |> |> |> cols.push_back(* Color(0.7, 0.7, 0.7)) # gris claro  
22 |> |> else:  
23 |> |> |> cols.push_back(* Color(0.3, 0.3, 0.5)) # gris oscuro  
24  
25 |> # triángulos  
26 |> var indices := PackedInt32Array()  
27 |> for i in n:  
28 |> |> indices.push_back(0)  
29 |> |> indices.push_back(1 + i)  
30 |> |> indices.push_back(1 + ((i + 1) % n))  
31  
32 |> tablas[Mesh.ARRAY_VERTEX] = verts  
33 |> tablas[Mesh.ARRAY_COLOR] = cols  
34 |> tablas[Mesh.ARRAY_INDEX] = indices  
35  
36 |> mesh = ArrayMesh.new()  
37 |> mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES, tablas)  
38 |>  
39 |> # La clave está en que ya no basta con los vértices del contorno:  
40 # Tenemos el vértice central (para los triángulos que van al centro).  
41 # Y tenemos los n vértices de la circunferencia.  
42 # Cada vértice necesita su propio color, porque ahí se hace la interpolación de colores.  
43 # Respuesta: se necesitan n + 1 vértices en total.
```

```
1  extends MeshInstance2D  
2  
3  const n : int = 8  
4  const r : float = 0.8  
5  
6  func _ready():  
7  |> var tablas : Array = []  
8  |> tablas.resize(Mesh.ARRAY_MAX)  
9  
10 |> var verts := PackedVector2Array()  
11 |> # centro  
12 |> verts.push_back(Vector2(0, 0))  
13  
14 |> # contorno  
15 |> for i in n:  
16 |> |> var ang := TAU * float(i) / float(n)  
17 |> |> verts.push_back(Vector2(r * cos(ang), r * sin(ang)))  
18  
19 |> # triángulos  
20 |> var indices := PackedInt32Array()  
21 |> for i in n:  
22 |> |> indices.push_back(0)  
23 |> |> indices.push_back(1 + i)  
24 |> |> indices.push_back(1 + ((i + 1) % n))  
25  
26 |> tablas[Mesh.ARRAY_VERTEX] = verts  
27 |> tablas[Mesh.ARRAY_INDEX] = indices  
28  
29 |> mesh = ArrayMesh.new()  
30 |> mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES, tablas)  
31  
32 |> modulate = ■ Color(1.0, 0.7, 0.0) # naranja plano
```

## Polígono regular relleno con gradaciones

### Problema 2.2:

Crea otro **Node2D**, y asigne 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 el primero y gris oscuro en el tercero.



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

### 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 ?

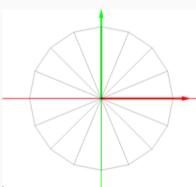
## Aristas del polígono regular

### 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.



```
5 extends Node
6 class_name Normales;
7 static func genSegNormales(
8     verts: PackedVector3Array,
9     norms: PackedVector3Array,
10    lon: float,
11    color: Color
12 ) -> MeshInstance3D:
13     assert(verts.size() == norms.size())
14
15     # 1) Construimos pares de puntos (A=v, B=v+n*lon) → PRIMITIVE_LINES sin indice
16     var seg_verts := PackedVector3Array()
17     var seg_cols := PackedColorArray()
18     seg_verts.resize(verts.size() * 2)
19     seg_cols.resize(verts.size() * 2)
20
21     var k := 0
22     for i in verts.size():
23         var a := verts[i]
24         var n := norms[i]
25         if n.length() > 0.0:
26             n = n.normalized()
27             var b := a + n * lon
28
29             seg_verts[k] = a
30             seg_verts[k+1] = b
31             seg_cols[k] = color
32             seg_cols[k+1] = color
33             k += 2
34
35     # 2) Empaquetaremos en ArrayMesh (LINES)
36     var arrays: Array = []
37     arrays.resize(Mesh.ARRAY_MAX)
38     arrays[Mesh.ARRAY_VERTEX] = seg_verts
39     arrays[Mesh.ARRAY_COLOR] = seg_cols
40
41     var mi := ArrayMesh.new()
42     mi.add_surface_from_arrays(Mesh.PRIMITIVE_LINES, arrays)
43
44     # 3) Material sin iluminación para que el color sea constante
45     var mat := StandardMaterial3D.new()
46     mat.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADDED
47     mat.vertex_color_use_as_albedo = true
48     mat.cull_mode = BaseMaterial3D.CULL_DISABLED
49
50     # 4) Devolvemos el nodo Listo
51     var mi := MeshInstance3D.new()
52     mi.name = "NormalsSegments"
53     mi.mesh = mi
54     mi.material_override = mat
55
56     return mi
```

```
1 extends MeshInstance2D
2 @export var n : int = 16
3 @export var r : float = 0.8
4
5 @func _ready():
6     var m := ArrayMesh.new()
7
8     # ---- 1) Contorno (LINE_STRIP) ----
9     var verta_cont := PackedVector2Array()
10    for i in n:
11        var ang := TAU * float(i) / float(n)
12        verta_cont.push_back(Vector2(r * cos(ang), r * sin(ang)))
13    verta_cont.push_back(Vector2(r, 0)) # cerrar
14
15    var a1 : Array = []
16    a1.resize(Mesh.ARRAY_MAX)
17    a1[Mesh.ARRAY_VERTEX] = verta_cont
18    m.add_surface_from_arrays(Mesh.PRIMITIVE_LINE_STRIP, a1)
19
20    # ---- 2) Radios centro vértice (LINES) ----
21    var verta_rad := PackedVector2Array()
22    for i in n:
23        var ang := TAU * float(i) / float(n)
24        verta_rad.push_back(Vector2(0, 0))
25        verta_rad.push_back(Vector2(r * cos(ang), r * sin(ang))) # centro
26
27    var cols_rad := PackedColorArray()
28    for i in verta_rad.size():
29        cols_rad.push_back(Color(0.7, 0.7, 0.7, 0.8)) # gris claro
30
31    var a2 : Array = []
32    a2.resize(Mesh.ARRAY_MAX)
33    a2[Mesh.ARRAY_VERTEX] = verta_rad
34    a2[Mesh.ARRAY_COLOR] = cols_rad
35    m.add_surface_from_arrays(Mesh.PRIMITIVE_LINES, a2)
36
37    mesh = m
38
39    # Colores en el contorno (negro) y radios (gris) vía modulado por superficie no es posible:
40    # pero puedes dar el contorno en negro con el modulado global:
41    # modulado = Color(0.0, 0.0, 0.0, 1.0)
```

### 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...)

Sesión 2: El engine Godot. Mallas.

Created 2025-09-30

Page 97 / 99.

3. Problemas.

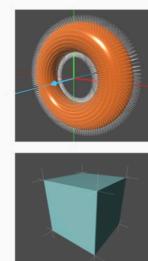
3.2. Problemas de creación de mallas 3D.

## 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.



# Diferencia entre mallas indexadas y mallas no indexadas

## 1. Malla No Indexada (Triangle Soup)

En este método, le das a Godot una lista bruta de vértices. La GPU simplemente lee los vértices en orden de 3 en 3 para formar triángulos.

- **Funcionamiento:** Si quieras dibujar un cuadrado (que se compone de 2 triángulos), necesitas definir 6 vértices.
- **El problema:** Dos de esos vértices están duplicados (la diagonal compartida), pero la GPU no lo sabe, así que guarda los datos dos veces y procesa el mismo vértice dos veces.
- **Uso en Godot:** Simplemente llenas el ARRAY\_VERTEX y dejas el ARRAY\_INDEX vacío.

## 2. Malla Indexada

Aquí separas la información en dos listas:

1. **Lista de Vértices (Vertex Buffer):** Una lista con las coordenadas de los puntos únicos. Para un cuadrado, solo guardas 4 vértices.
2. **Lista de Índices (Index Buffer):** Una lista de números enteros (enteros int) que le dice a Godot en qué orden conectar esos puntos.

<!-- end list -->

- **Funcionamiento:** Para el cuadrado, la lista de índices diría: "Conecta el 0, 1 y 2. Luego conecta el 2, 3 y 0".
- **La ventaja:** Reutilizas los vértices existentes. Ahorras memoria (porque un int ocupa mucho menos que un vértice completo con posición, normal, UV, color, etc.) y la GPU  la Caché de Vértices para no recalcular puntos ya procesados.

### Opción A: NO Indexada (6 vértices definidos)

gdscript

```
var vertices = PackedVector3Array([
    Vector3(0,0,0), Vector3(1,0,0), Vector3(1,1,0), # Triángulo 1
    Vector3(1,1,0), Vector3(0,1,0), Vector3(0,0,0) # Triángulo 2 (Repetimos 0,0,0 y 1,1,0)
])

# Crear ArrayMesh
var arr_mesh = ArrayMesh.new()
var arrays = []
arrays.resize(Mesh.ARRAY_MAX)
arrays[Mesh.ARRAY_VERTEX] = vertices
# NO pasamos nada en arrays[Mesh.ARRAY_INDEX]

arr_mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES, arrays)
```

### Opción B: Indexada (4 vértices + 6 índices)

gdscript

```
# Solo los 4 puntos únicos de las esquinas
var vertices = PackedVector3Array([
    Vector3(0,0,0), # Índice 0
    Vector3(1,0,0), # Índice 1
    Vector3(1,1,0), # Índice 2
    Vector3(0,1,0) # Índice 3
])

# Definimos el orden de conexión
var indices = PackedInt32Array([
    0, 1, 2, # Primer triángulo
    2, 3, 0 # Segundo triángulo (Reutilizamos el 2 y el 0)
])

var arr_mesh = ArrayMesh.new()
var arrays = []
arrays.resize(Mesh.ARRAY_MAX)
arrays[Mesh.ARRAY_VERTEX] = vertices
arrays[Mesh.ARRAY_INDEX] = indices # <--- AQUÍ ESTÁ LA CLAVE

arr_mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES, arrays)
```

Mejor

# Tema 3

## Problema 1.9.

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

En sistema orthonormal producto escalar  
 $\Rightarrow \begin{cases} i \cdot i = j \cdot j = k \cdot k = 1 \\ i \cdot j = i \cdot k = j \cdot k = 0 \end{cases}$

Sea  $C = [\hat{i}, \hat{j}, \hat{k}, \vec{0}]$  un marco de referencia cartesiano.

$$\begin{aligned} \text{Sea } \vec{a} &= C(a_x, a_y, a_z, 0)^t \quad \text{y } \vec{b} = C(b_x, b_y, b_z, 0)^t \Rightarrow \text{linealidad} \\ \Rightarrow \vec{a} \cdot \vec{b} &= (a_x \cdot \hat{i} + a_y \cdot \hat{j} + a_z \cdot \hat{k}) (b_x \cdot \hat{i} + b_y \cdot \hat{j} + b_z \cdot \hat{k}) = \\ &= a_x (\hat{i} \cdot (b_x \cdot \hat{i} + b_y \cdot \hat{j} + b_z \cdot \hat{k})) + a_y (\hat{j} \cdot (b_x \cdot \hat{i} + b_y \cdot \hat{j} + b_z \cdot \hat{k})) + \\ &\quad + a_z (\hat{k} \cdot (b_x \cdot \hat{i} + b_y \cdot \hat{j} + b_z \cdot \hat{k})) = \\ &= a_x (a_x \cdot \hat{i} \cdot \hat{i} + a_y \cdot \hat{j} \cdot \hat{i} + a_z \cdot \hat{k} \cdot \hat{i}) + a_y (a_x \cdot \hat{i} \cdot \hat{j} + a_y \cdot \hat{j} \cdot \hat{j} + a_z \cdot \hat{k} \cdot \hat{j}) + \\ &\quad + a_z (a_x \cdot \hat{i} \cdot \hat{k} + a_y \cdot \hat{j} \cdot \hat{k} + a_z \cdot \hat{k} \cdot \hat{k}) = \xrightarrow{\hat{i} \cdot \hat{i} = \hat{j} \cdot \hat{j} = \hat{k} \cdot \hat{k} = 1} \\ &= a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z \quad \text{perpendiculares} \end{aligned}$$

## Problema 1.10.

Demuestra que el producto vectorial de dos vectores se puede calcular usando sus coordenadas en cualquier marco cartesiano según se indica en la transparencia anterior, a partir de las propiedades que definen dicho producto vectorial.

Sistema orthonormal producto vectorial

$$\begin{aligned} i \cdot i = j \cdot j = k \cdot k &= 0 & i = \hat{i} \\ i \cdot j = j \cdot k = k \cdot i &= 1 & j = \hat{j} \\ j \cdot i = -k \cdot j = i \cdot k &= 0 & k = \hat{k} \end{aligned}$$

Sea  $C = [\hat{i}, \hat{j}, \hat{k}, \vec{0}]$  un marco de referencia cartesiano.

$$\begin{aligned} \text{Sea } \vec{a} &= C(a_x, a_y, a_z, 0)^t \quad \text{y } \vec{b} = C(b_x, b_y, b_z, 0)^t \Rightarrow \text{linealidad} \\ \Rightarrow \vec{a} \times \vec{b} &= (a_x \cdot \hat{i} + a_y \cdot \hat{j} + a_z \cdot \hat{k}) \times (b_x \cdot \hat{i} + b_y \cdot \hat{j} + b_z \cdot \hat{k}) = \\ &= a_x (\hat{i} \times (b_x \cdot \hat{i} + b_y \cdot \hat{j} + b_z \cdot \hat{k})) + a_y (\hat{j} \times (b_x \cdot \hat{i} + b_y \cdot \hat{j} + b_z \cdot \hat{k})) + \\ &\quad + a_z (\hat{k} \times (b_x \cdot \hat{i} + b_y \cdot \hat{j} + b_z \cdot \hat{k})) = \\ &= a_x (b_x (\hat{i} \times \hat{i}) + b_y (\hat{i} \times \hat{j}) + b_z (\hat{i} \times \hat{k})) + a_y (b_x (\hat{j} \times \hat{i}) + b_y (\hat{j} \times \hat{j}) + b_z (\hat{j} \times \hat{k})) + \\ &\quad + a_z (b_x (\hat{k} \times \hat{i}) + b_y (\hat{k} \times \hat{j}) + b_z (\hat{k} \times \hat{k})) = \xrightarrow{\substack{\hat{i} \times \hat{i} = 0 \\ \hat{j} \times \hat{j} = 0 \\ \hat{k} \times \hat{k} = 0}} \hat{i} \times \hat{i} = \hat{e}_y \quad \hat{i} \times \hat{j} = \hat{e}_z \\ &= a_x b_y \hat{i} - a_x b_y \hat{j} - a_y b_x \hat{i} + a_y b_x \hat{j} + a_z b_x \hat{i} - a_z b_x \hat{j} = \\ &= \hat{i} (a_y b_z - a_z b_y) + \hat{j} (a_z b_x - a_x b_z) + \hat{k} (a_x b_y - a_y b_x) = \begin{pmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{pmatrix} \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix} \end{aligned}$$

## Problema 1.11.

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

Sea  $C = [\hat{i}, \hat{j}, \hat{k}, \vec{0}]$  un marco de referencia cartesiano.

$$\text{Sea } \vec{a} = C(a_x, a_y, a_z, 0)^t \quad \text{y } \vec{b} = C(b_x, b_y, b_z, 0)^t$$

$$\text{Hemos visto que } \vec{a} \times \vec{b} = \hat{i} (a_y b_z - a_z b_y) + \hat{j} (a_z b_x - a_x b_z) + \hat{k} (a_x b_y - a_y b_x)$$

Luego:

$$\begin{aligned} \vec{a} \cdot (\vec{a} \times \vec{b}) &= (a_x \cdot \hat{i} + a_y \cdot \hat{j} + a_z \cdot \hat{k}) \cdot [\hat{i} (a_y b_z - a_z b_y) + \hat{j} (a_z b_x - a_x b_z) + \hat{k} (a_x b_y - a_y b_x)] \\ &= a_x (a_y b_z - a_z b_y) + a_y (a_z b_x - a_x b_z) + a_z (a_x b_y - a_y b_x) = \\ &\quad \xrightarrow{\text{distributiva y que } \hat{i} \cdot \hat{i} = 1 \quad \hat{i} \cdot \hat{j} = 0} \\ &= a_x a_y b_z - a_x a_z b_y + a_y a_z b_x - a_y a_x b_z + a_z a_x b_y - a_z a_y b_x = 0 \end{aligned}$$

Análogo para  $\vec{b}$ .

### Problema 2.10.

Demuestra que el producto escalar de vectores en 2D es invariante por rotación, es decir, que para cualquier ángulo  $\theta$  y vectores  $\vec{a}$  y  $\vec{b}$  se cumple:

$$R_\theta(\vec{a}) \cdot R_\theta(\vec{b}) = \vec{a} \cdot \vec{b}$$

(usa las coordenadas de  $\vec{a}$  y  $\vec{b}$  en un marco cartesiano cualquiera)

En la rotación se conservan los escalares y se conservan los ángulos

sean  $\vec{a}, \vec{b}$  dos vectores en el plano

$$\vec{a} \cdot \vec{b} = \|\vec{a}\| \cdot \|\vec{b}\| \cdot \cos(\vec{a}, \vec{b})$$

$$\left\{ \begin{array}{l} \|R_\theta(\vec{a})\| = \|\vec{a}\| \\ \|R_\theta(\vec{b})\| = \|\vec{b}\| \\ \cos(R_\theta(\vec{a}), R_\theta(\vec{b})) = \cos(\vec{a}, \vec{b}) \end{array} \right.$$

$$\Rightarrow \vec{a} \cdot \vec{b} = \|\vec{a}\| \cdot \|\vec{b}\| \cdot \cos(\vec{a}, \vec{b}) = \|\vec{a}\| \|\vec{b}\| \cdot \|\vec{R}_\theta(\vec{a})\| \|\vec{R}_\theta(\vec{b})\| \cdot \cos(\vec{R}_\theta(\vec{a}), \vec{R}_\theta(\vec{b})) = \vec{R}_\theta(\vec{a}) \cdot \vec{R}_\theta(\vec{b})$$

$$\left. \begin{array}{l} \vec{a} = a_x \hat{x} + a_y \hat{y} \\ \vec{b} = b_x \hat{x} + b_y \hat{y} \end{array} \right\} \quad \vec{a} \cdot \vec{b} = a_x b_x + a_y b_y$$

$$\begin{aligned} R(\vec{a}) &= (a_x \cos(\theta) - a_y \sin(\theta)) \hat{x} + (a_y \cos(\theta) + a_x \sin(\theta)) \hat{y} \\ R(\vec{b}) &= (b_x \cos(\theta) - b_y \sin(\theta)) \hat{x} + (b_y \cos(\theta) + b_x \sin(\theta)) \hat{y} \end{aligned} \quad \Rightarrow$$

$$R(\vec{a}) \cdot R(\vec{b}) = (a_x (\cos(\theta) - \sin(\theta)) (b_x \cos(\theta) - b_y \sin(\theta)) + (a_y \cos(\theta) + a_x \sin(\theta)) (b_y \cos(\theta) + b_x \sin(\theta)) =$$

$$\cancel{a_x b_x \cos^2 \theta - a_y b_x \sin \theta \cos \theta - a_x b_y \sin \theta \sin \theta + a_y b_y \sin^2 \theta} + \cancel{a_x b_y \cos^2 \theta + a_x b_x \sin \theta \cos \theta + a_y b_x \cos \theta \sin \theta + a_y b_y \sin \theta \sin \theta} =$$

$$a_x b_x + a_y b_y$$

### Problema 2.12.

Demuestra que el producto escalar de vectores en 3D es invariante por rotaciones elementales (usa tu solución al problema 10)

$$\begin{aligned} \vec{a} &= a_x \hat{x} + a_y \hat{y} + a_z \hat{z} \\ \vec{b} &= b_x \hat{x} + b_y \hat{y} + b_z \hat{z} \end{aligned} \quad \vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z$$

$$R_{(x,\theta)}(\vec{a}) = (a_x, a_y \cos(\theta) - a_z \sin(\theta), a_y \sin(\theta) + a_z \cos(\theta))$$

$$R_{(z,\theta)}(\vec{b}) = (b_x, b_y \cos(\theta) - b_z \sin(\theta), b_y \sin(\theta) + b_z \cos(\theta))$$

$$R_{(x,\theta)}(\vec{a}) \cdot R_{(z,\theta)}(\vec{b}) = a_x b_x + (a_y \cos(\theta) - a_z \sin(\theta)) (b_y \cos(\theta) - b_z \sin(\theta)) + (a_y \sin(\theta) + a_z \cos(\theta)) (b_y \sin(\theta) + b_z \cos(\theta)) =$$

$$\cancel{a_x b_x + a_y b_y \cos^2 \theta - a_y b_x \sin \theta \cos \theta - a_y b_z \sin \theta \sin \theta + a_z b_x \sin \theta \cos \theta + a_z b_y \sin^2 \theta} + \cancel{a_y b_x \sin \theta \cos \theta + a_z b_y \sin \theta \sin \theta + a_z b_z \cos^2 \theta} =$$

$$a_x b_x + a_y b_y + a_z b_z$$

$$R_{(y,\theta)}(\vec{a}) = (a_x \cos(\theta) + a_z \sin(\theta), a_y, -a_x \sin(\theta) + a_z \cos(\theta))$$

$$R_{(y,\theta)}(\vec{b}) = (b_x \cos(\theta) + b_z \sin(\theta), b_y, -b_x \sin(\theta) + b_z \cos(\theta))$$

$$R_{(y,\theta)}(\vec{a}) \cdot R_{(y,\theta)}(\vec{b}) = (a_x \cos(\theta) + a_z \sin(\theta)) (b_x \cos(\theta) + b_z \sin(\theta)) + a_y b_y + (-a_x \sin(\theta) + a_z \cos(\theta)) (-b_x \sin(\theta) + b_z \cos(\theta)) =$$

$$\cancel{a_x b_x \cos^2 \theta + a_x b_z \sin \theta \cos \theta + a_x b_z \sin \theta \cos \theta + a_z b_x \sin^2 \theta + a_y b_y \cos^2 \theta + a_y b_z \sin \theta \cos \theta - a_y b_z \sin \theta \cos \theta + a_z b_x \sin \theta \cos \theta}.$$

$$a_x b_x + a_y b_y + a_z b_z$$

$$R_{(z,\theta)}(\vec{a}) = (a_x \cos(\theta) - a_y \sin(\theta), a_x \sin(\theta) + a_y \cos(\theta), a_z)$$

$$R_{(z,\theta)}(\vec{b}) = (b_x \cos(\theta) - b_y \sin(\theta), b_x \sin(\theta) + b_y \cos(\theta), b_z)$$

$$R_{(z,\theta)}(\vec{a}) \cdot R_{(z,\theta)}(\vec{b}) = (a_x \cos(\theta) - a_y \sin(\theta), a_x \sin(\theta) + a_y \cos(\theta)) (b_x \cos(\theta) - b_y \sin(\theta), b_x \sin(\theta) + b_y \cos(\theta), b_z) =$$

$$(a_x \cos(\theta) - a_y \sin(\theta)) (b_x \cos(\theta) - b_y \sin(\theta)) + (a_x \sin(\theta) + a_y \cos(\theta)) (b_x \sin(\theta) + b_y \cos(\theta)) + a_z b_z =$$

$$\cancel{a_x b_x \cos^2 \theta - a_y b_x \sin \theta \cos \theta - a_y b_x \sin \theta \cos \theta + a_y b_y \sin^2 \theta + a_x b_y \sin \theta \cos \theta + a_y b_y \sin \theta \cos \theta + a_z b_z \cos^2 \theta} + \cancel{a_y b_z \sin \theta \cos \theta + a_z b_x \sin \theta \cos \theta} =$$

$$a_x b_x + a_y b_y + a_z b_z$$

### Problema 2.11.

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

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

$$\text{Sea } \vec{v} = a \hat{x} + b \hat{y}$$

$$\|\vec{v}\| = \sqrt{a^2 + b^2}$$

$$R_\theta(\vec{v}) = (a \cos(\theta) - b \sin(\theta), a \sin(\theta) + b \cos(\theta))$$

$$\|R_\theta(\vec{v})\| = \sqrt{(a \cos(\theta) - b \sin(\theta))^2 + (a \sin(\theta) + b \cos(\theta))^2} = \sqrt{a^2 \cos^2 \theta - 2ab \sin \theta \cos \theta + b^2 \sin^2 \theta + a^2 \sin^2 \theta + 2ab \sin \theta \cos \theta} =$$

$$= \sqrt{a^2 + b^2}$$



**Problema 2.7.**

Demuestra que  $\vec{u}$  y  $P(\vec{u})$  son siempre perpendiculares según la definición anterior (es decir, siempre  $\vec{u} \cdot P(\vec{u}) = 0$ ).

$$\begin{aligned}\vec{u} &= a\vec{x} + b\vec{y} \\ P(\vec{u}) &= -b\vec{x} + a\vec{y}\end{aligned}\quad \left\{ \begin{array}{l} \vec{u} \cdot P(\vec{u}) = (a\vec{x} + b\vec{y}) \cdot (-b\vec{x} + a\vec{y}) = -ab(\vec{x} \cdot \vec{x}) - bb(\vec{y} \cdot \vec{x}) + aa(\vec{x} \cdot \vec{y}) + ab(\vec{y} \cdot \vec{x}) = \\ ab - ba = 0 \end{array} \right.$$

**Problema 2.8.**

Describe como se podría definir una rotación hacia la derecha (en el sentido de las agujas del reloj) en lugar de a izquierdas.

Rotación a izquierdas:

$$P(\vec{v}) = -b\vec{x} + a\vec{y}$$

Rotación a derechas

$$P(\vec{v}) = b\vec{x} - a\vec{y}$$

**Problema 2.9.**

Demuestra que la transformación afín  $P$  (cuando se aplica a vectores, no a puntos) no depende del marco cartesiano  $C$  con respecto al cual expresamos las coordenadas  $(a, b)$  (en el caso de aplicarla a puntos, la rotación de  $90^\circ$  es entorno al punto origen ó de  $C$ ).

Una transformación afín se puede expresar como  $P(v) = Av + b$ , donde  $A$  es una matriz y  $b$  un vector de traslación. Cuando aplicamos  $P$  a un vector  $v$ , obtenemos un nuevo vector  $v'$

Supongamos que tenemos dos marcos cartesianos  $C$  y  $C'$ . Las coordenadas de  $v$  en  $C$  son  $(a, b)$  y las coordenadas de  $v$  en  $C'$  son  $(a', b')$ . La relación entre ambas es una transformación lineal  $Tv = v'$ , donde  $T$  es la matriz de transformación de  $C$  a  $C'$

Aplicamos la transformación afín en el marco  $C$ :  $P(v) = Av + b$

Aplicamos la transformación afín en marco  $C'$ :  $P(Tv) = ATv + b$ . Observamos que la forma de la transformación ha cambiado, solo que  $A$  se ha multiplicado por  $T \Rightarrow$  la transformación no depende del marco cartesiano

caso matriz de rotación:  $\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$

**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 las rotaciones elementales en 3D no modifican la longitud de un vector (usa tu solución al problema 8)

**Problema 3.8**

Demuestra que la rotación no cambia la longitud del vector (2D).

$$\|R_\theta(\vec{v})\|^2 = R_\theta(\vec{v}) \cdot R_\theta(\vec{v}) = \vec{v}^T R_\theta^T R_\theta \vec{v} = \vec{v}^T \vec{v} = \|\vec{v}\|^2$$

La rotación conserva la longitud.

**Problema 3.10**

Rotaciones 3D no cambian la longitud de un vector.

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

Mismo razonamiento que antes: las matrices de rotación son ortogonales  $\rightarrow$  preservan la longitud.

**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.

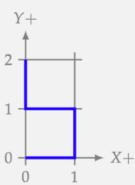
**Problema 3.11:**

Demuestra que el producto vectorial de dos vectores rota igual que lo hacen esos dos vectores, es decir, que para cualesquiera dos vectores  $\vec{u}$  y  $\vec{v}$  un ángulo

### 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.



### Global

```

1  # res://autoload/gancho.gd
2  extends Node
3  ## AutoLoad (singleton) con la función pedida 'gancho()'
4  ## Devuelve un objeto Mesh (ArrayMesh) que dibuja una POLILÍNEA
5  ## con la forma de la figura: (0,0) → (1,0) → (1,1) → (0,1) → (0,0)
6  ## En Godot 4 usamos Mesh.PRIMITIVE_LINE_STRIP para unir cada punto
7  ## con el siguiente en orden.
8
9  func gancho() -> Mesh:
10    >| # Coordenadas 2D en unidades tal como en el enunciado
11    >| var pts := PackedVector2Array([
12    >|   >| Vector2(0, 0),
13    >|   >| Vector2(1, 0),
14    >|   >| Vector2(1, 1),
15    >|   >| Vector2(0, 1),
16    >|   >| Vector2(0, 0),
17    >|   >])
18
19    >| # Construcción de la ArrayMesh
20    >| var arrays: Array = []
21    >| arrays.resize(Mesh.ARRAY_MAX)
22    >| arrays[Mesh.ARRAY_VERTEX] = pts
23
24    >| var am := ArrayMesh.new()
25    >| # LINE_STRIP = polilinea (no hace falta pasar indices)
26    >| am.add_surface_from_arrays(Mesh.PRIMITIVE_LINE_STRIP, arrays)
27
28    >| return am

```

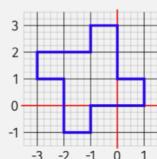
```

1  # res://scripts/GanchoMesh.gd
2  extends MeshInstance2D
3
4  func _ready() -> void:
5    >| # Llama a la función del autoload y asigna la malla resultante
6    >| mesh = Gancho.gancho()
7
8    >| # Color azul como pide el enunciado (propiedad 'modulate' del MeshInstance2D)
9    >| modulate = Color(0.0, 0.4, 1.0, 1.0)
10
11  >| # (Opcional) moverlo un poco para verlo centrado según tu viewport
12  >| position = Vector2(200, 200)
13  >| # scale = Vector2(80, 80) # Si quieres que se vea más grande manteniendo proporciones

```

### 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).



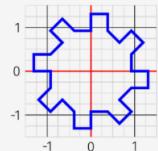
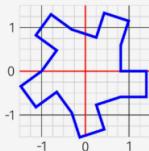
```

1  # res://scripts/Gancho_x4.gd
2  extends Node2D
3  ## Problema 3.13 – Gancho_x4
4  ## Crea 4 copias del gancho (malla del ejercicio 3.12)
5  ## aplicando transformaciones (rotaciones y desplazamientos)
6  ## para formar la cruz que se ve en la figura del enunciado.
7
8  @export var color := Color(0.0, 0.4, 1.0, 1.0) # azul del enunciado
9  @export var draw_scale := 1.0 # escala visual
10 @export var origin_px := Vector2(0, 0) # posición base en pantalla
11
12 func _ready() -> void:
13  >| # Escala y posición general (no afecta a la geometría del gancho)
14  >| scale = Vector2(draw_scale, draw_scale)
15  >| position = origin_px
16
17  >| # Malla base del gancho (desde el autoload del problema 3.12)
18  >| var m := Gancho.gancho()
19
20  >| # --- Creación de los 4 ganchos con sus transformaciones ---
21  >| add_gancho(m, 0.0, Vector2(0, 0)) # 1º gancho (abajo-derecha)
22  >| add_gancho(m, 90.0, Vector2(0, 2)) # 2º girado y desplazado arriba
23  >| add_gancho(m, 180.0, Vector2(-2, 2)) # 3º girado 180º, a la izquierda
24  >| add_gancho(m, 270.0, Vector2(-2, 0)) # 4º girado 270º, abajo
25
26  >|
27  # Función auxiliar (fuera de _ready) para crear y colocar cada gancho.
28  # -----
29  func add_gancho(m: Mesh, rot_deg: float, offset: Vector2) -> MeshInstance2D:
30    >| var mi := MeshInstance2D.new()
31    >| mi.mesh = m
32    >| mi.modulate = color
33    >| mi.rotation_degrees = rot_deg
34    >| mi.position = offset
35    >| add_child(mi)
36    >| return mi

```

### Problema 2.18.

Usa la función del problema anterior para construir estas dos nuevas figuras, en las cuales hay un número variable de instancias de la figura original, dispuestas en círculo (vemos los ejemplos para 5 y 8 instancias, respectivamente).



Para pintar una figura con  $n$  ganchos, los vértices (puntas iniciales y finales) son los radios  $n$ -ésimos de la unidad. Como el primer gancho no empieza en  $(1,0,0)$ , sino que empieza a la mitad, meanen un desfase al círculo mitad.

Vuelv engranaje (int n) {

float alpha, beta;

for (int i=0 ; i < n ; ++i) {

alpha = -(i - 0.5) \* 2.0 \* M\_PI / n

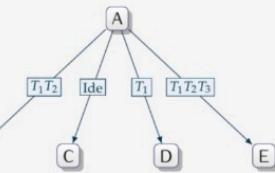
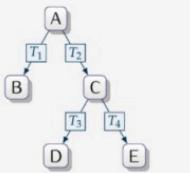
beta = (i + 0.5) \* 2.0 \* M\_PI / n

gancho -> p = ( vec3( cos(alpha) \* sin(beta), 0 ), vec3( cos(beta), sin(beta), 0 ) );

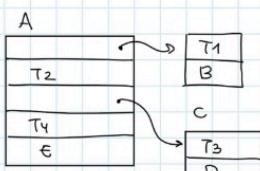
### Problema 2.19.

comenzado en clase (más apuntes)

Dados los dos siguientes grafos de escena sencillos:



Construye los grafos tipo PHIGS equivalentes más sencillos posible (en el sentido de menos nodos posibles). Nota: en el grafo de la derecha, hay que tener en cuenta que algunas de las transformaciones asociadas a los arcos son composiciones distintas de estas tres transformaciones:  $T_1, T_2$  y  $T_3$ .



# Tema 4

## Coolgosity

### 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)

Sesión 4: Modelos de Objetos. Mallas indexadas. - Created 2025-10-21 - Page 81 / 91.

#### 4. Problemas.

### 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  $k = 1024$ ).

Continúa en otra

#### 1. Forma A: Enumeración Espacial (Voxels)

El enunciado pide dividir el cubo que engloba a la esfera en celdas de ancho  $1/k$ . [cite\_start]Si el centro de la celda está dentro de la esfera (radio 0.5), se guarda un 1; si no, un 0 [cite: 869-870]. ☀

gdscript

```
# Genera una representación de voxels (0 o 1)
# k = número de celdas por lado del cubo
func generar_esfera_enumeracion_espatial(k: int) -> Array:
    voxels = [] # Almacenaremos los bits (0 o 1)
    var ancho_celda = 1.0 / float(k)
    var radio_cuadrado = 0.5 * 0.5 # 0.25
    
    # El cubo va de -0.5 a 0.5 en los tres ejes X, Y, Z
    for y in range(k):
        for x in range(k):
            for z in range(k):
                # Si el punto está en el centro de la celda actual
                # (x + 0.5, y + 0.5, z + 0.5) es decir, si no está en el centro del índice x
                var pos_x = -0.5 + (x + 0.5) * ancho_celda
                var pos_y = -0.5 + (y + 0.5) * ancho_celda
                var pos_z = -0.5 + (z + 0.5) * ancho_celda
                var distancia_cuadrada = pos_x*pos_x + pos_y*pos_y + pos_z*pos_z

                # Si la distancia es menor o igual al radio, guardamos 1
                if distancia_cuadrada <= radio_cuadrado:
                    voxels.append(1)
                else:
                    voxels.append(0)

    return voxels
```

#### 2. Forma B: Modelo de Fronteras (Malla Indexada)

[cite\_start]El enunciado pide una rejilla de triángulos siguiendo meridianos y paralelos, con  $k$  vértices por cada uno. Usaremos coordenadas esféricas para generar la Tabla de Vértices y luego conectaremos los puntos para la Tabla de Triángulos. ☀

gdscript

```
# Genera una malla indexada (ArrayMesh)
# k = número de vértices por lado del paralelo (resolución)
func generar_esfera_malla_indexada(k: int) -> ArrayMesh:
    var vértices = PackedVector3Array()
    var índices = PackedInt32Array()
    var radio = 1.0
    
    # --- 1. Generar Tabla de Vértices ---
    # Iteramos latitud (i) y longitud (j)
    for i in range(k + 1):
        var lat = PI*i / k # De 0 a PI (Polos)
        for j in range(k + 1):
            var lon = 2 * PI*j / k # De 0 a 2PI (Vuelta completa)
            
            # Conversión de Esféricas a Cartesianas
            var x = radio * sin(lat) * cos(lon)
            var y = radio * cos(lat) # Eje vertical
            var z = radio * sin(lat) * sin(lon)
            
            vértices.append(Vector3(x, y, z))
    
    # --- 2. Generar Tabla de Triángulos (índices) ---
    # Conectamos el vértice (i, j) con sus vecinos
    for i in range(k + 1):
        for j in range(k + 1):
            var primer_idx = i * (k + 1) * k + j * (k + 1)
            var segundo_idx = primer_idx + 1
            
            # Primer triángulo del cuadrado (face)
            índices.append(primer_idx)
            índices.append(segundo_idx)
            índices.append(primer_idx + 1)
            
            # Segundo triángulo del cuadrado
            índices.append(segundo_idx)
            índices.append(segundo_idx + 1)
            índices.append(primer_idx + 1)
    
    # --- 3. Crear el objeto Mesh para Godot ---
    var arrays = MeshARRAY()
    arrays.resize(Mesh.ARRAY_MAX)
    arrays[Mesh.ARRAY_VERTEX] = vértices
    arrays[Mesh.ARRAY_INDEX] = índices
    
    var mesh = ArrayMesh.new()
    mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES, arrays)
    return mesh
```

#### Enumeración espacial

$$\begin{aligned} &\rightarrow k \text{ celdas por lado} \rightarrow k^3 \text{ celdas} \\ &\rightarrow \text{memoria: } C_0 \cdot k^3 = k^3 \text{ bits} = \frac{k^3}{8} \text{ bytes} \\ &\quad \uparrow \\ &\quad 1 \text{ bit} \\ &\rightarrow k = 16 \rightarrow \frac{16^3}{8} \text{ bytes} = 512 \text{ bytes} \\ &\rightarrow k = 1024 \rightarrow \frac{1024^3}{8} \text{ bytes} = 2^{24} \text{ bytes} \end{aligned}$$

#### Modelo de fronteras :

$$\begin{aligned} &\rightarrow k^2 \text{ vértices}, 2k^2 \text{ triángulos} \\ &\rightarrow \text{cada vértice } 3 \times 4 \text{ bytes} = 12 \text{ bytes} \\ &\rightarrow \text{cada triángulo } 3 \times 4 = 12 \text{ bytes} \quad \left. \right\} 12k^2 + 24k^2 = 36k^2 \text{ bytes} \end{aligned}$$

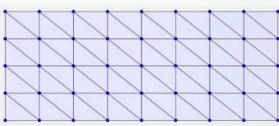
$$\rightarrow k = 16 \rightarrow 36 \cdot 16^2 \text{ bytes}$$

$$\rightarrow k = 1024 \rightarrow 36 \cdot 1024^2 \text{ bytes}$$

Conclusion: para  $k$  pequeños enumeración espacial. Cuando  $k$  grande, llega un punto en el que es más eficiente el de fronteras

### 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)

Sesión 4: Modelos de Objetos. Mallas indexadas.

Created 2025-10-21

4. Problemas.

### 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.

a) Hay  $2 \cdot n \cdot m$  triángulos:

$$1 \text{ triángulo} : 3 \cdot 4 \text{ bytes} = 12 \text{ bytes}$$

$\left. \begin{array}{l} 24 \cdot n \cdot m \text{ bytes} \\ 12 \cdot n \cdot m \text{ bytes} \end{array} \right\} 24 \cdot n \cdot m \text{ bytes}$

Hay  $(n+1)(m+1)$  vértices:

$$1 \text{ vértice} : 3 \times 4 = 12 \text{ bytes}$$

$\left. \begin{array}{l} 12 \cdot (n+1)(m+1) \text{ bytes} \\ 24 \cdot n \cdot m \text{ bytes} \end{array} \right\} 24 \cdot n \cdot m + 12 \cdot (n+1)(m+1) \text{ bytes}$

Tamaño total de la malla:  $24 \cdot n \cdot m + 12 \cdot (n+1)(m+1)$

b)  $M = n = 128$

$$\text{TAMAÑO} : 24 \cdot 128 \cdot 128 \cdot 128 + 12 \cdot 129 \cdot 129 \cdot 129 = 592\,908 \text{ bytes} = 592,908 \text{ KB}$$

$$\textcircled{c) } \frac{n \text{ caras}}{n \text{ vértices}} = \frac{24(n+1)(m+1)}{12nm} \xrightarrow{n,m \rightarrow \infty} 2 \Rightarrow n_C = 2n_V$$

### 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)

①

cada tira tiene  $2n$  triángulos. necesita  $2n+2$  vértices

tiras de triángulos.

cada vértice:  $3 \times 4$  bytes

$$m = n \text{ mallas}$$

tabla:  $4 + 8m$  bytes

$$\text{Memoria: } 4 + 8m + m(2n+2) \cdot 3 \cdot 4 = 4 + 8m + 24mn + 24m$$

$$n = m = 128 \rightarrow 397824 \text{ bytes} \approx 388,5 \text{ KB}$$

$$\text{b) tamaño malla indexada} = \frac{24nm + 12(n+1)(m+1)}{24nm + 32m + 4} \xrightarrow{n,m \rightarrow \infty} \frac{36}{24} = 1.5$$

Malla indexada ocupa 1.5 veces más

c) malla ind:  $(n+1)(m+1)$  vértices

tira: en el peor de los casos, vértices duplicados  $\Rightarrow (2n+2)2 \cdot m \rightarrow 4nm + 4m$  (4 veces más)

Sesión 4: Modelos de Objetos. Mallas indexadas.

Created 2025-10-21

Page 85 / 91

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

Problema 4.3 (continuación):

- Indica que cantidad de memoria ocupa esta representación, en estos dos casos:
  - Como función de  $n$  y  $m$ , en bytes.
  - Suponiendo  $m = n = 128$ , en KB.

(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 2.4.

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:

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

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

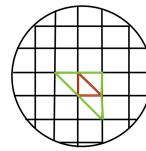
(nótese que, al igual que en el problema anterior, sigue siendo cierto que el número de caras es aproximadamente el doble que el de vértices).

$$\text{Euler : } n_V + n_C = n_A + 2$$

•) Todas las caras son triángulos

$$\left. \begin{array}{l} \\ \end{array} \right\} \frac{3n_C}{2} = n_A$$

•) cada arista comparte la para 2 caras



cada triángulo es adyacente a 3 triángulos, luego, es adyacente a 3 aristas y cada arista es adyacente a exactamente dos triángulos, por lo que :

$$3n_C = 2n_A \Rightarrow n_A = \frac{3n_C}{2}$$

$$n_A = \frac{3n_C}{2} = \frac{3(n_A + 2 - n_V)}{2} \Rightarrow 2n_A = 3n_A + 6 - 2n_V \Rightarrow -n_A = 6 - 2n_V \Rightarrow n_A = 3(n_V - 2)$$

$$n_C = n_A + 2 - n_V = \frac{3n_C}{2} + 2 - n_V \Rightarrow 2n_C = 3n_C + 4 - 2n_V \Rightarrow -n_C = 4 - 2n_V \Rightarrow n_C = 2(n_V - 2)$$

#### 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 lista 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)

Sesión 4: Modelos de Objetos. Mallas indexadas. - Created 2025-10-21 - Page 88 / 91.

4. Problemas.

#### 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.

#### Explicación Teórica para el Examen

Si te piden justificar por qué has usado estos algoritmos, esta es la respuesta basada en la teoría del PDF:

##### [cite\_start]Para el Caso A (No coherente / Abierta) [cite: 777-778]

- Problema: Una arista compartida puede aparecer como  $(1, 5)$  en un triángulo y  $(5, 1)$  en otro ( $o (1, 5)$  dos veces si no hay coherencia). No podemos predecir el orden.
- Estrategia: Normalizamos siempre guardando  $(\min, \max)$ . Usamos un Diccionario porque tiene coste de acceso  $O(1)$  promedio. Esto nos permite detectar si ya hemos procesado esa arista y evitar duplicados.
- Complejidad: Tiempo  $O(n)$  (lineal respecto al número de triángulos) y Memoria  $O(n)$  (para el diccionario).

##### [cite\_start]Para el Caso B (Coherente y Cerrada) [cite: 779-780]

- Problema: Sabemos que es una malla cerrada y coherente. Esto significa matemáticamente que cada arista física es frontera de exactamente dos triángulos y se recorre en sentidos opuestos.
- Estrategia (Optimización): No hace falta un Diccionario. Aprovechamos la propiedad matemática: para cualquier par de números distintos  $u$  y  $v$ , solo una de las relaciones  $u < v$  o  $v < u$  es verdadera.
  - Al recorrer todos los triángulos, encontraremos la arista como par  $(u, v)$  y más tarde (o antes) como par  $(v, u)$ .
  - Si solo guardamos cuando el primero es menor que el segundo ( $u < v$ ), seleccionamos automáticamente una única instancia de la arista y descartamos la inversa.
- Complejidad: Esta solución es más rápida y consume menos memoria que la del Caso A, ya que evitamos el overhead de gestionar las claves del Diccionario (hashing). Es la solución de mínima complejidad posible.

```
extends Node

# CASO A: Orientación NO coherente (o malla abierta)
# No sabemos el orden de los índices ni si la arista se repite.
# SOLUCIÓN: Usar un Diccionario (Hash Map) para filtrar duplicados.
func generar_aristas_caso_a(triangulos: PackedInt32Array) -> Array[Vector2i]:
    var aristas_unicas = {} # Usamos esto como un Set

    # Recorremos la tabla de triángulos de 3 en 3
    for i in range(0, triangulos.size(), 3):
        var t = [triangulos[i], triangulos[i+1], triangulos[i+2]]

        # Generamos las 3 aristas potenciales del triángulo
        for k in 3:
            var u = t[k]
            var v = t[(k + 1) % 3]

            # NORMALIZACIÓN: Como no hay coherencia, la arista (u,v)
            # es la misma que (v,u). Las ordenamos (min, max) para
            # que generen la misma clave en el diccionario.
            var clave = Vector2i(min(u, v), max(u, v))

            # Guardamos en el diccionario (sobrescribe si ya existe)
            aristas_unicas[clave] = true

    # Devolvemos las claves recolectadas
    return aristas_unicas.keys()

# CASO B: Orientación COHERENTE y Malla CERRADA
# Propiedad clave: Cada arista aparece EXACTAMENTE dos veces en la tabla:
# una vez como  $(u, v)$  y otra como  $(v, u)$  en el triángulo vecino.
# SOLUCIÓN: Filtrado directo sin memoria extra (Optimización máxima).
func generar_aristas_caso_b(triangulos: PackedInt32Array) -> Array[Vector2i]:
    var lista_aristas: Array[Vector2i] = []

    for i in range(0, triangulos.size(), 3):
        var t = [triangulos[i], triangulos[i+1], triangulos[i+2]]

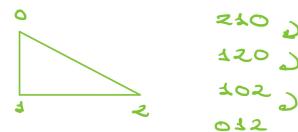
        for k in 3:
            var u = t[k]
            var v = t[(k + 1) % 3]

            # TRUCO DE OPTIMIZACIÓN:
            # Como sabemos que la arista existe en dos versiones:  $(u, v)$  y  $(v, u)$ ,
            # y sabemos que  $u \neq v$ , forzaremos en una versión el primer
            # índice es menor y en la otra es mayor.
            # Solo guardamos la versión donde  $u < v$ .

            if u < v:
                lista_aristas.append(Vector2i(u, v))

            # Si  $u > v$ , ignoramos esta arista porque sabemos que la versión
            #  $(v, u)$  (donde  $v < u$ ) aparecerá (o ya apareció) al procesar
            # el triángulo vecino, y será esa la que guardemos.

    return lista_aristas
```



## 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.

El enunciado especifica que la función recibe un array de `Vector3` (vértices) y un array de `Vector3i` (donde cada `Vector3i` contiene los 3 índices de un triángulo).

gdscript



```
# Calcula el área total de una malla indexada
# Recibe:
# - vertices: Array con las posiciones (Vector3)
# - triangulos: Array donde cada elemento (Vector3i) tiene los 3 índices del triángulo
func calcular_area_malla(vertices: Array[Vector3], triangulos: Array[Vector3i]) -> float:
    var area_total: float = 0.0

    # Recorremos cada triángulo de la lista
    for t in triangulos:
        # 1. Obtener los 3 vértices usando los índices (t.x, t.y, t.z)
        # (Esto es el acceso indirecto propio de las mallas indexadas)
        var A = vertices[t.x]
        var B = vertices[t.y]
        var C = vertices[t.z]

        # 2. Calcular dos vectores que formen los lados del triángulo (desde A)
        var lado_AB = B - A
        var lado_AC = C - A

        # 3. Calcular el producto vectorial (Cross Product)
        # El vector resultante es perpendicular al triángulo y su longitud
        # es igual al área del paralelogramo formado por los dos vectores.
        var vector_perpendicular = lado_AB.cross(lado_AC)

        # 4. El área del triángulo es la mitad de la longitud de ese vector
        var area_triangulo = 0.5 * vector_perpendicular.length()

        # 5. Acumular al total
        area_total += area_triangulo

    return area_total
```

### Explicación rápida para el examen:

1. Entrada: Tienes la geometría ( $x, y, z$  en vértices) y la topología (quién conecta con quién en triángulos).
2. Acceso: Para procesar un triángulo, no lees coordenadas directamente, lees índices (t.x, t.y, t.z) y buscas las coordenadas en la otra tabla.
3. Matemáticas: Usas la fórmula del área con producto cruz:  $rea = \frac{1}{2} \|(B - A) \times (C - A)\|$ .

# Jema 5/

```

1  # problema_5_1.gd
2
3  # Dibuje un cuadrado azul claro con borde azul oscuro
4  # y un triángulo blanco inscrito en su interior.
5  # Los parámetros auto_center y auto_scale permiten
6  # reutilizar la figura dentro de otras escenas (como 5.2).
7
8  extends Node2D
9
10
11  //export var auto_center: bool = true    // contra al ejecutar escena 5.1 sola
12  //export var auto_scale: float = 120.0    // factor de escala visible
13
14
15  func _ready():
16      # --- Colocar posición y escala ---
17      if auto_center:
18          position = get_viewport_rect().size * 0.5
19          scale = Vector2(auto_scale, auto_scale)
20
21      # --- Cuadrado centrado (lados = 2) ---
22      var s := 1.0
23      var square := PackedVector2Array([
24          Vector2(-s, -s),
25          Vector2( s, -s),
26          Vector2( s,  s),
27          Vector2(-s,  s),
28          1])
29      _add_polygon2d(square, *Color(0.65, 0.82, 1.0, 1.0), *Color(0.0, 0.0, 0.0, 1.0))
30
31      # --- Triángulo inscrito (centrado dentro) ---
32      var tri := PackedVector2Array([
33          Vector2(-0.35, -0.6),
34          Vector2( 0.35, -0.6),
35          Vector2( 0.0,  0.5),
36          1])
37      _add_polygon2d(tri, *Color(1, 1, 1, 1), *Color(0.0, 0.0, 0.4, 1.0))
38
39
40  func _add_polygon2d(points: PackedVector2Array, fill_color: Color, edge_color: Color) -> void:
41      # --- Retorno: Puedo triangulando desde el vértice 0 ---
42      var verts := PackedVector3Array()
43      var cols := PackedColorArray()
44      for i in range(1, points.size() - 1):
45          verts.push_back(points[0])
46          verts.push_back(points[i])
47          verts.push_back(points[i + 1])
48      for _j in range(2):
49          col.push_back(fill_color)
50
51      var arrays := []
52      arrays.resize(Math.ARRAY_MAX)
53      arrays[Math.ARRAY_VERTEX] = verts
54      arrays[Math.ARRAY_COLOR] = cols
55
56      var mesh := ArrayMesh.new()
57      mesh.add_surface_from_arrays(Math.PRIMITIVE_TRIANGLES, arrays)
58
59      # --- Borde: LINE_STRIP cerrado ---
60      var edge := PackedVector2Array(points)
61      edge.push_back(points[0])
62      var edge_cols := PackedColorArray()
63      for _i in edge.size():
64          edge_cols.push_back(edge_color)
65
66      var arrays2 := []
67      arrays2.resize(Math.ARRAY_MAX)
68      arrays2[Math.ARRAY_VERTEX] = edge
69      arrays2[Math.ARRAY_COLOR] = edge_cols
70      mesh.add_surface_from_arrays(Math.PRIMITIVE_LINE_STRIP, arrays2)
71
72      # --- Instanciar la malla y añadirlo ---
73      var m1 := MeshInstance2D.new()
74      m1.mesh = mesh
75      add_child(m1)

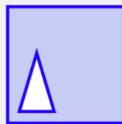
```

## Escena simple



## Problema 5.1:

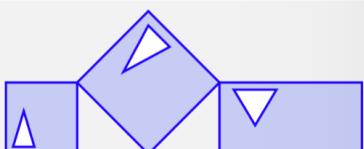
Implementa un proyecto cuya escena principal tenga un de tipo **Node2D** con varios nodos hijos, que formen la figura con un cuadrado de lado 2, centrado en el origen, y con un triángulo inscrito. El cuadrado debe estar lleno de azul claro, el triángulo de blanco, y las aristas deben verse de color azul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



```

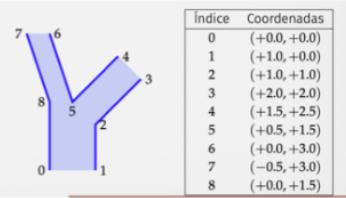
1  # res://scripts/problema_5_2_control.gd
2  extends Node2D
3
4  # Este script posiciona y rota las tres figuras hijas (del 5.1)
5  # para formar la composición del problema 5.2.
6
7  //export var auto_scale: float = 120.0    // factor de escala visible
8
9
10 func _ready() -> void:
11     # Referencias a los hijos (asegúrate de que se llamen igual)
12     var izq: Node2D = $Figura_Izq
13     var centro: Node2D = $Figura_Centro
14     var der: Node2D = $Figura_Der
15
16     # --- FIGURA IZQUIERDA ---
17     # Más pequeña, sin rotación.
18     izq.scale = Vector2(100, 100)
19     izq.rotation_degrees = 0
20     izq.position = Vector2(200, 400)
21
22     # --- FIGURA CENTRAL ---
23     # Más grande, rotada 45°.
24     centro.scale = Vector2(145, 145)
25     centro.rotation_degrees = 45
26     # La colocamos manualmente un poco arriba y al centro.
27     centro.position = Vector2(500, 500)
28
29     # --- FIGURA DERECHA ---
30     # Misma escala que la izquierda, sin rotación o con rotación opuesta si quieras variar.
31     der.scale = Vector2(170, 100)
32     der.rotation_degrees = -180
33     # Queda tocando el vértice derecho del rombo central.
34     der.position = Vector2(500, 400)

```

## 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).



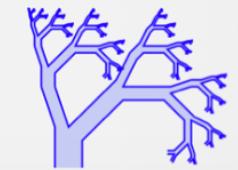
Session 5: Modelos Jerárquicos - Created 2025-10-14 - Page 60 / 63.

4. Problemas.

## Figura recursiva

### Problema 5.4:

Implementa otro proyecto Godot que use la función del problema anterior para otra función, `Árbol(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`.



### Solución Problema 5.4: Función crear\_arbol (Recursiva)

El problema 5.4 pide usar la función anterior para crear un árbol recursivo.

Basándonos en la teoría de grafos de escena, un modelo jerárquico permite añadir hijos que heredan las transformaciones del padre.  $\varphi +1$

#### Lógica de diseño:

Observando la forma del tronco, tiene dos "puntas" claras donde deberían crecer las siguientes ramas para formar el árbol:

1. Punta Superior: En la coordenada (0, 3) (índices 6-7).
2. Punta Derecha: En la coordenada (2, 2) (índices 3-4).

Vamos a colocar versiones más pequeñas (scale) y rotadas (rotation) del árbol en esos puntos.

#### Código GDScript:

```
gdscript

func crear_arbol(nivel: int) -> Node2D:
    # 1. Creamos la base (el tronco)
    var nodo_actual = crear_tronco()

    # CASO BASE: Si el nivel es 0, no añadimos más ramas, devolvemos solo el tronco.
    if nivel <= 0:
        return nodo_actual

    # 2. RECURSIVIDAD: Crear las ramas hijas (nivel - 1)
    # --- Rama Superior (Cree desde la punta de arriba: 0, 3) ---
    var rama_arriba = crear_arbol(nivel - 1)
    rama_arriba.position = Vector2(0, 3) # Movemos al extremo superior
    rama_arriba.scale = Vector2(0.7, 0.7) # Hacemos más pequeña
    rama_arriba.rotation_degrees = -10 # La rotación alejada o fija
    nodo_actual.add_child(rama_arriba) # !Importante! [<span>_start</span>]<span>_end</span> Añadir como hijo
    return nodo_actual

    # --- Rama Derecha (Cree desde la punta derecha: 2, 2) ---
    var rama_derecha = crear_arbol(nivel - 1)
    rama_derecha.position = Vector2(2, 2) # Movemos al extremo derecho
    rama_derecha.scale = Vector2(0.7, 0.7) # Hacemos más pequeña
    rama_derecha.rotation_degrees = -45 # Rotamos para que siga la dirección de la rama
    nodo_actual.add_child(rama_derecha)
    return nodo_actual
```

### Solución Problema 5.3: Función crear\_tronco

El objetivo es crear una función que devuelva un `Node2D` que contenga la geometría.

#### Código GDScript:

##### gdscript

```
extends Node2D

# Definimos las coordenadas tal cual aparecen en la tabla del PDF (Problema 5.3)
[span l="1" style="color: #ccc;">const COORDENADAS TRONCO = [
    Vector2(0.0, 0.0), # 0
    Vector2(1.0, 0.0), # 1
    Vector2(1.0, 1.0), # 2
    Vector2(2.0, 2.0), # 3 (Punta rama derecha)
    Vector2(1.5, 1.5), # 4
    Vector2(0.5, 1.5), # 5 (Intersección interna)
    Vector2(0.0, 3.0), # 6 (Punta rama superior)
    Vector2(-0.5, 3.0), # 7
    Vector2(0.0, 1.5) # 8
]

func crear_tronco() -> Node2D:
    # 1. Nodo Raiz de la pieza
    var raiz = Node2D.new()
    raiz.name = "Tronco"

    # 2. Nodo para el Relleno (Color sólido)
    var poligono = Polygon2D.new()
    poligono.polygon = PackedVector2Array(COORDENADAS TRONCO)
    poligono.color = Color.SADDLE_BROWN # Marrón madera
    raiz.add_child(poligono)

    # 3. Nodo para el Bordo (Líneas)
    # Para que el borde se cierre, repetimos el primer punto al final
    var puntos_borde = COORDENADAS TRONCO.duplicate()
    puntos_borde.append(COORDENADAS TRONCO[0])

    var linea = Line2D.new()
    linea.points = PackedVector2Array(puntos_borde)
    linea.width = 0.05 # Línea fina
    linea.default_color = Color.BLACK
    raiz.add_child(linea)

    return raiz
```

#### Cómo probarlo en tu escena \_ready

Para ver el resultado, añade este código en la función `_ready` de tu script principal:

##### gdscript

```
func _ready():
    # Crear un árbol de nivel 4 (complejidad media)
    var mi_arbol = crear_arbol(4)

    # Lo colocamos en el centro de la pantalla para verlo bien
    mi_arbol.position = Vector2(400, 500)
    mi_arbol.scale = Vector2(50, -50) # Escala grande y Y invertida porque en Godot Y crece hacia abajo
    add_child(mi_arbol)
```

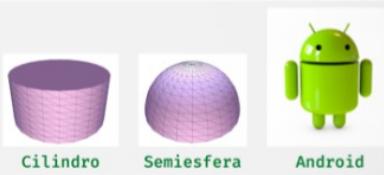
#### Puntos Clave Teóricos (para el examen)

1. Modelo Jerárquico: Al mover el nodo raíz (`mi_arbol`), se mueven todas las ramas recursivamente porque son nodos hijos.  $\varphi$
2. [cita\_start]Transformaciones: La scale y rotation de los hijos son relativas a su padre (Tronco), no al mundo global [cita: 1011-1014].
3. Orden de Dibujado: Primero se dibuja el padre y luego los hijos encima (aunque aquí no importa mucho porque son del mismo color).

## 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.



```
1 ## Nombre: Raúl, Apellidos: Martínez Bustos, Titulación: GIADE
2 ## email: raulmtz@correo.uvgr.es, DNI o pasaporte: 20531007-V
3
4
5 # res://scripts/android.gd
6 extends Node3D
7
8
9 # ===== AJUSTES GENERALES (detalle y color) =====
10
11 Oexport var lados_cil:int = 64          # detalle cilindros
12 Oexport var u_sem:int = 64              # meridianos semiesfera
13 Oexport var v_sem:int = 32              # paralelos semiesfera
14 Oexport var verde: Color = #Color(0.56, 0.79, 0.22)
15 Oexport var negro: Color = #Color(0,0,0)
16
17
18 # ===== CUERPO =====
19 Oexport var cuerpo_radio: float = 1.8
20 Oexport var cuerpo_altura: float = 2.0
21 Oexport var cuerpo_pos: Vector3 = Vector3(0, 0, 0)
22
23
24 # ===== CABEZA (semiesfera superior) =====
25 Oexport var cabeza_radio: float = 1.8
26 Oexport var cabeza_pos_y: float = 2      # se asienta justo encima del cuerpo
27
28
29 # ===== BRAZOS (cilindros horizontales) =====
30 Oexport var brazo_radio: float = 0.38
31 Oexport var brazo_largo: float = 1.88    # se aplica sobre el eje Y del cilindro
32 Oexport var brazo_altura_y: float = 1.10
33 Oexport var brazo_sep_x: float = 1.10   # desplazamiento sX (abre/cierra brazos)
34
35
36 # ===== PIERNAS (cilindros verticales) =====
37 Oexport var pierna_radio: float = 0.28
38 Oexport var pierna_largo: float = 2.20
39 Oexport var pierna_sep_x: float = 0.58
40 Oexport var pierna_base_y: float = -1
41
42
43 # ===== ANTENAS (cilindros finos, inclinados) =====
44 Oexport var antena_radio: float = 0.08
45 Oexport var antena_largo: float = 1.18
46 Oexport var antena_inclin_deg: float = 25.8  # abre/cierra hacia los lados
47 Oexport var antena_altura_y: float = 2.75
48 Oexport var antena_sep_x: float = 0.43
49
50
51 # ===== OJOS (discos que miran al +Z) =====
52 Oexport var ojo_radio: float = 0.08
53 Oexport var ojo_grosor: float = 0.08       # "altura" del cilindro
54 Oexport var ojo_sep_x: float = 0.35
55 Oexport var ojo_altura_y: float = 2.10
56 Oexport var ojo_salida_z: float = 0.95     # cuanto sobresalen hacia +Z
57
58
59 # ====== CÓDIGO DE CREACIÓN ======
60
61 func _ready() -> void:
62     # 1) Primitivas
63     var an_cil1: ArrayMesh = Primitivas3D.crear_cilindro_y(lados_cil, 2.0, 1.0) # base unidad
64     var an_hem1: ArrayMesh = Primitivas3D.crear_semiesfera_y(u_sem, v_sem, 1.0)
65
66     # 2) CUERPO
67     var cuerpo := Primitivas3D.instanciar(an_cil1, verde)
68     cuerpo.transform = Transform3D().scaled(Vector3(cuerpo_radio, cuerpo_altura+0.5, cuerpo_radio))
69     # nota: el cilindro base mide 2 en Y; escalar Y por (altura/2) ± altura_total
70     cuerpo.position = cuerpo_pos
71     add_child(cuerpo)
72
73     # 3) CABEZA (semiesfera superior centrada en XZ)
74     var cabeza := Primitivas3D.instanciar(an_hem1, verde)
75     var t_cab := Transform3D().scaled(Vector3(cabeza_radio, cabeza_radio, cabeza_radio))
76     t_cab.origin = Vector3(0, cabeza_pos_y, 0) + cuerpo_pos
77     cabeza.transform = t_cab
78     add_child(cabeza)
79
80     # 4) BRAZOS (cilindros horizontales, eje YX y escalado)
81     var t_bra_base := Transform3D().rotated(Vector3(0,0,1), PI*0.5) # gira cilindro para que apunte a X
82     t_bra_base = t_bra_base.scaled(Vector3(brazo_radio, brazo_largo+0.5, brazo_radio))
83     # Izquierdo
84     var bra_i_1 := Primitivas3D.instanciar(an_cil1, verde)
85     bra_i_1.transform = t_bra_base
86     bra_i_1.position = cuerpo_pos + Vector3(-brazo_sep_x, brazo_altura_y, 0)
87     add_child(bra_i_1)
88     # Derecho
89     var bra_d_1 := Primitivas3D.instanciar(an_cil1, verde)
90     bra_d_1.transform = t_bra_base
91     bra_d_1.position = cuerpo_pos + Vector3(+brazo_sep_x+0.6, brazo_altura_y, 0)
92     add_child(bra_d_1)
93
94     # 5) PIERNAS (verticales bajo el cuerpo)
95     var t_pier_base := Transform3D().scaled(Vector3(pierna_radio, pierna_largo+0.5, pierna_radio))
96     var pi_1 := Primitivas3D.instanciar(an_cil1, verde)
97     pi_1.transform = t_pier_base
98     pi_1.position = cuerpo_pos + Vector3(-pierna_sep_x, pierna_base_y, 0)
99     add_child(pi_1)
100
101    var pi_d_1 := Primitivas3D.instanciar(an_cil1, verde)
102    pi_d_1.transform = t_pier_base
103    pi_d_1.position = cuerpo_pos + Vector3(+pierna_sep_x, pierna_base_y, 0)
104    add_child(pi_d_1)
105
106    # 6) ANTENAS (finas, inclinadas hacia fuera)
107    var t_ant_base := Transform3D().rotated(Vector3(0,0,1), deg_to_rad(-antena_inclin_deg))
108    # Izquierda (inclina hacia -X)
109    var ant_i_1 := Primitivas3D.instanciar(an_cil1, verde)
110    ant_i_1.transform = t_ant_base.rotated(Vector3(0,0,1), deg_to_rad(-antena_inclin_deg))
111    ant_i_1.position = cuerpo_pos + Vector3(-antena_sep_x, antena_altura_y, 0)
112    add_child(ant_i_1)
113    # Derecha (inclina hacia +X)
114    var ant_d_1 := Primitivas3D.instanciar(an_cil1, verde)
115    ant_d_1.transform = t_ant_base.rotated(Vector3(0,0,1), deg_to_rad(+antena_inclin_deg))
116    ant_d_1.position = cuerpo_pos + Vector3(-antena_sep_x, antena_altura_y, 0)
117    add_child(ant_d_1)
118
119    # 7) OJOS (discos mirando al frente + eje YZ)
120    var t_ojo_base := Transform3D().rotated(Vector3(1,0,0), PI*0.5) \
121        .scaled(Vector3(ojo_radio, ojo_grosor*0.5, ojo_radio))
122    var ojo_i_1 := Primitivas3D.instanciar(an_cil1, negro)
123    ojo_i_1.transform = t_ojo_base
124    ojo_i_1.position = cuerpo_pos + Vector3(-ojo_sep_x, ojo_altura_y, +ojo_salida_z)
125    add_child(ojo_i_1)
126
127    var ojo_d_1 := Primitivas3D.instanciar(an_cil1, negro)
128    ojo_d_1.transform = t_ojo_base
129    ojo_d_1.position = cuerpo_pos + Vector3(+ojo_sep_x, ojo_altura_y, +ojo_salida_z)
130    add_child(ojo_d_1)
```

El marco de referencia de vista  $\mathbb{V}$ , se define a partir de los siguientes parámetros:

- $\hat{o}_{\text{vc}}$  = posición del observador: es el punto del espacio foco de la proyección, donde estaría situado el observador ficticio que contempla la escena (projection reference point, PRP)
- $\hat{n}$  = vector normal: vector libre perpendicular al plano de visión (plano ficticio donde se proyecta la imagen perpendicular al eje óptico de la cámara virtual) (view plane normal, VPN).
- $\hat{a}$  = punto de atención: punto en el eje óptico, por tanto se va a proyectar en el centro de la imagen (look-at-point).
- $\hat{u}$  = vector hacia arriba: es un vector libre que indica una dirección que el observador ve proyectada en vertical en la imagen (apuntando hacia arriba) (view-up vector, VUP).

De los tres parámetros  $\hat{o}_{\text{vc}}$ ,  $\hat{n}$  y  $\hat{a}$  sólo hay que especificar dos, ya que no son independientes, se cumple:  $\hat{o}_{\text{vc}} = \hat{a} + \hat{n}$ .

## Cámara que sigue un objetivo

### Problema 6.1:

Escribe el código GDScript para adjuntar a un nodo de tipo Camera3D, de forma que en cada frame la cámara apunte a un objeto móvil objetivo (por ejemplo un coche), con estos requerimientos:

- La posición y el vector de velocidad del objetivo (en coordenadas de mundo) se podrán obtener con dos funciones globales, llamadas `objetivo.posicion()` y `objetivo.velocidad()`, ambas devuelven un objeto de tipo `Vector3`.
- La cámara debe situarse detrás del objetivo, de forma que el punto devuelto por `objetivo.posicion()` se proyecte en el centro del viewport, y además la cámara esté situada 3 unidades en horizontal por detrás del objetivo, y 2 unidades por encima (en el eje Y).

### Problema 6.2:

Supongamos una escena que contiene una representación visible del marco de coordenadas del mundo como tres flechas (roja X, verde Y y azul Z), como ocurre en las prácticas. Queremos visualizar esa escena en pantalla, de forma que:

- El eje Y aparezca vertical, hacia arriba, el eje X horizontal, hacia la derecha, el eje Z horizontal, hacia la izquierda (los ejes X y Z se visualizan con la misma longitud aparente).
- El punto de coordenadas  $(0, 0, 0)$  (aparece como un disco de color morado en la figura) debe aparecer en el centro del viewport
- El observador (foco de la proyección) estará a 3 unidades de distancia del punto  $(0, 0, 0)$

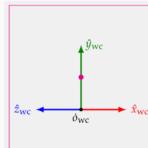
(continua en la siguiente transparencia).

Sesión 6: Transformación de vértices      Created 2025-10-28      Page 79 / 87  
5. Problemas.  
5.1. Transformación de vista.

## Parámetros para una vista concreta (2/3)

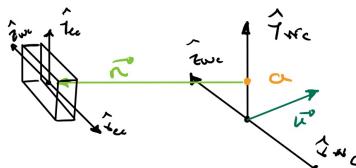
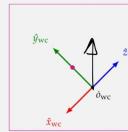
### Problema 6.2 (continuación):

Escribe unos valores que podríamos usar para  $a$ ,  $u$  y  $n$  de forma que se cumplen estos requisitos. En la figura se observa una vista esquemática de como quedaría la figura en un viewport cuadrado, no necesariamente a escala.



### Problema 3.2.

Repite el problema anterior, pero ahora para esta vista:



# Tema 6

Objetivo: Crear un script para una cámara que siga a un objetivo, situándose detrás y arriba.

## Fundamento Teórico:

La transformación de vista requiere definir una posición del observador ( $\hat{o}_{\text{vc}}$ ), un punto de atención ( $\hat{a}$ ) y un vector "arriba" ( $\hat{u}$ ). En Godot, esto se puede manejar usando el método `look_at_from_position`.

## Solución en GDScript:

```
gdscript
```

```
extends Camera3D

# Referencia al nodo objetivo (debe asignarse en el editor o buscarse)
@export var nodo_objetivo: Node3D

func proces(delta):
    if nodo_objetivo:
        # 1. Obtener datos globales del objetivo
        # Suponemos que el nodo tiene estas funciones como indica el enunciado
        var pos_obj = nodo_objetivo.objetivo.posicion() # Vector3
        var vel_obj = nodo_objetivo.objetivo.velocidad() # Vector3

        # 2. Calcular la posición de la cámara (observador)
        # Requisito: 3 unidades detrás (contrario a la velocidad) y 2 arriba (Y)
        # Normalizamos la velocidad para tener solo dirección
        var direccional = -vel_obj.normalized()
        var offset_detrás = direccional * delta * -3.0
        var offset_arriba = Vector3(0, 2, 0)

        var pos_cámara = pos_obj + offset_detrás + offset_arriba

        # 3. Orientar la cámara
        # El punto de atención es la posición del objetivo
        # Usamos Vector3.UP como vector auxiliar "arriba" estándar
        look_at_from_position(pos_cámara, pos_obj, Vector3.UP)
```

Sabemos que :

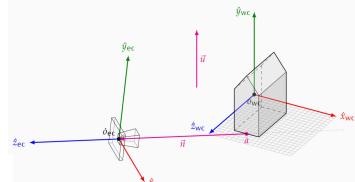
- $\hat{a}$  es el punto de atención
- $\hat{u}$  indica la dirección en la que el observador proyecta en vertical la imagen
- $\hat{n}$  vector libre perpendicular al plano de visión

Como el punto mirado debe aparecer en el centro del viewpoint, ese será nuestro punto de atención  $\Rightarrow \hat{a} = (0, 0, 0, 0)$

Sabemos que  $\hat{n}$  es el vector que va del punto de atención  $\hat{a}$  en el mundo real al punto  $\hat{o}_{\text{vc}}$ , que es el punto del espacio donde estaría situado el observador que contempla la escena, en nuestro proyecto. Como el observador se encuentra a 3 unidades:

$$3 = \|\hat{n}\| = \|\hat{o}_{\text{vc}} - \hat{a}\| = \sqrt{(\hat{o}_x - 0)^2 + (\hat{o}_y - 0)^2 + (\hat{o}_z - 0)^2} = \sqrt{0x^2 + (0y - 0)^2 + 0z^2}$$

Si tenemos  $0y = 0.5$  (como no nos ponen restricciones sobre  $0x$ , tomamos el que queramos), tenemos que  $3 = \sqrt{0x^2 + 0.5^2}$ . Por otro lado, el enunciado nos dice que los ejes  $x$  e  $z$  se visualizan con la misma longitud, luego  $0x = 0z$ , lo que nos lleva a que  $0x = 0z$  (por como se define la matriz de vista  $V$ )  
 $\Rightarrow 3 = \sqrt{20.5^2} = \sqrt{2} \cdot 0x \Rightarrow 0x = \frac{\sqrt{2}}{2} \Rightarrow \hat{n} = \left( \frac{\sqrt{2}}{2}, 0, \frac{\sqrt{2}}{2} \right)$   
 Al igual que en las prácticas, proyectamos en vertical siguiendo el eje  $Y \Rightarrow \hat{u} = (0, 1, 0)$ .



En este caso, el punto de atención sigue siendo el mismo  $(\hat{a} = (0, 0.5, 0))$

Por otro lado, como se invierten los ejes  $X$  e  $Z$ ,  $Y$ , además,

se mantienen las proporciones,

tenemos que  $\hat{n} = \left( -\frac{3}{\sqrt{2}}, 0, \frac{3}{\sqrt{2}} \right)$

sigue siendo el mismo, ya que es el vector perpendicular al plano de visión y este no cambia  $\hat{n} = \left( \frac{3}{\sqrt{2}}, 0, \frac{3}{\sqrt{2}} \right)$

Como podemos ver en la ilustración de la izquierda,  $\hat{u} = (1, 1, 0)$

### Problema 3.3.

Escribe el código para calcular los vectores de coordenadas  $x_{ec}$ ,  $y_{ec}$ ,  $z_{ec}$  y  $o_{ec}$  que definen el marco de vista a partir de los vectores de coordenadas  $a$ ,  $u$  y  $n$  (todos estos vectores de coordenadas son de tipo `vec3`).

*Será pasar esas fórmulas a código ↗*

```
extends Node

# Declaración de variables miembro (equivalente a las globales del ejemplo)
var oec: Vector3
var u: Vector3
var n: Vector3
var zec: Vector3
var yec: Vector3

# Función traducida
# Nota: Cambié el tercer argumento de 'm' a 'n' para que coincida con el uso interno
func calcular_zec(u: Vector3, n: Vector3) void:
    # Suma de vectores
    oec = n + a

    # Normalizar (En Godot es .normalized())
    zec = n.normalized()

    # Producto Cruz (En Godot es .cross()) y luego normalizar
    # ProductoVectorial(u, n) -> u.cross(n)
    xec = u.cross(n).normalized()

    # Producto Cruz final
    yec = zec.cross(xec)
```

### Construcción de la matriz de vista

3.3  
1)

#### Problema 6.4:

Escribe el código GDScript para calcular los vectores de coordenadas  $x_{ec}$ ,  $y_{ec}$ ,  $z_{ec}$  y  $o_{ec}$  que definen el marco de vista a partir de los vectores de coordenadas  $a$ ,  $u$  y  $n$  (todos estos vectores de coordenadas de mundo, en objetos de tipo `Vector3`).

#### Problema 6.5:

Partiendo de los vectores de coordenadas  $x_{ec}$ ,  $y_{ec}$ ,  $z_{ec}$  y  $o_{ec}$  que se calculan en el problema anterior, escribe el código que calcula explícitamente la matriz de vista, es una variable de tipo `Transform3D`.

#### Problema 6.6: Cálculo explícito de la Matriz de Vista

Objetivo: Construir la `Transform3D` que representa la matriz de vista ( $V$ ).

##### Fundamento Teórico:

La matriz de vista  $V$  convierte coordenadas de mundo a coordenadas de ojo. Es la inversa de la transformación de la cámara.

La fórmula explícita se da en la diapositiva 19:

$$V = \begin{pmatrix} \hat{x}_x & \hat{x}_y & \hat{x}_z & d_x \\ \hat{y}_x & \hat{y}_y & \hat{y}_z & d_y \\ \hat{z}_x & \hat{z}_y & \hat{z}_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Donde  $d_x = -\hat{x}_{ec} \cdot \hat{o}_{ec}$  etc.

##### Nota técnica de Godot:

Transform3D en Godot almacena la base (columnas, ejes) y el origen. Sin embargo, matemáticamente se comporta como la matriz de transformación. La matriz de vista no es la transform de la cámara, sino su inversa. Si debemos construirla explícitamente como una matriz matemática de proyección  $4 \times 4$  (`Transform`) que multiplique puntos:

$$\begin{aligned} a &= x_{ec} & d_x &= -x_{ec} \cdot \hat{o}_{ec} \\ b &= y_{ec} & d_y &= -y_{ec} \cdot \hat{o}_{ec} \\ c &= z_{ec} & d_z &= -z_{ec} \cdot \hat{o}_{ec} \end{aligned}$$

$$\text{Como } \vec{n} = \hat{o}_{ec} - \vec{a} \Rightarrow \hat{o}_{ec} = \vec{n} + \vec{a}$$

$$\hat{z}_{ec} = \frac{\vec{n}}{\|\vec{n}\|} \quad (\text{eje Z paralelo a VPN, normalizado})$$

$$\hat{x}_{ec} = \frac{\vec{u} \times \vec{n}}{\|\vec{u} \times \vec{n}\|} \quad (\text{eje X perpendicular a VPN y VUP, normalizado})$$

$$\hat{y}_{ec} = \hat{z}_{ec} \times \hat{x}_{ec} \quad (\text{eje Y perpendicular a los otros dos})$$

Objetivo: Código para calcular  $\hat{x}_{ec}$ ,  $\hat{y}_{ec}$ ,  $\hat{z}_{ec}$ ,  $\hat{o}_{ec}$  a partir de  $a$ ,  $u$ ,  $n$ .

##### Fundamento Teórico:

Se sigue el procedimiento de ortonormalización descrito en la diapositiva 16.

- $\hat{z}_{ec}$  es  $\vec{n}$  normalizado.
- $\hat{x}_{ec}$  es el producto cruz normalizado de  $\vec{u}$  y  $\hat{z}_{ec}$ .
- $\hat{y}_{ec}$  es el producto cruz de  $\hat{z}_{ec}$  y  $\hat{x}_{ec}$ .
- $\hat{o}_{ec} = \vec{a} + \vec{n}$  (según la definición  $\vec{o} = \vec{a} + \vec{n}$  en diapositiva 15, asumiendo que el input  $n$  aquí representa el vector desplazamiento VPN y no solo la dirección).

##### Solución en GDScript:

###### gdscript

```
func calcular_marco_vista(a: Vector3, u: Vector3, n: Vector3):
    # n es el vector desde 'a' hasta el observador (VPN)
    # x_ec es perpendicular a u y z (VUP x VPN)
    var z_ec: Vector3 = n.normalized() #[span_15](end_span)
    # x_ec es perpendicular a u y z (VUP x VPN)
    var x_ec: Vector3 = u.cross(z_ec).normalized() #[span_16](end_span)
    # y_ec es perpendicular a u y x (VPN x VUP)
    var y_ec: Vector3 = z_ec.cross(x_ec) #[span_17](end_span)
    # Nota: y_ec ya sale unitario si x y z son unitarios y ortogonales
    # o_ec es perpendicular a u y z (VPN x VUP)
    var o_ec: Vector3 = a + n #[span_18](end_span)
    return [x_ec, y_ec, z_ec, o_ec]
```

##### Solución en GDScript:

###### gdscript

```
func calcular_matriz_vista(x_ec: Vector3, y_ec: Vector3, z_ec: Vector3, o_ec: Vector3) -> Transform3D:
    # [span_20](start_span)Calculamos los términos de traslación (productos punto
    negativos)#[span_20](end_span)
    var dx = -x_ec.dot(o_ec)
    var dy = -y_ec.dot(o_ec)
    var dz = -z_ec.dot(o_ec)

    # Construimos la Transform3D
    # CUIDADO: Transform3D(Vector3 x_axis, Vector3 y_axis, Vector3 z_axis, Vector3 origin)
    # constructor espera las COLUMNAS de la matriz.
    # La matriz V tiene en sus FILAS a los vectores x_ec, y_ec, z_ec.

    # Columna 0 de V: (x_ec.x, y_ec.x, z_ec.x)
    var col_x = Vector3(x_ec.x, y_ec.x, z_ec.x)

    # Columna 1 de V: (x_ec.y, y_ec.y, z_ec.y)
    var col_y = Vector3(x_ec.y, y_ec.y, z_ec.y)

    # Columna 2 de V: (x_ec.z, y_ec.z, z_ec.z)
    var col_z = Vector3(x_ec.z, y_ec.z, z_ec.z)

    # Columna 3 (Origen) de V: (dx, dy, dz)
    var origin = Vector3(dx, dy, dz)

    var view_matrix = Transform3D(col_x, col_y, col_z, origin)
    return view_matrix
```

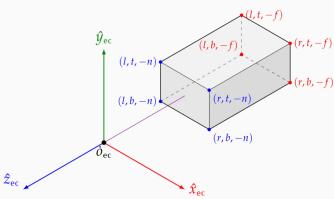
## Ajuste dinámico

### Problema 6.6:

En una copia independiente del código de prácticas, modifica el nodo de la cámara orbital simple para conseguir que el fov mínimo (vertical u horizontal) sea siempre de  $75^\circ$ , serviría, por ejemplo, para ver el cubo de las prácticas siempre completo independientemente del ancho y alto de la ventana (ver transparencia 58 y siguiente). Para ello:

1. Añadir al script del nodo de cámara una función que se ejecute siempre que se redimensione la ventana (y al inicio), en esa función:
2. obtener el tamaño (alto y ancho) del viewport,
3. calcular la relación de aspecto (ancho/alto)
4. usar ajuste de la proyección en vertical si el viewport es más ancho que alto, y ajuste en horizontal en caso contrario.

$$l \leq x_{ec} \leq r \quad b \leq y_{ec} \leq t \quad -f \leq z_{ec} \leq -n$$



### Problema 3.5.

Queremos visualizar una escena con mallas indexadas de la cual sabemos que tiene todos los vértices dentro de un cubo de lado  $s$  unidades cuyo centro es el punto de coordenadas del mundo  $c = (c_x, c_y, c_z)$ .

Para construir la matriz de vista, se sitúa el observador en el punto  $\text{O}_{ec} = (c_x, c_y, c_z + s + 2)$ , el punto de atención  $a$  se hace igual a  $c$  (el centro del cubo se ve en el centro de la imagen), y el vector  $u$  es  $(0, 1, 0)$ . Se visualizará en un viewport cuadrado.

(continua en la siguiente página)

### Problema 6.6: Ajuste dinámico del FOV

Objetivo: Mantener siempre un FOV mínimo de  $75^\circ$  independientemente de si la ventana se ensancha o alarga.

#### Fundamento Teórico:

- Si  $\text{width} < \text{height}$  (ventana alta), el FOV horizontal se reduce si usamos `KEEP_HEIGHT`. Debemos usar `KEEP_WIDTH` para fijar el ángulo horizontal.
- Si  $\text{width} > \text{height}$  (ventana ancha), el FOV vertical se reduce si usamos `KEEP_WIDTH`. Debemos usar `KEEP_HEIGHT` para fijar el ángulo vertical.
- Referencia: Diapositivas 59 y 60. ↗ +1

#### Solución Conceptual:

1. Calcular relación de aspecto  $r = \text{ancho}/\text{alto}$ .
2. Si  $r < 1$  (más alta que ancha), fijar el FOV en horizontal (`KEEP_WIDTH`).
3. Si  $r \geq 1$  (más ancha que alta), fijar el FOV en vertical (`KEEP_HEIGHT`).

#### Solución en GDScript:

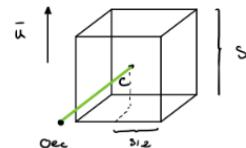
gdscript

```
func _process(delta):
    if width < height: # Si la ventana es más alta que ancha
        var aspect = float(vp_size.x) / float(vp_size.y) # Relación de aspecto
        # Fov deseado 75 grados
        if aspect < 1.0:
            # Ventana vertical: fijamos el ancho para no perder visión lateral
        else:
            # Ventana apaisada: fijamos la altura (comportamiento por defecto)
    # Fov deseado 75 grados
    if width > height: # Si la ventana es más ancha que alta
        # Fijar el FOV en vertical
    else:
        # Fijar el FOV en horizontal
```

Queremos construir la matriz de proyección perspectiva  $Q$  de forma que se cumplan estos requerimientos:

1. No se recorta ningún triángulo.
2. El tamaño aparente de los objetos (proyectados en pantalla) es el mayor posible.
3. El valor del parámetro  $n$  es el mayor posible.
4. El valor del parámetro  $f$  es el menor posible.
5. Los objetos no aparecen deformados.

Con estos requerimientos, indica como calcular los valores  $l, r, t, b, n$  y  $f$  (para obtener la matriz  $Q$  de proyección), en función de  $s$  y  $(c_x, c_y, c_z)$ .



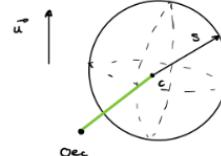
Como tenemos un cubo de lado  $s$ , se trata de proyección ortográfica. Por otro lado, tenemos que  $a = c = (c_x, c_y, c_z)$  como el observador está fuera del cubo y el punto de atención en el centro del mismo, podemos usar los bordes para delimitar

la distancia horizontal hasta los bordes  $\Rightarrow l = cx - \frac{s}{2}, r = cx + \frac{s}{2}$ .

Dado que  $u = (0, 1, 0)$ , podemos repetir el mismo razonamiento que acabamos de aplicar y obtenemos que  $\Rightarrow b = cy - \frac{s}{2}, t = cy + \frac{s}{2}$ . Para maximizar el tamaño de los objetos, sin que se recorte ningún triángulo, necesitaremos que todos los vértices queden dentro del cubo, lo más cercano posible a las paredes, sin salirse del mismo. Nuevamente, usando el razonamiento de antes  $\Rightarrow -n = cz + \frac{s}{2}, -f = cz - \frac{s}{2}$

### Problema 3.6.

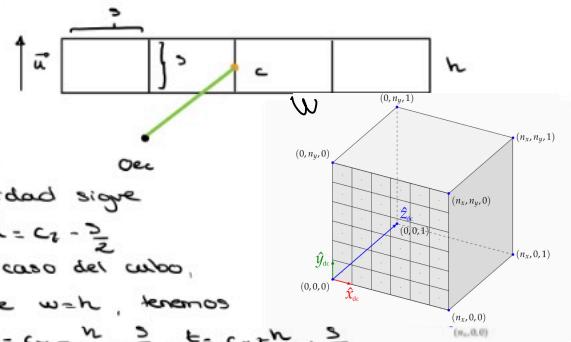
Repite el problema anterior 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora la escena, en lugar de estar contenida en un cubo de lado  $s$  unidades, está contenida en una esfera de radio  $r$  unidades (con centro igualmente en  $c$ ).



Para la distancia horizontal, proyectamos la esfera en el plano  $YZ$  y obtenemos que (usando el mismo razonamiento que el ejercicio anterior) que  $l = cx - r, r = cx + r$ . Análogamente al ejercicio anterior y el razonamiento que acabamos de hacer, obtenemos que  $b = cy - r, t = cy + r, -n = cz + r, -f = cz - r$

### Problema 3.7.

Repite el problema 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora, en lugar de suponer que el viewport es cuadrado, sabemos que tiene  $w$  columnas de pixels y  $h$  filas de pixels, y no podemos suponer que  $w = h$ .



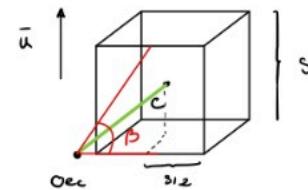
Como podemos observar, la profundidad sigue siendo la misma  $\Rightarrow -n = c_z + \frac{s}{2}$ ,  $-l = c_z - \frac{s}{2}$ . El resto del problema es análogo al caso del cubo, salvo que como no podemos suponer que  $w=h$ , tenemos que:  $l = cx - \frac{w}{h} \cdot \frac{s}{2}$ ,  $r = cy + \frac{w}{h} \cdot \frac{s}{2}$ ,  $b = cz - \frac{n}{h} \cdot \frac{s}{2}$ ,  $t = cz + \frac{n}{h} \cdot \frac{s}{2}$ . Con esto, conseguimos mantener las proporciones. Para hallar esos valores, basta ver que  $\frac{r-l}{t-b} = \frac{w}{h} \Leftrightarrow r-l = \frac{w}{h} (t-b) = \frac{w}{h} s$

### Problema 3.8.

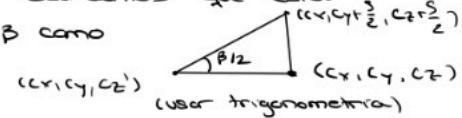
Repite el problema 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora se nos da un ángulo  $\beta$  en grados que debe ser la apertura de campo vertical de la proyección perspectiva. Para ello, ahora tenemos libertad para situar al observador en la línea paralela al eje Z que pasa por  $c$ , de forma que la apertura de campo vertical sea exactamente  $\beta$ .

Indica como calcular la coordenada Z que debemos usar ahora para  $Oec$  (la X y la Y son las mismas que antes), de forma que se cumpla lo dicho, también indica como debemos de calcular ahora los valores de  $I, r, t, b, n$  y  $f$  (todo ello en función de  $\beta, s$  y  $c = (c_x, c_y, c_z)$ ).

$\beta$  está contenido entre  $0$  y  $180^\circ$ . Como tenemos que  $Oec$  está en la recta paralela al eje Z que pasa por  $c$  (que recordemos que está centrado en el cubo), podemos calcular  $\beta$  como



sabemos que la apertura vertical de



Leyendo en el anterior

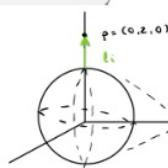
# Tema 7

## Problema 3.9.

Suponemos que visualizamos una esfera de radio unidad centrada en el origen. Se ilumina con una fuente de luz puntual situada en el punto  $p = (0, 2, 0)$ . El observador está situado en  $\mathbf{o} = (2, 0, 0)$ . En estas condiciones:

- Describe razonadamente en qué punto de la superficie de la esfera el brillo será máximo si el material es puramente difuso ( $k_d = 1$  en todos los puntos,  $y k_a$  y  $k_s$  a 0) ¿es ese punto visible para el observador?
- Repite el razonamiento anterior asumiendo ahora que el material es puramente pseudo-especular ( $M_S = (1, 1, 1)$ , resto a cero). Indica si dicho punto es visible para el observador.

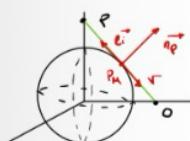
En el caso de los materiales puramente difusos, el brillo es máximo cuando la luz incide directamente sobre la superficie, ya que no reflejan la luz.



La normal en una esfera centrada es el vector que va del origen al punto normalizado (longitud = 1)  $\Rightarrow \vec{n}_p = \frac{1}{\sqrt{x^2+y^2+z^2}}(x, y, z)$ . Como nuestra esfera tiene  $r = 1$ ,  $\vec{n}_p = (x, y, z)$ .

Para maximizarlo, queremos que la normal sea paralela al vector que va del punto a la fuente de luz, es decir, queremos que la posición del punto en la esfera sea tal que  $(x, y, z)$  sea paralelo al vector que apunta a  $(0, 2, 0)$  desde el origen. Buscamos las coordenadas máximas que lo cumplen (el más cercano a la fuente de luz), que en este caso es el  $(0, 1, 0)$ . Este punto no es visible desde la posición del espectador.

La componente especular se refiere a la parte de la reflexión de la luz que causa reflejos o destellos intensos en una superficie.



Que  $M_S = (1, 1, 1)$  y resto a 0 ( $k_d, k_s, k_a$ ), significa que la componente especular domina completamente en la reflexión de la luz. El brillo especular será máximo cuando la luz se refleje directamente hacia el espectador.

Para un material pseudo - especular, esto ocurrirá cuando la dirección de la reflexión esté alineada desde el punto de la esfera al observador.

Como queremos maximizarlo,  $\vec{r}_i = \vec{v}$  y  $\beta = 0^\circ$

$$\begin{aligned}\vec{r}_i &= (0, 2, 0) - \vec{p}_H = (0, 2, 0) - \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right) = \\ &= \left(-\frac{\sqrt{2}}{2}, 2 - \frac{\sqrt{2}}{2}, 0\right) \\ \vec{r}_i &= (2, 0, 0) - \vec{p}_H = \left(2 - \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right) \\ \vec{n}_p &= \vec{P} = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right) \\ \vec{r}_i &= 2(\vec{r}_i \cdot \vec{n}_p)\vec{n}_p - \vec{r}_i = \vec{r}_i = \left(2 - \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right)\end{aligned}$$

El punto sí es visible para el espectador.

Si  $\alpha > 90^\circ$ , entonces:

- $\cos(\alpha)$  es negativo.
- la superficie, en  $\mathbf{p}$ , está orientada de espaldas a la fuente de luz.
- la contribución de esa fuente debe ser 0.

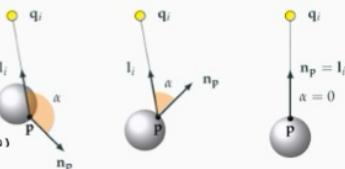
Si  $0^\circ \leq \alpha \leq 90^\circ$ , entonces:

- la superficie, en  $\mathbf{p}$ , está orientada de cara a la fuente de luz.
- $\cos(\alpha)$  estará entre 0 y 1 (entre  $\cos(90^\circ)$  y  $\cos(0^\circ)$ ).
- se puede demostrar que el valor  $\cos(\alpha)$  es proporcional a la densidad de fotones por unidad de área que inciden en el entorno de  $\mathbf{p}$ , provenientes de la  $i$ -ésima fuente de luz.

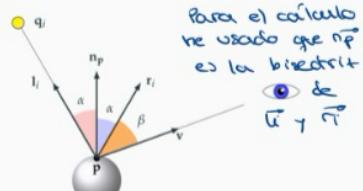
$0^\circ < \alpha$   
 $0 > \cos(\alpha)$

$0^\circ < \alpha < 90^\circ$   
 $1 > \cos(\alpha) > 0$

$\alpha = 0^\circ$   
 $\cos(\alpha) = 1$



El valor  $\mathbf{r}_i \cdot \mathbf{v}$  es el coseno del ángulo  $\beta$  que hay entre la dirección de máxima brillo  $\mathbf{r}_i$  y la dirección  $\mathbf{v}$  hacia el observador. Cuando  $\mathbf{r}_i = \mathbf{v}$  entonces  $\beta = 0^\circ$ ,  $\cos(\beta) = 1$ , y el brillo es máximo:



3. Haz lo mismo para un material pseudo-especular, pero con el modelo de Blinn-Phong (ecuación (7) de la transparencia 43).

### 3. Material Pseudo-Especular con modelo de Blinn-Phong

El modelo de Blinn-Phong utiliza el vector  $\text{halfway}(h)$  (bisectriz) en lugar del vector reflejado. El brillo es máximo cuando la normal de la superficie ( $n_p$ ) coincide con este vector  $h$ .

La condición de máximo brillo se da cuando el ángulo entre la normal  $n_p$  y el vector bisectriz  $h$  es 0, es decir:  $n_p = h$ .

#### 1. Cálculo de vectores de dirección:

Asumimos vectores direccionales normalizados desde el origen hacia la luz y el observador (igual que en el razonamiento de la imagen anterior):

- Vector hacia la luz ( $l$ ): Apunta hacia  $(0, 2, 0) \rightarrow$  normalizado  $\hat{l} = (0, 1, 0)$ .
- Vector hacia el observador ( $v$ ): Apunta hacia  $(2, 0, 0) \rightarrow$  normalizado  $\hat{v} = (1, 0, 0)$ .

#### 2. Cálculo del vector Halfway ( $h$ ):

El vector  $h$  es la suma de  $l$  y  $v$  normalizada.  $\varnothing$

$$\vec{h} = \hat{l} + \hat{v} = (0, 1, 0) + (1, 0, 0) = (1, 1, 0)$$

Para normalizarlo calculamos su módulo:

$$\|\vec{h}\| = \sqrt{1^2 + 1^2 + 0^2} = \sqrt{2}$$

El vector unitario es:

$$\hat{h} = \left( \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0 \right) = \left( \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0 \right)$$

### Problemas: evaluación de la componente pseudo-especular de modelo de Phong

#### Problema 7.1:

Escribe el código GDScript de una función para calcular la reflectividad debida a la componente pseudo-especular del modelo de iluminación local de Phong, es decir, el valor real resultante de evaluar la expresión de  $f_{ph}$ , que aparece en la ecuación (6) de la transparencia 40.

La función recibirá como parámetros los vectores unitarios  $n_p$ ,  $v$  y  $l_i$ , (todos de tipo `Vec3`), el exponente  $e$  y el valor escalar  $k_{ph}$  (ambos de tipo `float`). La función devolverá un valor de tipo `float`.

Escribe otra versión de la función, con los mismos parámetros, pero ahora para la componente pseudo-especular del modelo de Blinn-Phong, es decir, para evaluar la expresión de  $f_{bp}$ , que aparece en la ecuación (7) de la transparencia 43.

### 1. Componente Pseudo-Especular de Phong

Esta función implementa la ecuación (6) de la página 37.  $\varnothing$

Para calcularla necesitamos:

1. El factor  $d_i$ , que descarta la luz si la superficie está "de espaldas" a la fuente ( $n_p \cdot l_i \leq 0$ ).  $\varnothing$
2. El vector reflejado  $r_i$ , que se calcula como  $2(l_i \cdot n_p)n_p - l_i$  según la página 38.  $\varnothing$

<-- end list -->

#### gdscript

```
func calcular_phong_especular(np: Vector3, v: Vector3, l: Vector3, e: float, k_ph: float) -> float:
    # Calculamos el producto escalar (n . l)
    var n_dot_l = np.dot(l)

    # Implementación del factor 'di' (página 38):
    # Si la superficie no mira a la luz, no hay componente especular.
    if n_dot_l <= 0.0:
        return 0.0

    # Calculamos el vector reflejado 'r' (página 38)
    # Fórmula: r = 2 * (l . n) * n - l
    var r = 2.0 * n_dot_l * np - l

    # Calculamos el factor de brillo: (max(0, r . v))^e
    var r_dot_v = max(0.0, r.dot(v))
    var factor_especular = pow(r_dot_v, e)

    # Resultado final según ecuación (6): k_ph * di * (r.v)^e
    return k_ph * factor_especular
```

### 3. Determinación del Punto P:

Como queremos maximizar el brillo, la normal de la superficie debe ser igual a  $\hat{h}$ . Dado que estamos en una esfera de radio unidad centrada en el origen, la normal en un punto  $F$  es el propio punto  $F$ .

$$n_p = \hat{h} \Rightarrow P = \left( \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0 \right)$$

### 4. Comprobación de visibilidad:

Para saber si es visible, comprobamos si el ángulo entre la normal en  $F$  y el vector que va de  $F$  al observador ( $O$ ) es menor de  $90^\circ$  (producto escalar  $> 0$ ).

- Observador  $O = (2, 0, 0)$
- Vector visión desde superficie  $\vec{obs} = O - F$

$$\vec{obs} = (2, 0, 0) - \left( \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0 \right) = \left( 2 - \frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}, 0 \right)$$

Producto escalar  $n_p \cdot \vec{obs}$ :

$$\left( \frac{\sqrt{2}}{2} \right) \cdot \left( 2 - \frac{\sqrt{2}}{2} \right) + \left( \frac{\sqrt{2}}{2} \right) \cdot \left( -\frac{\sqrt{2}}{2} \right)$$

$$= \left( \sqrt{2} - \frac{2}{4} \right) - \frac{2}{4}$$

$$= \sqrt{2} - 0.5 - 0.5 = \sqrt{2} - 1 \approx 0.414$$

Como  $0.414 > 0$ , el ángulo es menor de  $90^\circ$ .

Conclusión:

El punto de máximo brillo es  $P = \left( \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0 \right)$  y si es visible para el espectador.

$$f_{bp}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = k_{bp}(\mathbf{p}) d_i [ \mathbf{n}_p \cdot \mathbf{h}_i ]^e$$

### 2. Componente Pseudo-Especular de Blinn-Phong

Esta función implementa la ecuación (7) de la página 40.  $\varnothing$

La diferencia principal es el uso del vector  $\text{halfway}(h)$ , definido en la página 41 como la bisectriz

$$\text{entre la luz y la vista: } h_i = \frac{l_i + v}{\|l_i + v\|} \varnothing$$

#### gdscript

```
func calcular_blinn_phong_especular(np: Vector3, v: Vector3, l: Vector3, e: float, k_bp: float) -> float:
    # Normalizamos la luz que incide desde el frente (mismo factor 'di')
    if np.dot(l) <= 0.0:
        return 0.0

    # Calculamos el vector halfway 'h' (página 41)
    # Notación: h = (l + v) normalizado
    var h = (l + v).normalized()

    # Calculamos el factor de brillo: (n . h)^e
    # Implementación del factor 'di' (página 38):
    var n_dot_h = max(0.0, np.dot(h))
    var factor_especular = pow(n_dot_h, e)

    # Resultado final según ecuación (7): k_bp * di * (n.h)^e
    return k_bp * factor_especular
```

#### Notas sobre la implementación:

- Vectores Unitarios: El enunciado asume que  $np$ ,  $v$  y  $l$  vienen normalizados (son unitarios), por lo que no hace falta llamar a `normalized()` sobre ellos, pero si es necesario normalizar el vector  $h$  en Blinn-Phong porque la suma de dos vectores unitarios no es necesariamente unitaria.
- Parámetro  $d_i$ : En ambas funciones, el `if` inicial maneja el parámetro  $d_i$  descrito en la página 38, devolviendo 0 inmediatamente si la luz está detrás de la superficie.  $\varnothing$

## Problema: evaluación de la BRDF GGX

$$f_{\text{ggx}}(\dot{x}, \mathbf{w}_i, \mathbf{w}_o) = \frac{F_R(\mathbf{w}_i, \mathbf{w}_h) D(\mathbf{w}_h) G_2(\mathbf{w}_i, \mathbf{w}_o, \mathbf{w}_h)}{4(\mathbf{w}_i \cdot \mathbf{n}_{\dot{x}})(\mathbf{w}_o \cdot \mathbf{n}_{\dot{x}})}$$

### Problema 7.3:

Escribe el código GDScript de una función para calcular la reflectividad debida a la BRDF de microfacetas GGX, es decir, el color resultado de evaluar la expresión de  $f_{\text{ggx}}$  que aparece en la ecuación (10) de la transparencia 79.

La función recibirá como parámetros los vectores unitarios  $\mathbf{w}_i$ ,  $\mathbf{w}_o$ ,  $\mathbf{t}_x$ ,  $\mathbf{t}_y$  y  $\mathbf{n}_{\dot{x}}$ , (todos de tipo `Vec3`), y los valores  $\alpha_x$  y  $\alpha_y$  (ambos de tipo `float`). La función devolverá un valor de tipo `float`.

Esta función implementa la BRDF GGX completa (Ec. 10, pág. 77 del PDF), calculando sus tres componentes principales:

1. D (Distribución de normales): Usando la fórmula anisotrópica (pág. 73).
2. G (Enmascaramiento/Sombras): Usando la aproximación de Smith con la función  $\Lambda$  (pág. 75).
3. F (Fresnel): Usando la aproximación de Schlick (pág. 76).

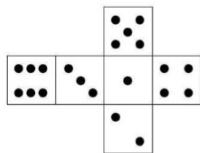
Nota importante: La fórmula de Fresnel requiere un parámetro  $f_0$  (índice de refracción en incidencia normal). Como el enunciado no incluye  $f_0$  en la lista de parámetros de la función, he definido una variable local  $f0 = 0.04$  (valor típico para materiales dieléctricos como plástico o agua) para que el cálculo sea funcional.

```
func calcular_brdf_ggx(wi: Vector3, wo: Vector3, tx: Vector3, ty: Vector3, nx: Vector3, alpha_x: float, alpha_y: float) -> float:  
    # 1. Producto escalar de N con Wi y Wo  
    # Necesarios para el denominador y para comprobar si estamos debajo de la superficie  
    var n_dot_wi = nx.dot(wi)  
    var n_dot_wo = nx.dot(wo)  
  
    # Si la luz viene de abajo o la vista es desde abajo, la reflectividad es 0  
    if n_dot_wi <= 0.0 or n_dot_wo <= 0.0:  
        return 0.0  
  
    # 2. Calcular el vector Halfway (wh)  
    # wh es la bisectriz entre wi y wo normalizada (Pág. 72)  
    var wh = (wi + wo).normalized()  
  
    # -----  
    # CÁLCULO DE D (Distribución de normales) - Pág. 73 (Anisotrópica)  
    # -----  
    # Proyectamos wh sobre el marco de referencia (tx, ty, nx)  
    var wh_x = wh.dot(tx)  
    var wh_y = wh.dot(ty)  
    var wh_z = wh.dot(nx)  
  
    # Evitamos división por cero si alpha es muy pequeño  
    var ax = max(0.001, alpha_x)  
    var ay = max(0.001, alpha_y)  
  
    var term_d = (wh_x * wh_x) / (ax * ax) + (wh_y * wh_y) / (ay * ay) + (wh_z * wh_z)  
    var D = 1.0 / (PI * ax * ay * term_d * term_d)  
  
    # -----  
    # CÁLCULO DE G (Masking-Shadowing G2) - Pág. 75  
    # -----  
    # Función auxiliar Lambda para Wi  
    # Proyectamos wi sobre el marco local para la fórmula de Lambda  
    var wi_x = wi.dot(tx)  
    var wi_y = wi.dot(ty)  
    # wi_z es n_dot_wi  
    var term_wi = (ax * ax * wi_x * wi_x + ay * ay * wi_y * wi_y) / (n_dot_wi * n_dot_wi)  
    var lambda_wi = 0.5 * (-1.0 + sqrt(1.0 + term_wi))  
  
    # Función auxiliar Lambda para Wo  
    var wo_x = wo.dot(tx)  
    var wo_y = wo.dot(ty)  
    # wo_z es n_dot_wo  
    var term_wo = (ax * ax * wo_x * wo_x + ay * ay * wo_y * wo_y) / (n_dot_wo * n_dot_wo)  
    var lambda_wo = 0.5 * (-1.0 + sqrt(1.0 + term_wo))  
  
    # G2 = 1 / (1 + lambda_wi + lambda_wo)  
    # (Asumimos chi+ = 1 porque ya comprobamos n_dot_wi > 0 y n_dot_wo > 0 al inicio)  
    var G2 = 1.0 / (1.0 + lambda_wi + lambda_wo)  
  
    # -----  
    # CÁLCULO DE F (Fresnel Schlick) - Pág. 76  
    # -----  
    # Variable local f0 (No provista en parámetros, asumimos dieléctrico común)  
    var f0 = 0.04  
    var wi_dot_wh = max(0.0, wi.dot(wh))  
    var F = f0 + (1.0 - f0) * pow(1.0 - wi_dot_wh, 5)  
  
    # -----  
    # EXPRESIÓN FINAL - Ec. (10) Pág. 77  
    # -----  
    # f_ggx = (F * D * G) / (4 * (n.wi) * (n.wo))  
    var numerador = F * D * G2  
    var denominador = 4.0 * n_dot_wi * n_dot_wo  
  
    return numerador / max(0.0001, denominador)
```

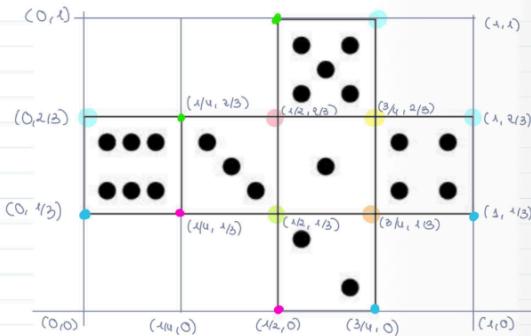
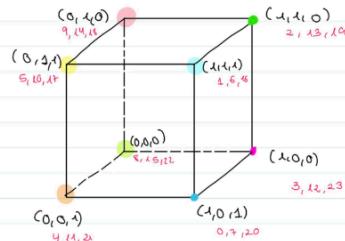
# Tarea 8

## Problema 8.1:

Supongamos que se desea crear una malla indexada para un cubo, de forma que deseamos aplicar un texture que incluya las caras de un dado. Para ello disponemos de una imagen de texture que tiene una relación de aspecto 4:3. La imagen aparece aquí:



a) Necesitaremos como mínimo 24 vértices (3 por cada vértice del cubo)



```

vertices = []
//cara derecha
{1,0,1}, // 0
{1,1,1}, // 1
{1,1,0}, // 2
{1,0,0}, // 3

//cara frente
{0,0,1}, // 4
{0,1,1}, // 5
{1,1,1}, // 6
{1,0,1}, // 7

//cara izquierda
{0,0,0}, // 8
{0,1,0}, // 9
{0,1,1}, // 10
{0,0,1}, // 11

//atrás
{1,0,0}, // 12
{1,1,0}, // 13
{0,1,0}, // 14
{0,0,0}, // 15

//techo
{1,1,1}, // 16
{0,1,1}, // 17
{0,1,0}, // 18
{1,1,0}, // 19

//suelo
{1,0,1}, // 20
{0,0,1}, // 21
{0,0,0}, // 22
{1,0,1} // 23

```

## Problema 8.1 (continuación):

Responde a estas cuestiones:

- Describe razonadamente cuantos vértices (como mínimo) tendrá el modelo.
- Escribe la tabla de coordenadas de vértices, la tabla de coordenadas de textura, y la tabla de triángulos. Ten en cuenta que el cubo tiene lado unidad y su centro está en  $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ . Dibuja un esquema de la textura en la cual cada vértice del modelo aparezca etiquetado con su número de vértice más sus coordenadas de textura.

```

cc - t = }

//cara derecha
{0, 1/3}, {0, 2/3}, {1/4, 2/3}, {1/4, 1/3}

//cara frente
{3/4, 1/3}, {3/4, 2/3}, {1, 2/3}, {1, 1/3}

//cara izquierda
{3/4, 1/3}, {3/4, 2/3}, {3/4, 2/3}, {3/4, 1/3}

//atrás
{1/4, 1/3}, {1/4, 2/3}, {1/4, 2/3}, {1/4, 1/3}

//techo
{3/4, 1/3}, {3/4, 2/3}, {1/2, 2/3}, {1/2, 1/3}

//suelo
{1/2, 0}, {3/4, 1/3}, {1/2, 1/3}, {3/4, 0}

```

### Problema 3.11.

Considera de nuevo el cubo y la textura del problema anterior. Ahora supón que queremos visualizar con OpenGL el cubo usando sombreado de Gouraud o de Phong, para lo cual debemos de asignar normales a los vértices. Responde a estas cuestiones

- Describe razonadamente si sería posible usar la misma tabla de vértices y la misma tabla de coordenadas de textura que has escrito en el problema anterior, o es necesario usar otra tabla distinta.
- Si has respondido que no es posible usar las mismas tablas, escribe la nueva tabla de vértices, la nueva tabla de coordenadas de textura. Asimismo, escribe como sería la tabla de normales.

Necesitamos escribir otra distinta para las normales ya que estas son vec3, no vec2.

se usa el siguiente algoritmo:

$$\text{Para cada vértice } \vec{n} = \frac{\vec{s}}{\|\vec{s}\|} \quad \vec{s} = \sum_{k=0}^{k=3} \vec{m}_k$$

donde  $\vec{m}_k$  las normales de las caras adyacentes al vértice

$$\text{normales de caras: } \vec{n} = \frac{\vec{a} \times \vec{b}}{\|\vec{a} \times \vec{b}\|}$$

### Problema 3.12.

Considera un cubo (de nuevo de lado unidad, y con centro en  $(1/2, 1/2, 1/2)$ ) que se quiere visualizar con una textura a partir de una única imagen (cuadrada) que se replicará en las 6 caras de dicho cubo. Asume que no se va a usar iluminación (no es necesario calcular la tabla de normales). Escribe ahora la tabla de coordenadas de vértices y la tabla de coordenadas de textura.

nor\_ver =  $\{ \{-1, 0, 0\}, \{-1, 0, 0\}, \{-1, 0, 0\}, \{-1, 0, 0\}, \{1, 0, 0\}, \{1, 0, 0\}, \{1, 0, 0\}, \{1, 0, 0\}, \{0, -1, 0\}, \{0, -1, 0\}, \{0, -1, 0\}, \{0, -1, 0\}, \{0, 1, 0\}, \{0, 1, 0\}, \{0, 1, 0\}, \{0, 1, 0\}, \{0, 0, -1\}, \{0, 0, -1\}, \{0, 0, -1\}, \{0, 0, -1\}, \{0, 0, 1\}, \{0, 0, 1\}, \{0, 0, 1\}, \{0, 0, 1\} \}$

Mismo que cubo 24 con la diferencia de que el cubo 24 está centrado en  $(0, 0, 0)$

3.44

Vértices y triángulos igual  
Cambiar coordenadas de textura  
Es un cuadrado para cada cara  
 $(0,0), (1,0), (1,1), (0,1)$   
para todas las caras!!!

### Problema 4.1.

### Antiguos ↓ TGcreo

Supongamos que queremos visualizar una secuencia de frames, en los cuales la cámara va cambiando. Para ello queremos escribir el código de una función que fija la matriz de vista en el cauce. La función acepta como parámetro un valor real  $t$ , que es el tiempo en segundos transcurrido desde el inicio de la animación. Suponemos que la animación dura  $s$  segundos en total.

En ese tiempo el observador de cámara se desplaza con un movimiento uniforme desde un punto de coordenadas de mundo  $\mathbf{o}_0$  (para  $t = 0$ ) hasta un punto destino  $\mathbf{o}_1$  (para  $t = 1$ ). Además el punto de atención de la cámara también se desplaza desde  $\mathbf{a}_0$  hasta  $\mathbf{a}_1$ . Durante toda la animación, el vector VUP es  $(0, 1, 0)$ .

Escribe el pseudo-código de la citada función.

```
void desplazar (float t, float s) {
     $\mathbf{o}_{ec} = (\mathbf{1} - \frac{t}{s}) \mathbf{o}_0 + \frac{t}{s} \mathbf{o}_1$ 
     $\mathbf{a}_{ec} = (\mathbf{1} - \frac{t}{s}) \mathbf{a}_0 + \frac{t}{s} \mathbf{a}_1$ 
     $\vec{n} = \mathbf{o}_{ec} - \mathbf{a}_{ec}$ 
     $\vec{u} = (0, 1, 0)$ 
     $\vec{z}_{ec} = \frac{\vec{n}}{\|\vec{n}\|}$   $\vec{x}_{ec} = \frac{\vec{u} \times \vec{z}_{ec}}{\|\vec{u} \times \vec{z}_{ec}\|}$ 
     $\vec{y}_{ec} = \vec{z}_{ec} \times \vec{x}_{ec}$ 
}
```

En el modo de primera persona con traslaciones, la actualización de la cámara supone simplemente trasladar el origen del marco de cámara  $\mathbf{o}_{ec}$  y el punto de atención  $\mathbf{a}_t$  de forma solidaria:

► La operación **desplRotarXY**( $\Delta_a, \Delta_b$ ) supone:

- $\mathbf{a}_t = \mathbf{a}_t + \Delta_x \mathbf{x}_{ec} + \Delta_y \mathbf{y}_{ec}$
- $\mathbf{o}_{ec} = \mathbf{a}_t + \mathbf{n}$

► La operación **moverZ**( $\Delta_z$ ) supone:

- $\mathbf{a}_t = \mathbf{a}_t + \Delta_z \mathbf{z}_{ec}$
- $\mathbf{o}_{ec} = \mathbf{a}_t + \mathbf{n}$

Las tuplas  $\mathbf{s}, \mathbf{n}, \mathbf{x}_{ec}, \mathbf{y}_{ec}, \mathbf{z}_{ec}$  no cambian.

El marco de referencia de vista  $\mathcal{V}$ , se define a partir de los siguientes parámetros

$\mathbf{o}_{ec}$  = es el punto del espacio foco de la proyección, donde estaría situado el observador ficticio que contempla la escena (projection reference point, PRP)

$\vec{n}$  = vector libre perpendicular al plano de visión (plano ficticio donde se proyecta la imagen perpendicular al eje óptico de la cámara virtual). (view plane normal, VPN).

$\mathbf{a}$  = punto en el eje óptico, también llamado punto de atención o look-at point.

$\vec{u}$  = es un vector libre que indica una dirección que el observador ve proyectada en vertical en la imagen (apuntando hacia arriba) (view-up vector, VUP)

De los tres parámetros  $\mathbf{o}_{ec}$ ,  $\vec{n}$  y  $\mathbf{a}$  solo hay que especificar dos, ya que no son independientes (se cumple  $\mathbf{o}_{ec} = \mathbf{a} + \vec{n}$ ).

$$\hat{z}_{ec} = \frac{\vec{n}}{\|\vec{n}\|} \quad (\text{eje Z paralelo a VPN, normalizado})$$

$$\hat{x}_{ec} = \frac{\vec{u} \times \vec{n}}{\|\vec{u} \times \vec{n}\|} \quad (\text{eje X perpendicular a VPN y VUP, normalizado})$$

$$\hat{y}_{ec} = \hat{z}_{ec} \times \hat{x}_{ec} \quad (\text{eje Y perpendicular a los otros dos})$$

```
const GLfloat V[4][4] = // matriz V asociada al marco  $\mathcal{V}$  (por filas)
{{ a_x, a_y, a_z, d_x }, // coords. de mundo de  $\mathbf{x}_{ec}$ , y  $d_x = -\mathbf{o}_{ec} \cdot \mathbf{x}_{ec}$ 
 { b_x, b_y, b_z, d_y }, // coords. de mundo de  $\mathbf{y}_{ec}$ , y  $d_y = -\mathbf{o}_{ec} \cdot \mathbf{y}_{ec}$ 
 { c_x, c_y, c_z, d_z }, // coords. de mundo de  $\mathbf{z}_{ec}$ , y  $d_z = -\mathbf{o}_{ec} \cdot \mathbf{z}_{ec}$ 
 { 0, 0, 0, 1 } // origen de  $\mathcal{V}$ 
};

glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glMultTransposeMatrixf( V );
```

### Problema 4.2.

Una posibilidad para hacer selección en mallas de triángulos es usar cálculo de intersecciones entre un rayo (una semirrecta que pasa por el centro de un pixel) y cada uno de los triángulos de la malla. Diseña un algoritmo en pseudo-código para el cálculo de intersecciones entre un rayo y un triángulo:

- ▶ El rayo tiene como origen o extremo el punto cuyas coordenadas del mundo es la tupla  $\mathbf{o}$ , y como vector de dirección la tupla  $\mathbf{d}$  (la suponemos normalizada).
- ▶ Las coordenadas del mundo de los vértices del triángulo son  $\mathbf{v}_0, \mathbf{v}_1$  y  $\mathbf{v}_2$ .
- ▶ El algoritmo debe de indicar si hay intersección o no, y, en caso de que la haya, calcular las coordenadas del mundo del punto de intersección.

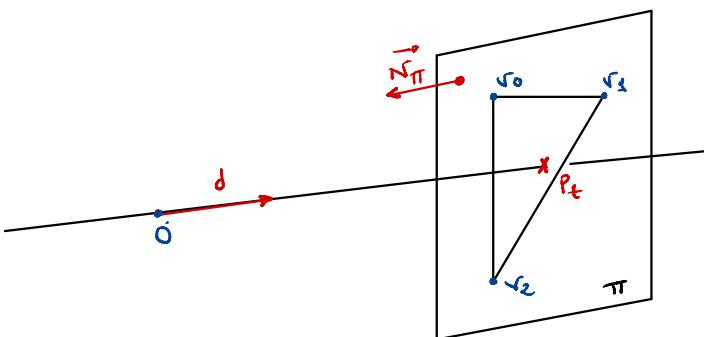
(ver la siguiente transparencia).

### Problema 4.2. (continuación)

Ten en cuenta que habrá intersección si y solo si se cumplen cada una de estas dos condiciones:

1. El rayo interseca con el plano que contiene al triángulo, es decir, existe  $t > 0$  tal que el punto  $\mathbf{p}_t \equiv \mathbf{o} + t\mathbf{d}$  está en dicho plano. Equivale a decir que el vector  $\mathbf{p}_t - \mathbf{v}_0$  es perpendicular a la normal al plano.
2. El punto  $\mathbf{p}_t$  citado arriba está dentro del triángulo. Es decir, hay dos valores reales no negativos  $a$  y  $b$  (con  $0 \leq a + b \leq 1$ ) tales que el vector  $\mathbf{p}_t - \mathbf{v}_0$  es igual a  $a(\mathbf{v}_1 - \mathbf{v}_0) + b(\mathbf{v}_2 - \mathbf{v}_0)$ .

(a los tres valores  $a, b$  y  $c \equiv 1 - a - b$  se les llama *coordenadas baricéntricas* de  $\mathbf{p}_t$  en el triángulo, se usan en ray-tracing).



La recta y el plano no intersecan y solo si  $\vec{d} \perp \vec{N}_{\pi}$  (perpendiculares) (evidentemente,  $\vec{d}$  no está contenido en el plano), luego, si intersecan, tenemos que  $\vec{d} \neq \vec{N}_{\pi} \Rightarrow \vec{d} \cdot \vec{N}_{\pi} \neq 0$

Podemos calcular el punto de intersección como  $\mathbf{p}_t = \mathbf{o} + t\mathbf{d}$ . Si  $t < 0$ , el rayo no interseca con el plano del triángulo. En caso contrario, calcularemos las coordenadas baricentricas de  $\mathbf{p}_t - \mathbf{v}_0 = a(\mathbf{v}_1 - \mathbf{v}_0) + b(\mathbf{v}_2 - \mathbf{v}_0)$  y si  $0 \leq a+b \leq 1$ ,  $a, b \geq 0$ , devolvemos true.

Para calcular el punto intersección, el plano es el conjunto de puntos  $\mathbf{x}$  que satisfacen  $(\mathbf{x} - \mathbf{v}_0) \cdot \vec{N}_{\pi} = 0$ . cogiendo  $\mathbf{p}_t = \mathbf{o} + t\mathbf{d}$  y sustituyendo:

$$((\mathbf{o} + t\mathbf{d}) - \mathbf{v}_0) \cdot \vec{N}_{\pi} = 0 \Rightarrow$$

$$\Rightarrow (\mathbf{o}^0 + t\mathbf{d}^0 - \mathbf{v}_0^0) \mathbf{N}_{\pi}^0 + (\mathbf{o}^1 + t\mathbf{d}^1 - \mathbf{v}_0^1) \mathbf{N}_{\pi}^1 + (\mathbf{o}^2 + t\mathbf{d}^2 - \mathbf{v}_0^2) \mathbf{N}_{\pi}^2 = 0$$

$$\Rightarrow t = \frac{-\mathbf{N}_{\pi}^0 \mathbf{o}^0 + \mathbf{N}_{\pi}^0 \mathbf{v}_0^0 - \mathbf{N}_{\pi}^1 \mathbf{o}^1 + \mathbf{N}_{\pi}^1 \mathbf{v}_0^1 - \mathbf{N}_{\pi}^2 \mathbf{o}^2 + \mathbf{N}_{\pi}^2 \mathbf{v}_0^2}{\mathbf{N}_{\pi}^0 \mathbf{d}^0 + \mathbf{N}_{\pi}^1 \mathbf{d}^1 + \mathbf{N}_{\pi}^2 \mathbf{d}^2}$$

Para calcular las coordenadas baricentricas:

$$(\mathbf{p}_t - \mathbf{v}_0) = a(\mathbf{v}_1 - \mathbf{v}_0) + b(\mathbf{v}_2 - \mathbf{v}_0), a, b \in \mathbb{R} \Rightarrow$$

$$\mathbf{p}_t^0 - \mathbf{v}_0^0 = a(\mathbf{v}_1^0 - \mathbf{v}_0^0) + b(\mathbf{v}_2^0 - \mathbf{v}_0^0) \quad \rightarrow a = \frac{\mathbf{p}_t^0 - \mathbf{v}_0^0 - b(\mathbf{v}_2^0 - \mathbf{v}_0^0)}{\mathbf{v}_1^0 - \mathbf{v}_0^0}$$

$$\Rightarrow \mathbf{p}_t^1 - \mathbf{v}_0^1 = a(\mathbf{v}_1^1 - \mathbf{v}_0^1) + b(\mathbf{v}_2^1 - \mathbf{v}_0^1)$$

$$\mathbf{p}_t^2 - \mathbf{v}_0^2 = a(\mathbf{v}_1^2 - \mathbf{v}_0^2) + b(\mathbf{v}_2^2 - \mathbf{v}_0^2) \quad \text{Redundante, ya que } \mathbf{p}_t - \mathbf{v}_0 \text{ en el plano y existe solución para } a, b$$

Sustituyendo:  $\mathbf{p}_t^1 - \mathbf{v}_0^1 = \frac{\mathbf{p}_t^0 - \mathbf{v}_0^0 - b(\mathbf{v}_2^0 - \mathbf{v}_0^0)}{\mathbf{v}_1^0 - \mathbf{v}_0^0} (\mathbf{v}_1^1 - \mathbf{v}_0^1) + b(\mathbf{v}_2^1 - \mathbf{v}_0^1)$

$$b = \frac{(\mathbf{p}_t^1 - \mathbf{v}_0^1)(\mathbf{v}_1^0 - \mathbf{v}_0^0) - (\mathbf{v}_1^1 - \mathbf{v}_0^1)(\mathbf{p}_t^0 - \mathbf{v}_0^0 - b(\mathbf{v}_2^0 - \mathbf{v}_0^0))}{(\mathbf{v}_1^0 - \mathbf{v}_0^0)(\mathbf{v}_2^1 - \mathbf{v}_0^1)}$$

$$a = \frac{p_t^0 - r_0^0 - \left[ \frac{(p_t^1 - r_0^1)(r_1^0 - r_0^0) - (r_1^1 - r_0^1)(p_t^0 - r_0^0) - b(r_2^0 - r_0^0)}{(r_1^0 - r_0^0)(r_2^1 - r_0^1)} \right] (r_2^0 - r_0^0)}{r_1^0 - r_0^0}$$

↑ cosas mías,  
pasad de ello

buscamos intersección  $(0, d, r_0, r_1, r_2)$

$$u = r_1 - r_0$$

$$v = r_2 - r_0$$

$$N_{\pi} = \| u \times v \|$$

if ( $d \cdot N_{\pi} = 0$ )

return false

$$t = \frac{-N_{\pi}^0 r_0^0 + N_{\pi}^0 r_0^0 - N_{\pi}^0 r_1^0 + N_{\pi}^1 r_0^0 - N_{\pi}^2 r_0^0 + N_{\pi}^2 r_2^0}{N_{\pi}^0 d^0 + N_{\pi}^1 d^1 + N_{\pi}^2 d^2}$$

if ( $t < 0$ )

return false

$$p_t = \vec{o} + t\vec{d}$$

$$b = \frac{(p_t^1 - r_0^1)(r_1^0 - r_0^0) - (r_1^1 - r_0^1)(p_t^0 - r_0^0) - b(r_2^0 - r_0^0)}{(r_1^0 - r_0^0)(r_2^1 - r_0^1)}$$

$$a = \frac{p_t^0 - r_0^0 - b(r_2^0 - r_0^0)}{r_1^0 - r_0^0}$$

if ( $a < 0 \text{ || } b < 0 \text{ || } a+b < 0 \text{ || } a+b \geq 2$ )

return false

return true

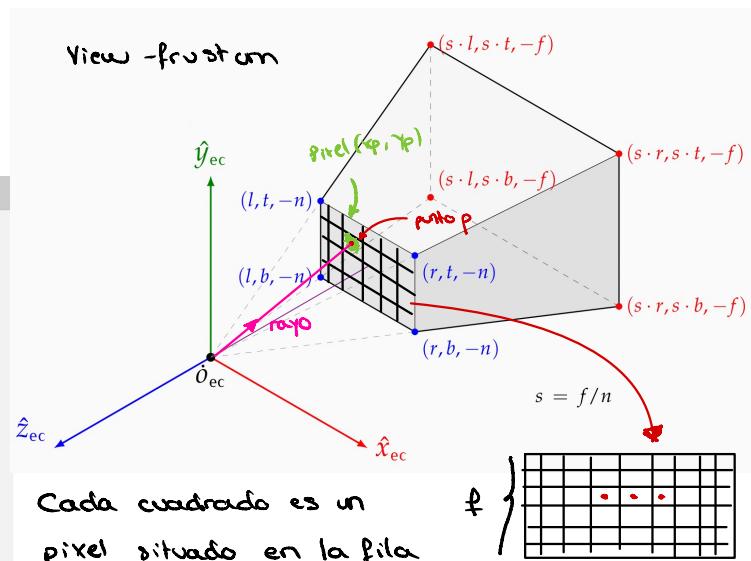
### Problema 4.3.

Para implementar la selección usando intersecciones es necesario calcular el rayo que tiene como origen el observador y pasa por centro del pixel donde se ha hecho click. Escribe el pseudo-código del algoritmo que calcula el rayo a partir de las coordenadas del pixel donde se ha hecho click:

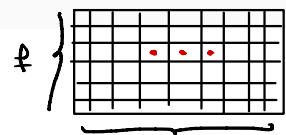
- ▶ Tenemos una vista perspectiva, y conocemos los 6 valores  $l, r, t, b, n, f$  usados para construir la matriz de proyección.
- ▶ También conocemos el marco de coordenadas de vista, es decir, las tuplas  $\mathbf{x}_{ec}, \mathbf{y}_{ec}$  y  $\mathbf{o}_{ec}$  con los versores y la tupla  $\mathbf{o}_{ec}$  con el punto origen (todos en coordenadas del mundo).
- ▶ El viewport tiene  $w$  columnas y  $f$  filas de pixels. Se ha hecho click en el pixel de coordenadas enteras  $x_p$  e  $y_p$

El algoritmo debe producir como salida las tuplas  $\mathbf{o}$  y  $\mathbf{d}$  (normalizado) que definen el rayo.

Tiene de origen de coordenadas  $\mathbf{o}_{ec} = (0, 0, 0)$  y  $\mathbf{d} = \mathbf{p} - \mathbf{o}_{ec}$ , siendo  $\mathbf{p}$  el centro del pixel (Hablando en coordenadas de vista). Como los pixeles mantienen siempre la misma proporción, podemos calcular cuánto ocupa cada pixel y, así, hallar fácilmente su punto central. Con eso bastaría para el ejercicio, pero, adicionalmente, vamos a pasar de coordenadas de vista a coordenadas de mundo.

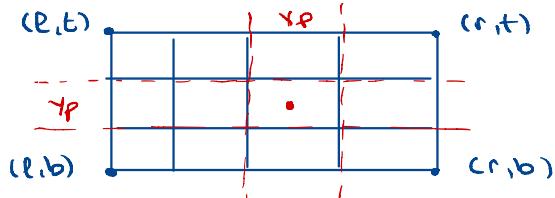


Cada cuadrado es un pixel situado en la fila  $i \in [0, f-1]$  y columna  $j \in [0, w-1]$  y tiene un punto central llamado centro del pixel. Si has hecho click en el pixel  $j=x_p, i=y_p$ . El rayo



calcular Rayo ( $x_p, y_p$ ) {

// Calculamos las coordenadas del punto central del pixel.



$$(t - b) / w = \Delta w$$

Incremento en cada eje

$$(t - b) / f = \Delta f$$

$$\Delta x = \frac{(x_{p+1} \Delta w) - (x_p \Delta w)}{2}$$

$$\Delta y = \frac{(y_{p+1} \Delta f) - (y_p \Delta f)}{2}$$

Incremento dentro de la celda hasta el punto

Para hallar el punto bastaría con

$$P_x = x_p \Delta w + \Delta x$$

$$P_y = y_p \Delta f + \Delta y$$

$P_z = -n$  (porque estamos en el plano near)

Ese ha sido el cálculo geométrico de  $P$ , también podemos hallarlo por interpolación:

$$P_x = l + (r - l) \cdot \frac{x_p + 0.5}{w} \quad P_y = b + (t - b) \cdot \frac{y_p + 0.5}{h}$$

vec3 p = vec3(  $P_x, P_y, P_z$  ) // Usar fórmula que quieras

vec3 d =  $p - \text{Oc}$  =  $P$

d = d.normalize()

return (Oc, d)

t

Como bien he dicho antes, estas coordenadas son de vista, que es suficiente para el enunciado. Si quisieramos las coordenadas de mundo, tendríamos que hacer lo siguiente:

calcularRayo ( $x_p, y_p$ ) {

vec3 p = vec3(  $P_x, P_y, P_z$  ) // Usar fórmula que quieras

vec3 d =  $p - \text{Oc}$  // =  $P$

Como solo nos interesa la dirección del rayo en coordenadas de mundo, basta con multiplicar cada componente de  $d$  por el marco [Vec, Tex, Zec] sin considerar  $w$ .

vec3 d\_wc = (  $d_x \cdot \text{Vec}, d_y \cdot \text{Tex}, d_z \cdot \text{Zec}$  )

return (Oc, d\_wc - normalize())

t

**Problema 9.1:**

En una aplicación Godot cualquiera, añade código al nodo raíz de forma que cada vez que se pulse y luego se levante una tecla (por ejemplo la tecla P), se imprima en pantalla un mensaje con el tiempo total en segundos que dicha tecla ha estado pulsada, en los casos en los que ha permanecido pulsada al menos el tiempo de un frame.

**Código GDScript (compatible con Godot 4.x)**

```
gdscript

extends Node

# Variable para almacenar el momento exacto en que se pulsó la tecla
var tiempo_inicio: float = 0.0

func _input(event):
    # Verificamos si el evento es de teclado y si corresponde a la tecla P
    if event is InputEventKey and event.keycode == KEY_P:

        # CASO 1: La tecla se acaba de pulsar (y no es un eco de mantenerla)
        if event.pressed and not event.echo:
            # Guardamos el tiempo actual del sistema en milisegundos
            tiempo_inicio = Time.get_ticks_msec()

        # CASO 2: La tecla se acaba de soltar (event.pressed es falso)
        elif not event.pressed:
            # Obtenemos el tiempo actual al soltar
            var tiempo_fin = Time.get_ticks_msec()

            # Calculamos la duración restando y convirtiendo a segundos
            var duracion_total = (tiempo_fin - tiempo_inicio) / 1000.0

            # Obtenemos la duración del último frame para la condición del enunciado
            var tiempo_frame = get_process_delta_time()

            # Condición: Solo imprimimos si ha estado pulsada al menos un frame
            if duracion_total >= tiempo_frame:
                print("La tecla P ha estado pulsada: ", duracion_total, " segundos.")


```

**Explicación del código:**

1. `_input(event)`: Utilizamos esta función integrada para interceptar los eventos de entrada antes de que sean procesados por otros nodos o la física. Es ideal para detectar el momento exacto de pulsar (`pressed = true`) y soltar (`pressed = false`).
2. `event.echo`: Es importante comprobar `not event.echo` al pulsar. Si no lo hicieramos, el sistema operativo enviaría múltiples eventos de "pulsado" si el usuario mantiene la tecla, reiniciando nuestro contador `tiempo_inicio` constantemente.
3. `Time.get_ticks_msec()`: Devuelve el tiempo en milisegundos desde que se inició el motor. Es más preciso para medir intervalos de tiempo real que acumular delta en `_process`.
4. `get_process_delta_time()`: Devuelve el tiempo en segundos que duró el último frame. Usamos esto para cumplir la condición del enunciado ("al menos el tiempo de un frame").

## Problema: intersección rayo-tríángulo (1/2)

### Problema 9.2:

Una posibilidad para hacer selección en mallas de triángulos es usar cálculo de intersecciones entre un rayo (una semirrecta que pasa por el centro de un pixel) y cada uno de los triángulos de la malla. Diseña un algoritmo en pseudo-código para el cálculo de intersecciones entre un rayo y un triángulo:

- El rayo tiene como origen o extremo el punto cuyas coordenadas del mundo es la tupla  $\mathbf{o}$ , y como vector de dirección la tupla  $\mathbf{d}$  (la suponemos normalizada).
- Las coordenadas del mundo de los vértices del triángulo son  $\mathbf{v}_0, \mathbf{v}_1$  y  $\mathbf{v}_2$ .
- El algoritmo debe de indicar si hay intersección o no, y, en caso de que la haya, calcular las coordenadas del mundo del punto de intersección.

(ver la siguiente transparencia).

### Problema 9.2 (continuación):

Ten en cuenta que habrá intersección si y solo si se cumplen cada una de estas dos condiciones:

1. El rayo interseca con el plano que contiene al triángulo, es decir, existe  $t > 0$  tal que el punto  $\mathbf{p}_t \equiv \mathbf{o} + t\mathbf{d}$  está en dicho plano. Equivale a decir que el vector  $\mathbf{p}_t - \mathbf{v}_0$  es perpendicular a la normal al plano  $\mathbf{n}$  (el producto escalar de ambos es nulo).

2. El punto  $\mathbf{p}_t$  citado arriba está dentro del triángulo. Es decir, hay dos valores reales no negativos  $a$  y  $b$  (con  $0 \leq a + b \leq 1$ ) tales que el vector  $\mathbf{p}_t - \mathbf{v}_0$  es igual a  $a(\mathbf{v}_1 - \mathbf{v}_0) + b(\mathbf{v}_2 - \mathbf{v}_0)$ .

(a los tres valores  $a$ ,  $b$  y  $c \equiv 1 - a - b$  se les llama *coordenadas baricéntricas* de  $\mathbf{p}_t$  en el triángulo, se usan en ray-tracing).

Esta implementación asume que el script está adjunto a un recurso Animation y utiliza los métodos de la API de Godot mencionados en el enunciado para acceder a los datos del track.

```
gdscript

extends Animation

# Función para obtener el valor interpolado de un track en un tiempo específico.
# Se basa en la interpolación lineal entre keyframes (Pág. 9 del PDF).
func track_get_key_value(track_idx: int, time: float) -> Variant:
    # 1. Obtener la cantidad total de keys en el track
    var key_count = track_get_key_count(track_idx)

    # --- Manejo de casos borde (0 o 1 key) ---
    if key_count == 0:
        # Si no hay keys, no hay valor que devolver.
        return null
    if key_count == 1:
        # Si solo hay una key, el valor es constante en todo el tiempo.
        # Usamos la sobrecarga existente: track_get_key_value(track_idx, key_idx)
        return track_get_key_value(track_idx, 0)

    # --- Manejo de tiempos fuera del rango de keys ---
    # Si el tiempo solicitado es anterior a la primera key
    var first_time = track_get_key_time(track_idx, 0)
    if time <= first_time:
        return track_get_key_value(track_idx, 0)

    # Si el tiempo solicitado es posterior a la última key
    var last_time = track_get_key_time(track_idx, key_count - 1)
    if time >= last_time:
        return track_get_key_value(track_idx, key_count - 1)

    # --- Búsqueda del intervalo e Interpolación ---
    # Iteraremos a través de las keys para encontrar el par que rodea a 'time'.
    for i in range(key_count - 1):
        var t_i = track_get_key_time(track_idx, i)
        var t_next = track_get_key_time(track_idx, i + 1)

        # Comprobamos si 'time' cae dentro del intervalo actual [t_i, t_next]
        if time >= t_i and time < t_next:
            # Hemos encontrado las keys (i) e (i+1). Obtenemos sus valores.
            var v_i = track_get_key_value(track_idx, i)
            var v_next = track_get_key_value(track_idx, i + 1)

            # 1. Calcular el factor de interpolación 'u' (entre 0.0 y 1.0)
            # Según la fórmula de la página 9 del PDF: u = (t - t_i) / (t_{i+1} - t_i)
            var u = (time - t_i) / (t_next - t_i)

            # 2. Realizar la interpolación lineal (lerp)
            # La función lerp(a, b, u) calcula: a + u * (b - a)
            # Godot maneja la interpolación de tipos simples (Vector3, float, etc.)
            return lerp(v_i, v_next, u)

    # Por seguridad, si el bucle termina sin retornar (no debería ocurrir con la
    # lógica anterior)
    return track_get_key_value(track_idx, key_count - 1)
```

#### Explicación del Código

1. **Casos Borde:** Primero se manejan los casos donde el track no tiene suficientes datos para interpolar (0 o 1 key) o cuando el tiempo solicitado está fuera del rango definido por la primera y última key. En estos casos, se devuelve el valor de la key más cercana (comportamiento estándar de "clamp").
2. **Búsqueda del Intervalo:** Se itera a través de los pares de keys consecutivas ( $i, i+1$ ). Se busca el intervalo donde el tiempo de la key actual  $t_i$  es menor o igual a  $time$ , y el tiempo de la siguiente key  $t_{next}$  es mayor que  $time$ .
3. **Cálculo del Factor  $u$ :** Una vez encontrado el intervalo, se calcula el factor de interpolación  $u$  que representa en qué punto entre  $t_i$  ( $u=0.0$ ) y  $t_{next}$  ( $u=1.0$ ) se encuentra el tiempo solicitado. Esto corresponde a la fórmula  $u = \frac{t - t_i}{t_{i+1} - t_i}$  de la página 9 del PDF.
4. **Interpolación Lineal (lerp):** Finalmente, se utiliza la función nativa `lerp` de Godot para calcular el valor intermedio entre los valores de las dos keys ( $v_i$  y  $v_{next}$ ) usando el factor  $u$ . Como indica el enunciado, esto funciona porque los valores son de tipos simples interpolables.

### Problema 9.3:

Para implementar la selección usando intersecciones es necesario calcular el rayo que tiene como origen el observador y pasa por centro del pixel donde se ha hecho click. Escribe el pseudo-código del algoritmo que calcula el rayo a partir de las coordenadas del pixel donde se ha hecho click:

- Tenemos una vista perspectiva, y conocemos los 6 valores  $l, r, t, b, n, f$  usados para construir la matriz de proyección.
- También conocemos el marco de coordenadas de vista, es decir, las tuplas  $x_{ec}, y_{ec}$  y  $z_{ec}$  con los versores y la tupla  $o_{ec}$  con el punto origen (todos en coordenadas del mundo).
- El viewport tiene  $w$  columnas y  $f$  filas de pixels. Se ha hecho click en el pixel de coordenadas enteras  $x_p$  e  $y_p$ .

El algoritmo debe producir como salida las tuplas  $o$  y  $d$  (normalizado) que definen el rayo.

### Algoritmo: Cálculo del Rayo de Selección

#### Entrada:

- Dimensiones del viewport:  $w$  (ancho/columnas),  $h$  (alto/filas, nota: usaré  $h$  en lugar de  $f$  para evitar confusión con el plano far).
- Pixel clicado:  $x_p, y_p$  (enteros).
- Parámetros de proyección:  $l$  (left),  $r$  (right),  $b$  (bottom),  $t$  (top),  $n$  (near).
- Marco de la cámara (Mundo): Origen  $o_{ec}$ , Ejes  $x_{ec}, y_{ec}, z_{ec}$  (normalizados).

#### Salida:

- Origen del rayo:  $o$  (Vector3)
- Dirección del rayo:  $d$  (Vector3 normalizado)

#### Pseudocódigo:

text

```
funcion CalcularRayoSelección(xp, yp, w, h, l, r, b, t, n, o_ec, x_ec, y_ec, z_ec):
    // 1. Calcular el centro del pixel en coordenadas normalizadas [0, 1]
    // Se suma 0.5 para tomar el centro geométrico del pixel.
    u = (xp + 0.5) / w
    v = (yp + 0.5) / h

    // 2. Calcular las coordenadas en el plano de proyección (Plano Near)
    // Interpolamos entre los límites l (izquierda) y r (derecha) para X.
    x_local = l + u * (r - l)

    // Para Y, depende del sistema de coordenadas del origen de pixels.
    // CASO A: Si el pixel (0,0) está arriba a la izquierda (típico en pantallas):
    // y_local = t + v * (b - t) // Vamos de top (t) hacia bottom (b)

    // CASO B: Si el pixel (0,0) está abajo a la izquierda (típico cartesiano):
    y_local = b + v * (t - b) // Vamos de bottom (b) hacia top (t)

    // Asumiremos CASO A (común en interacción de ratón):
    y_local = t - v * (t - b)

    // 3. Construir el vector de dirección en el Espacio de Vista (View Space)
    // El observador está en (0,0,0) local y mira hacia -z local.
    // El plano de proyección está a una distancia 'n' en dirección -z.
    // Por tanto, el punto destino local es (x_local, y_local, -n).
    d_local_x = x_local
    d_local_y = y_local
    d_local_z = -n

    // 4. Transformar la dirección a Coordenadas del Mundo
    // Usamos la base de la cámara para rotar el vector local al mundo.
    // d_mundo = d_local.x * Xec + d_local.y * Yec + d_local.z * Zec
    d_mundo = (d_local_x * x_ec) + (d_local_y * y_ec) + (d_local_z * z_ec)

    // 5. Definir el resultado
    // El origen del rayo es la posición de la cámara
    o = o_ec

    // La dirección debe estar normalizada
    d = Normalizar(d_mundo)

    retornar o, d
```

#### Explicación de los pasos:

1. Centro del Pixel: Los píxeles son áreas discretas. Para obtener precisión, calculamos el rayo que pasa por el centro exacto sumando 0.5 a las coordenadas enteras  $x_p, y_p$ .
2. Mapeo al Plano Near: Convertimos la posición relativa del pixel (0 a 1) a dimensiones físicas en el plano de la cámara usando los límites del frustum ( $l, r, b, t$ ).
3. Vector Local: En el espacio local de la cámara, el ojo está en el origen  $(0, 0, 0)$ . El rayo va desde el ojo hasta el punto en el plano near. Como la cámara mira hacia el eje  $-Z$  (convención estándar), la coordenada  $Z$  del punto en el plano es  $-n$ .
4. Transformación al Mundo: Multiplicamos las componentes del vector local por los vectores base de la cámara ( $x_{ec}, y_{ec}, z_{ec}$ ) para orientar el rayo en el espacio universal.
5. Normalización: Finalmente, normalizamos el vector dirección  $d$  para que tenga longitud 1, como requiere el enunciado.

# Tema 10

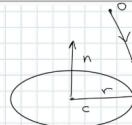
## Problema 5.1.

Supongamos que un rayo (una semirecta en 3D) tiene como origen o extremo el punto cuyas coordenadas del mundo es la tupla  $\mathbf{o}$ , y como vector de dirección la tupla  $\mathbf{d}$  (la suponemos normalizada).

Además sabemos que un disco de radio  $r$  tiene como centro el punto de coordenadas de mundo  $\mathbf{c}$  y está en el plano perpendicular al vector  $\mathbf{n}$ .

Con estos datos de entrada, diseña un algoritmo para calcular si hay intersección entre el rayo y el disco.

(ten en cuenta las indicaciones que hay en la siguiente transparencia).



```
#función IntersecciónRayoCírculo(o: Vector3, d: Vector3, n: Vector3, r: float, c: Vector3)
-> Variant:
    Var epsilon = 0.0001

    # Ver si el rayo es paralelo al círculo (perpendicular a la normal)
    var denominador = n.dot(d)
    if abs(denominador) < epsilon:
        return null

    # Calcular parámetro de intersección t
    var t = (c - o).dot(n) / denominador

    # Si t negativo, está por detrás y no hay intersección
    if t < 0:
        return null

    # Calculo la intersección
    var p = o + d * t

    # Comprobar que el punto está dentro del disco
    if p.distance_to(c) < r:
        return p # Retorna el punto de intersección
    else:
        return null # No hubo intersección válida
```

## Problema: intersección rayo-esfera

### Problema 10.2.

Diseña un algoritmo para calcular la primera intersección entre un rayo (con origen en  $\mathbf{o}$  y vector  $\mathbf{d}$ , normalizado) y una esfera de radio unidad y centro en el origen, si hay alguna.

Ten en cuenta que un punto cualquiera  $\mathbf{p}$  está en esfera si y solo si el módulo de  $\mathbf{p}$  es la unidad, es decir, si y solo si  $F(\mathbf{p}) = 0$ , donde  $F$  es el campo escalar definido así:

$$F(\mathbf{p}) \equiv \mathbf{p} \cdot \mathbf{p} - 1$$

Describe como podría usarse ese mismo algoritmo para calcular la intersección con una esfera con centro y radio arbitrarios (este problema puede reducirse al anterior si el rayo se traslada a un espacio de coordenadas donde la esfera tiene centro en el origen y radio unidad).

En el caso de centro y radio arbitrario,

```
función IntersecciónRayoEsferaArbitraria(o, d, c, r):
    # Paso 1: Trasladar el origen del rayo
    o' = o - c

    # Paso 2: Escalar la dirección del rayo
    d' = d / r

    # Paso 3: Llamar al algoritmo de intersección con la esfera de radio 1 y centro en el origen
    retornar IntersecciónRayoEsfera(o', d')
```

## Problema 5.1. (continuación)

Ten en cuenta que habrá intersección si y solo si se cumplen cada una de estas dos condiciones:

1. El rayo interseca con el plano que contiene al triángulo, es decir, existe  $t > 0$  tal que el punto  $\mathbf{p}_t \equiv \mathbf{o} + \mathbf{d}t$  está en dicho plano. Equivale a decir que el vector  $\mathbf{p}_t - \mathbf{v}_0$  es perpendicular a la normal al plano  $\mathbf{n}$ .
2. El punto  $\mathbf{p}_t$  citado arriba está dentro del disco, es decir, su distancia a  $\mathbf{c}$  es inferior al radio.

1. Calcular el punto de intersección entre el rayo y el plano: Sabemos que el disco está en un plano perpendicular a  $\mathbf{n}$ . El primer paso es determinar si el rayo interseca el plano en el que reside el disco.

La ecuación general del plano es:

$$(\mathbf{P} - \mathbf{c}) \cdot \mathbf{n} = 0$$

Donde  $\mathbf{P}$  es cualquier punto en el plano,  $\mathbf{c}$  es el centro del disco y  $\mathbf{n}$  es el vector normal del plano.

La ecuación paramétrica del rayo es:

$$\mathbf{P}(t) = \mathbf{o} + t \cdot \mathbf{d}$$

Donde  $\mathbf{o}$  es el origen del rayo y  $\mathbf{d}$  es la dirección del rayo (un vector normalizado),  $t$  es un parámetro que indica cuán lejos está un punto del origen a lo largo del rayo.

Sustituyendo  $\mathbf{P}(t)$  en la ecuación del plano, obtenemos:

$$(\mathbf{o} + t \cdot \mathbf{d} - \mathbf{c}) \cdot \mathbf{n} = 0$$

2. Resolver para  $t$ : Resolviendo la ecuación anterior, podemos encontrar el valor de  $t$ , el parámetro de la intersección del rayo con el plano:

$$t = \frac{(\mathbf{c} - \mathbf{o}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

$$\text{Ecuación esfera con radio 1: } F(\mathbf{p}) = \mathbf{p} \cdot \mathbf{p} - 1 = 0$$

$$\text{Ecuación del rayo: } \mathbf{P}(t) = \mathbf{o} + t\mathbf{d}$$

$$\text{Intersección Rayo - Esfera: } F(\mathbf{r}(t)) = (\mathbf{o} + t\mathbf{d})(\mathbf{o} + t\mathbf{d}) - 1 \Rightarrow \mathbf{o}^2 + t^2\mathbf{d}^2 + 2\mathbf{o} \cdot \mathbf{d} - 1 \Rightarrow t^2 + 2\mathbf{o} \cdot \mathbf{d} + (\mathbf{o}^2 - 1) = 0$$

$$t = \frac{-2\mathbf{o} \cdot \mathbf{d} \pm \sqrt{4\mathbf{o}^2\mathbf{d}^2 - 4(\mathbf{o}^2 - 1)}}{2} = -\mathbf{o} \cdot \mathbf{d} \pm \sqrt{\mathbf{o}^2\mathbf{d}^2 - 4\mathbf{o}^2 - 1}$$

• Si el discriminante negativo, no hay intersección.

• Si  $\mathbf{o} \cdot \mathbf{d} = 0 \Rightarrow 2$  soluciones

• Si  $\mathbf{o} \cdot \mathbf{d} > 0 \Rightarrow 2$  soluciones. La más pequeña es la que nos quedamos

función IntersecciónRayoEsfera(o, d):

# Paso 1: Calcular los coeficientes de la ecuación cuadrática

```
A = 1
B = 2 * (o · d)
C = (o · o) - 1
```

# Paso 2: Calcular el discriminante

discriminante = B^2 - 4 \* A \* C

si discriminante < 0:

retornar "No hay intersección"

# Paso 3: Calcular las dos soluciones de la ecuación cuadrática

t1 = (-B - sqrt(discriminante)) / (2 \* A)

t2 = (-B + sqrt(discriminante)) / (2 \* A)

# Paso 4: Determinar la primera intersección

si t1 > 0:

retornar t1 # Primer punto de intersección

si t2 >= 0:

retornar t2 # Segundo punto de intersección, si el primero es negativo

retornar "No hay intersección" # Ambos puntos están detrás del origen

### Cono

$$\frac{x^2 + y^2}{z^2} = 1 \quad 0 \leq z \leq 1$$

$$F(x, y, z) = x^2 + y^2 - z^2 = 0$$

$$F(r(t)) = (0_x + t d_x)^2 + (0_y + t d_y)^2 - (0_z + t d_z)^2 = 0$$

$$A = d_x^2 + d_y^2 - d_z^2$$

$$B = 2 * (0_x * d_x + 0_y * d_y - 0_z * d_z)$$

$$C = 0_x^2 + 0_y^2 - 0_z^2$$

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

función IntersecciónRayoCono(o, d):

# Paso 1: Calcular los coeficientes de la ecuación cuadrática

$$A = d_x^2 + d_y^2 - d_z^2$$

$$B = 2 * (0_x * d_x + 0_y * d_y - 0_z * d_z)$$

$$C = 0_x^2 + 0_y^2 - 0_z^2$$

# Paso 2: Calcular el discriminante

$$\text{discriminante} = B^2 - 4 * A * C$$

si discriminante < 0:

retornar "No hay intersección" # El rayo no intersecta el cono

# Paso 3: Calcular las soluciones de la ecuación cuadrática

$$t1 = (-B - \sqrt{\text{discriminante}}) / (2 * A)$$

$$t2 = (-B + \sqrt{\text{discriminante}}) / (2 * A)$$

# Paso 4: Verificar que la intersección esté dentro del cono ( $0 \leq z \leq 1$ )

$$z1 = 0_z + t1 * d_z$$

$$z2 = 0_z + t2 * d_z$$

si  $0 \leq z1 \leq 1$ :

retornar t1 # Primer punto de intersección

si  $0 \leq z2 \leq 1$ :

retornar t2 # Segundo punto de intersección

retornar "No hay intersección" # Ninguna intersección está dentro del cono

## Problema: intersección rayo-cilindro y rayo-cono

### Problema 10.3.

Describe como podemos definir el campo escalar cuyos ceros son los puntos en un cilindro con altura unidad y radio unidad (sin considerar los discos que forman la base ni la tapa).

Usando esa definición diseña el algoritmo para calcular la intersección rayo-cilindro.

Describe asimismo el campo escalar y el algoritmo correspondientes a un cono de altura unidad y radio de la base unidad (sin considerar el disco de la base).

### Caso Utilizado



$$F(p) = F(x, y, z)$$

$$\nabla F(p) \cdot x^2 + z^2 = 0$$

$$x = 0_x + t d_x \quad \left. \begin{array}{l} \\ z = 0_z + t d_z \end{array} \right\} \text{Intersección rayo - cilindro: extensión infinita}$$

Si es extensión finita, ver si la altura se queda dentro de las coordenadas y

$$r(t) = 0 + t d$$

$$F(r(t)) = (0_x + t d_x)^2 + (0_y + t d_y)^2 - 1 = 0$$

$$0_x^2 + t^2 d_x^2 + 2t 0_x d_y + 0_y^2 + t^2 d_y^2 + 2 0_y t d_y =$$

$$(d_x^2 + d_y^2) t^2 + 2 (0_x d_x + 0_y d_y) t + (0_x^2 + 0_y^2 - 1)$$

$$\left| \begin{array}{l} A = d_x^2 + d_y^2 \\ B = 0_x d_x + 0_y d_y \\ C = 0_x^2 + 0_y^2 - 1 \end{array} \right. \quad t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Copiar

función IntersecciónRayoCilindro(o, d):

# Paso 1: Calcular los coeficientes de la ecuación cuadrática

$$A = d_x^2 + d_y^2$$

$$B = 2 * (0_x * d_x + 0_y * d_y)$$

$$C = 0_x^2 + 0_y^2 - 1$$

# Paso 2: Calcular el discriminante

$$\text{discriminante} = B^2 - 4 * A * C$$

si discriminante < 0:

retornar "No hay intersección" # El rayo no intersecta la superficie lateral

# Paso 3: Calcular las soluciones de la ecuación cuadrática

$$t1 = (-B - \sqrt{\text{discriminante}}) / (2 * A)$$

$$t2 = (-B + \sqrt{\text{discriminante}}) / (2 * A)$$

# Paso 4: Verificar que la intersección esté dentro del cilindro ( $0 \leq z \leq 1$ )

$$z1 = 0_z + t1 * d_z$$

$$z2 = 0_z + t2 * d_z$$

si  $0 \leq z1 \leq 1$ :

retornar t1 # Primer punto de intersección

si  $0 \leq z2 \leq 1$ :

retornar t2 # Segundo punto de intersección

retornar "No hay intersección" # Ninguna intersección está dentro del cilindro

# Tema 13

## Problema: curva Hermite para una trayectoria

### Problema 11.1:

Implementar un proyecto en Godot en el cual el nodo raíz tiene un script que define dos arrays con: una serie de  $n$  puntos  $p_0, p_1, \dots, p_{n-1}$  del plano  $y = 0$ , y; una serie de instantes de tiempo  $t_0, t_1, \dots, t_{n-1}$  (en segundos) con  $t_0 = 0$

- Sitúa en cada uno de esos puntos un disco pequeño visible, a modo de marcador.
- Incluye una función para calcular la posición y velocidad de la curva de Hermite que pasa por los puntos en los instantes dados, a partir de un  $t$  en  $[0, t_{n-1}]$ . En cada punto  $p_i$  el vector de velocidad  $v_i$  se calcula a partir de los puntos anterior y siguiente (como en el ejemplo anterior).
- En el método `_process(delta)` del script, calcula la posición y velocidad de la curva en el tiempo transcurrido desde el inicio, y mueve un objeto (un coche, por ejemplo) a esa posición, alineado con la dirección de la curva (usando el vector de velocidad como vector de dirección).

### Solución en GDScript

Crea una escena nueva en Godot, añade un nodo raíz (por ejemplo un Node3D), añade un hijo que sea el objeto a mover (por ejemplo un MeshInstance3D llamado "Coche") y adjunta este script al nodo raíz.

gdscript

```
# --- CONFIGURACIÓN DE DATOS ---
# Array de puntos (Vector3) en el plano y=0
var puntos: Array = []
# Array de instantes de tiempo (segundos)
var tiempos: Array<float> = []

# Referencia al objeto que vamos a mover
#&onready var coche = $Coche # Asegúrate de tener un nodo llamado "Coche"

# Variable para llevar la cuenta del tiempo transcurrido
var tiempo_actual: float = 0.0

func ready():
    # 1. Inicializamos los datos (Ejemplo: una curva en forma de S)
    puntos = [
        Vector3(0, 0, 0),
        Vector3(0, 0, 2),
        Vector3(0, 2, 2),
        Vector3(0, 2, 0),
        Vector3(0, 0, 0),
        Vector3(0, 0, 5)
    ]

    # Definimos los tiempos para cada punto (t0 siempre 0)
    tiempos = [0.0, 2.0, 4.0, 6.0, 8.0]

    # 2. Situar marcadores visuales (discos)
    crear_marcadores()

func process(delta):
    # 3. Gestión del tiempo
    tiempo_actual += delta

    # Si llegamos al final, reiniciamos (loop) para ver la animación continua
    if tiempo_actual > tiempos.size() - 1:
        tiempo_actual = 0.0

    # 4. Calcular posición y velocidad
    var resultado = calcular_posicion(tiempo_actual)
    var velocidad = resultado["velocidad"]
    var posicion = resultado["posicion"]

    # 5. Mover el objeto
    if coche:
        coche.global_position = posicion

    # Orientar el coche según el vector de velocidad (mirar hacia adelante)
    # Verificamos que la velocidad no sea casi cero para evitar errores
    if velocidad.length_squared() > 0.001:
        coche.look_at(posicion + velocidad, Vector3.UP)
```

# --- LÓGICA MATEMÁTICA ---

```
func crear_marcadores():
    # Crea pequeños cilindros planos (discos) en cada punto P
    for p in puntos:
        var marcador = CCSylinder3D.new()
        marcador.radius = 0.2
        marcador.height = 0.05
        marcador.material = StandardMaterial3D.new()
        marcador.material.albedo_color = Color.RED
        add_child(marcador)
        marcador.global_position = p

func calcular_velocidad_en_punto(indice: int) -> Vector3:
    # El enunciado pide calcular  $v_i$  basándose en el anterior y el siguiente.
    # Esto es típico de las Splines de Catmull-Rom.
    # Fórmula:  $v_i = (P_{next} - P_{prev}) / (t_{next} - t_{prev})$ 
    var n = puntos.size()

    # Casos extremos (inicio y fin): Velocidad 0 o dirección hacia el vecino
    if indice == 0:
        return (puntos[1] - puntos[0]) / (tiempos[1] - tiempos[0])
    if indice == n - 1:
        return (puntos[n-1] - puntos[n-2]) / (tiempos[n-1] - tiempos[n-2])

    # Caso general (puntos intermedios)
    var p_prev = puntos[indice - 1]
    var p_next = puntos[indice + 1]
    var t_prev = tiempos[indice - 1]
    var t_next = tiempos[indice + 1]

    return (p_next - p_prev) / (t_next - t_prev)

func calcular_hermite(t: float) -> Dictionary:
    # 1. Encuentra el segmento activo:  $t_{i-1} \leq t \leq t_{i+1}$ 
    var k = 0
    while k < tiempos.size() - 1 and t > tiempos[k]:
        k += 1

    # Datos del segmento
    var t0 = tiempos[k]
    var t1 = tiempos[k+1]
    var p0 = puntos[k]
    var p1 = puntos[k+1]

    # Velocidades (Tangentes) en los extremos del segmento
    var v0 = calcular_velocidad_en_punto(k)
    var v1 = calcular_velocidad_en_punto(k+1)

    # 2. Calcular  $u$  (tiempo normalizado entre 0 y 1 para este segmento)
    var delta_t = t - t0
    if delta_t == 0: return {"posicion": p0, "velocidad": Vector3.ZERO}

    var u = (t - t0) / delta_t
    var u2 = u * u
    var u3 = u2 * u

    # 3. Polinomios de Hermite (Funciones base)
    # H0 = 2*u^3 - 3*u^2 + 1
    # H1 = -2*u^3 + 3*u^2
    # H2 = u^3 - 2*u^2 + u
    # H3 = u^3 - u^2

    var h0 = 2*u3 - 3*u2 + 1
    var h1 = -2*u3 + 3*u2
    var h2 = u3 - 2*u2 + u
    var h3 = u3 - u2

    # 4. Cálculo de la POSICIÓN final
    # Nota: Multiplicamos  $v$  por  $\delta t$  para ajustar la escala temporal al dominio [0,1]
    var pos = h0*p0 + h1*p1 + h2*v0 * delta_t + h3*v1 * delta_t

    # 5. Cálculo de la VELOCIDAD instantánea (Derivada de la posición)
    # Derivadas de las bases respecto a  $u$ 
    # dH0 = -6*u^2 + 6u
    # dH1 = -6*u^2 + 6u
    # dH2 = 3*u^2 - 4u + 1
    # dH3 = 3*u^2 - 2u

    var dh0 = -6*u2 + 6*u
    var dh1 = -6*u2 + 6*u
    var dh2 = 3*u2 - 4*u + 1
    var dh3 = 3*u2 - 2*u

    # La derivada calculada es  $dR/du$ . Para tener  $dP/dt$  (velocidad real), dividimos por  $\delta t$ 
    # Al multiplicar las tangentes originales por  $\delta t$  en la fórmula, se simplifica.
    var vel = (dh0*p0 + dh1*p1)/delta_t + dh2*v0 + dh3*v1

    return {"posicion": pos, "velocidad": vel}
```

### Explicación teórica de la implementación

#### 1. Cálculo de Tangentes ('v'):

El problema pide calcular la velocidad basándose en el punto anterior y siguiente. Matemáticamente, esto se approxima con la diferencia finita centrada:

$$v_i = \frac{P_{i+1} - P_{i-1}}{t_{i+1} - t_{i-1}}$$

Esto asegura que la curva pase suavemente por los puntos sin cambios bruscos de dirección.

#### 2. Interpolación de Hermite:

Para encontrar un punto intermedio entre  $P_k$  y  $P_{k+1}$ , primero normalizamos el tiempo global  $t$  a una variable local  $u \in [0, 1]$ .

$$u = \frac{t - t_k}{t_{k+1} - t_k}$$

Luego aplicamos la base de Hermite cúbica. Es crucial notar que en la fórmula estándar de Hermite, las tangentes deben escalarse por la duración del intervalo ( $\Delta t$ ) porque el dominio paramétrico  $u$  va de 0 a 1, mientras que el tiempo real va de  $t_k$  a  $t_{k+1}$ .

#### 3. Orientación (look\_at):

El problema pide alinear el coche con la curva. Como la derivada de la curva de posición nos da el vector de velocidad (tangente), usamos ese vector para decirle a Godot hacia dónde debe "mirar" el coche: `look_at(posicion_actual + vector_velocidad)`.