



**UNIVERSIDAD  
DE GRANADA**

Informática Gráfica (curso 2025-26)  
**Guiones de Prácticas**

Dpt. Lenguajes y Sistemas Informáticos.  
ETSI Informática y de Telecomunicación.

# Índice

1. Escena básica y modos de visualización .....	6
1.1. Objetivos .....	6
1.2. Requisitos previos .....	6
1.3. Actividades .....	6
1.3.1. Crear un nuevo proyecto Godot .....	7
1.3.2. Crear un cubo en la escena .....	7
1.3.3. Añadir ejes de coordenadas .....	8
1.3.4. Añadir cámara .....	9
1.3.5. Asignar materiales .....	9
1.3.6. Añadir luz .....	11
1.3.7. Crea una pirámide .....	12
1.3.8. Cambiar el material para que se vea rojo.....	14
1.3.9. Controlar una cámara orbital por teclado y ratón .....	14
1.4. Entrega de la práctica .....	16
2. Carga de modelos externos y normales .....	17
2.1. Objetivos .....	17
2.2. Requisitos previos .....	17
2.3. Actividades .....	17
2.3.1. Añadir modo de visualización en alambre ( <i>wireframe</i> ) .....	17
2.3.2. Cargar modelos 3D en formato <i>glb</i> .....	18
2.3.3. Cargar modelos 3D en formato <i>obj</i> .....	20
2.3.4. Cálculo de normales de objetos <i>suaves</i> y tipos de sombreado de Godot. .	20
2.3.5. Normales de objetos con aristas reales ( <i>no suaves</i> ).....	23
2.3.6. Creación de mallas por revolución de un perfil.....	25
2.4. Entrega de la práctica .....	26
3. Grafos de escena .....	27
3.1. Objetivos .....	27
3.2. Requisitos previos y recomendaciones .....	27
3.3. Actividades .....	28
3.3.1. Diseñar el grafo de escena .....	28
3.3.2. Crear el modelo en Godot .....	29
3.3.3. Generar una animación del modelo .....	30
3.3.4. Activar y desactivar la animación .....	30
3.4. Entrega de la práctica .....	30
4. Iluminación, materiales y texturas .....	31
4.1. Objetivos .....	31
4.2. Requisitos previos .....	31
4.3. Actividades .....	31
4.3.1. Creación de la escena .....	31
4.3.2. Iluminación .....	32
4.3.3. Materiales .....	32
4.3.4. Texturas .....	33

4.3.5. Textura en el objeto de revolución.....	34
4.4. Entrega de la práctica .....	35
5. Interacción con ratón y selección de objetos .....	38
5.1. Objetivos .....	38
5.2. Requisitos previos .....	38
5.3. Actividades .....	38
5.3.1. Creación de escena base .....	38
5.3.2. Añadir colisionadores .....	38
5.3.3. Creación de de un nodo <i>RayCast3D</i> .....	39
5.3.4. Selección de objetos .....	40
5.3.5. Lectura de posiciones con el ratón y creación de objetos .....	40
5.3.6. Interacción con el modelo jerárquico .....	41
5.4. Entrega de la práctica .....	41

# Listado de figuras

Figura 1	Crear proyecto .....	7
Figura 2	Crear escena .....	7
Figura 3	Añadiendo cubo .....	8
Figura 4	Ejes de coordenadas .....	8
Figura 5	Cámara añadida a la escena .....	9
Figura 6	Asignación de material .....	10
Figura 7	Cambiando el color .....	10
Figura 8	Ejecución tras añadir cámara .....	11
Figura 9	Fuente de luz direccional añadida a la escena .....	11
Figura 10	Ejecución con cámara y luz direccional .....	12
Figura 11	Código del <i>script</i> para la pirámide de la práctica 1 (en el archivo <code>piramide.gd</code> ) .....	12
Figura 12	Creación de un script para la pirámide .....	12
Figura 13	Ejecución con pirámide sobre el cubo .....	13
Figura 14	Ejecución con pirámide sobre el cubo .....	14
Figura 15	Código del <i>script</i> para la cámara orbital (en el archivo <code>camara_3d_orbital_simple.gd</code> ) .....	16
Figura 16	Donut en modo normal (izquierda) y en modo <i>wireframe</i> (derecha). ....	17
Figura 17	Código para poder activar el modo <i>wireframe</i> con una tecla .....	18
Figura 18	Ejemplo de carga de modelo con formato <code>glb</code> .....	19
Figura 19	Ejemplo de carga de modelo con formato <code>obj</code> .....	20
Figura 20	Código del <i>script</i> para el donut, en el archivo <code>donut.gd</code> .....	20
Figura 21	Función que calcula las normales (en <code>utilidades.gd</code> ) .....	20
Figura 22	Donut con las normales calculadas por el algoritmo. ....	21
Figura 23	Donut con sombreado por pixel (izquierda) y por vértice (derecha). ....	23
Figura 24	Cubo de 8 vértices (izquierda) y de 24 vértices (derecha). ....	24
Figura 25	Ejemplo de una perfil que por revolución produce un modelo de un peón de ajedrez. ....	25
Figura 26	Aristas del objeto de revolución de un perfil. ....	25
Figura 27	Ejemplo de modelo jerárquico (izquierda), y los objetos usados (derecha). ....	28
Figura 28	Ejemplos de diversos objetos jerárquicos. ....	29
Figura 29	Escena de partida .....	32
Figura 30	Luces .....	32
Figura 31	Materiales .....	33
Figura 32	Texturas .....	34
Figura 33	Código del <i>script</i> para texturas (parte 1) .....	34
Figura 34	Código del <i>script</i> para texturas (parte 2) .....	34
Figura 35	Código a insertar en la función <code>ready</code> ) .....	35
Figura 36	Función de calculo de coordenadas de textura .....	35
Figura 37	Escena de partida .....	38
Figura 38	Código de selección con ratón en el nodo cámara .....	40
Figura 39	Creación de un objeto en el punto de click .....	41

Figura 40 Código de la función `crear_cubo_en` ..... 41

## Práctica 1.

# Escena básica y modos de visualización.

### 1.1. Objetivos

El objetivo de esta práctica es familiarizarse con el entorno de desarrollo de Godot, el sistema de nodos y la creación de escenas 3D simples. Los estudiantes aprenderán a:

- Crear una escena 3D con geometría básica (cubo y pirámide).
- Aplicar distintos materiales a los objetos.
- Implementar cambios de modo de visualización usando entrada por teclado.
- Controlar la cámara mediante teclas.

### 1.2. Requisitos previos

Para realizar esta práctica no se requiere experiencia previa en motores de juego ni gráficos 3D, pero antes de hacerla debes asegurarte de:

- Haber instalado Godot Engine (versión 4 o superior).
- Conocer los conceptos básicos de programación (estructuras de control, funciones, clases).
- Descargar los archivos de script (extensión `.gd`) que se usarán en esta práctica, son los siguientes:
  - `ejes3D.gd`: script para dibujar ejes de coordenadas en la escena.
  - `piramide.gd`: script para crear una pirámide mediante código.
  - `camara_3d_orbital_simple.gd`: script para controlar una cámara orbital con teclado y ratón.

### 1.3. Actividades

En las subsecciones a continuación se detallan cada una de las actividades a realizar, son las siguientes:

1. Crear un nuevo proyecto Godot
2. Crear un cubo en la escena
3. Añadir ejes de coordenadas
4. Añadir cámara
5. Asignar materiales
6. Añadir luz
7. Crea una pirámide
8. Cambiar el material para que se vea rojo.
9. Controlar una cámara orbital

### 1.3.1. Crear un nuevo proyecto Godot

1. Abrir Godot Engine y seleccionar “Nuevo proyecto”.
2. Asignar un nombre y una carpeta de destino.
3. Crear el proyecto con el renderizador por defecto (Forward+).

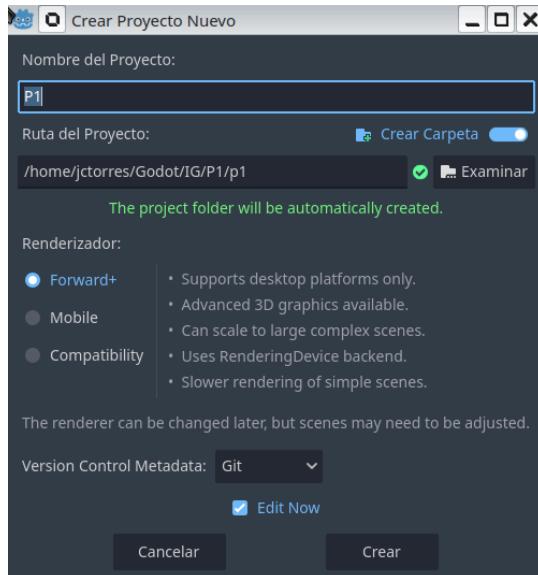


Figura 1 : Crear proyecto

4. Abrir el editor y crear una nueva escena.
5. Añadir un nodo **Node3D** como nodo raíz, y renombrarlo a **EscenaPrincipal**.

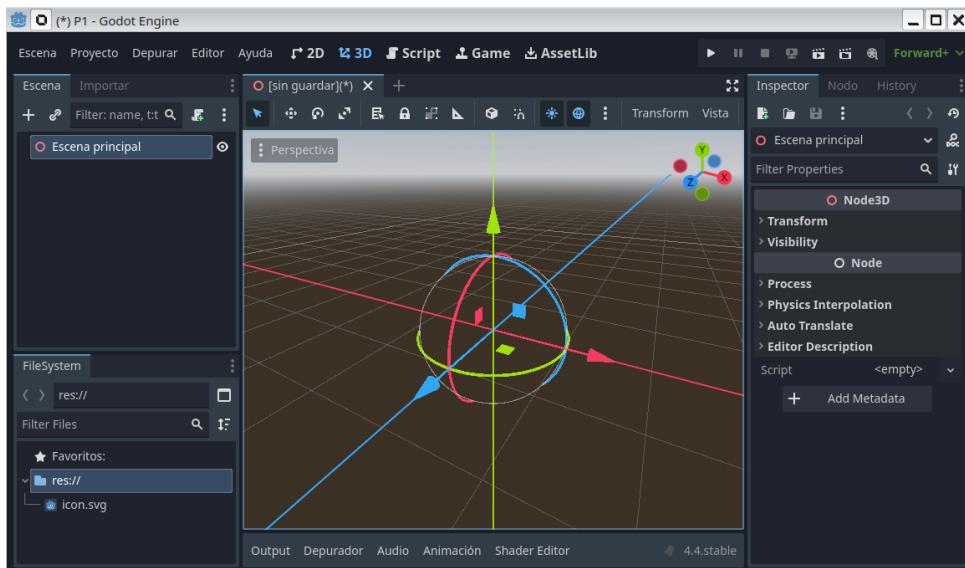


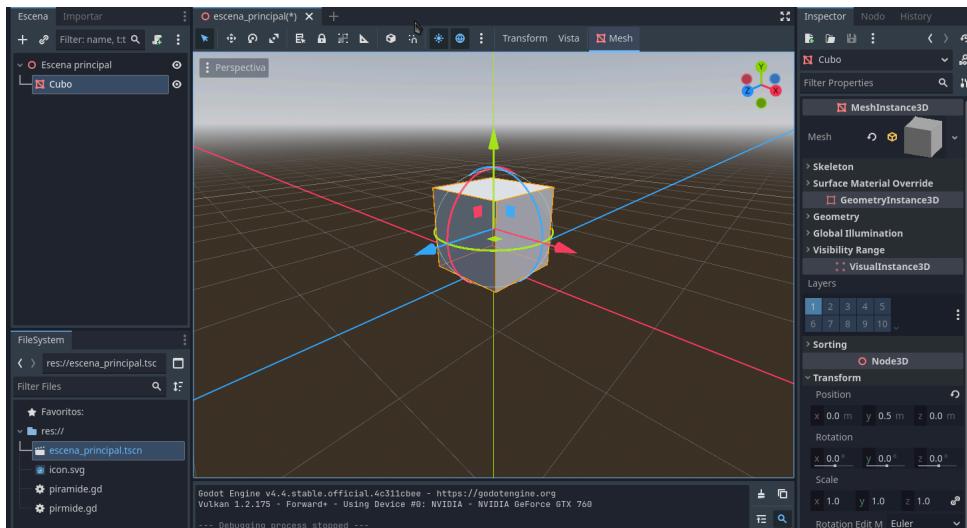
Figura 2 : Crear escena

### 1.3.2. Crear un cubo en la escena

Se añadirá un objeto de tipo malla con forma de cubo, para ello se usará la clase **CubeMesh** de Godot, que ya está predefinida en el motor.

1. Añadir un nodo **MeshInstance3D** como hijo del nodo raíz.

2. Asigna una malla **CubeMesh** desde el panel Inspector (campo **Mesh**).
3. Renombrar el nodo como **Cubo**.
4. Moverlo a la posición **(0, 0.5, 0)**.



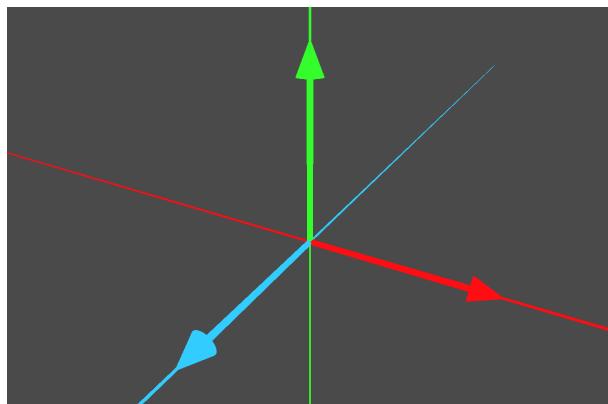
**Figura 3 : Añadiendo cubo**

### 1.3.3. Añadir ejes de coordenadas

Se añade un nodo 3D con un script que al inicio genera mallas en los ejes de coordenadas, parecidas a las que aparecen en la vista 3D del editor, sirve para entender visualmente mejor la posición y tamaño de los objetos que se añaden.

1. Añadir un nodo de tipo **Node3D** y renombrarlo como **Ejes3D**.
2. Añadir a ese nodo un *script*, y al hacerlo, asociarle el archivo existente **ejes3D.gd**.

En la siguiente imagen se observan los ejes (tienen colores a los que no le afecta la iluminación)



**Figura 4 : Ejes de coordenadas**

### 1.3.4. Añadir cámara

Para poder visualizar la escena, es necesario añadir una cámara que define la posición y orientación del observador virtual que se usa para producir la imagen. Esta cámara es muy simple, luego se añadirá otra más completa.

1. Añadir un nodo **Camera3D** como hijo del nodo raíz.
2. Colocarlo en una posición adecuada para ver toda la escena, por ejemplo en **(1.5, 1.5, 2.0)** y hacer que la cámara mire al origen **(0, 0, 0)**. Esto se puede hacer añadiéndole el siguiente script a la cámara:

```
extends Camera3D
func _ready():
    position = Vector3( 1.5, 1.5, 2.0 )
    look_at( Vector3( 0.0, 0.0, 0.0 ), Vector3.UP )
```

3. Ejecutar.

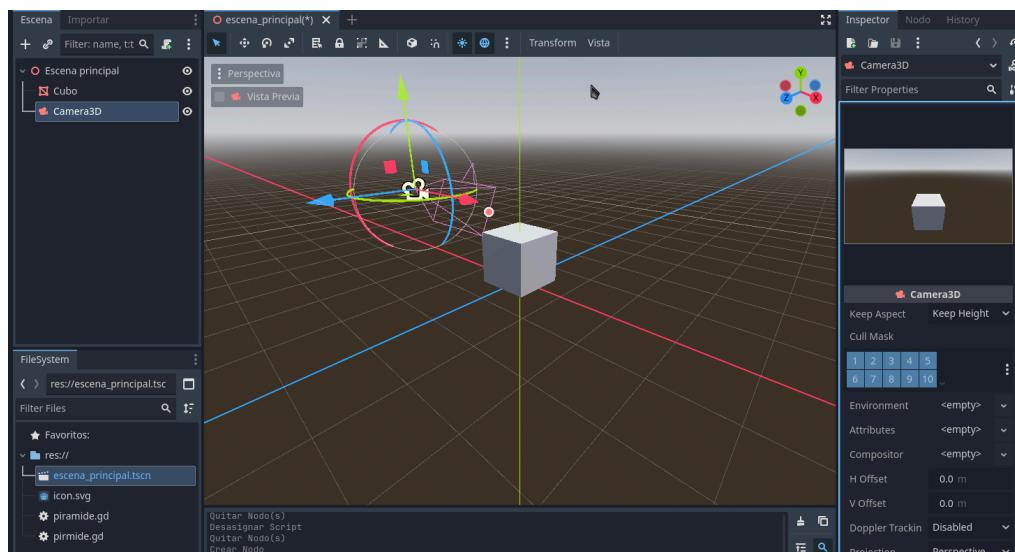


Figura 5 : Cámara añadida a la escena

### 1.3.5. Asignar materiales

1. Con el cubo seleccionado, seleccionar **StandardMaterial3D** en la entrada **Surface Material Override** del panel **Inspector**.

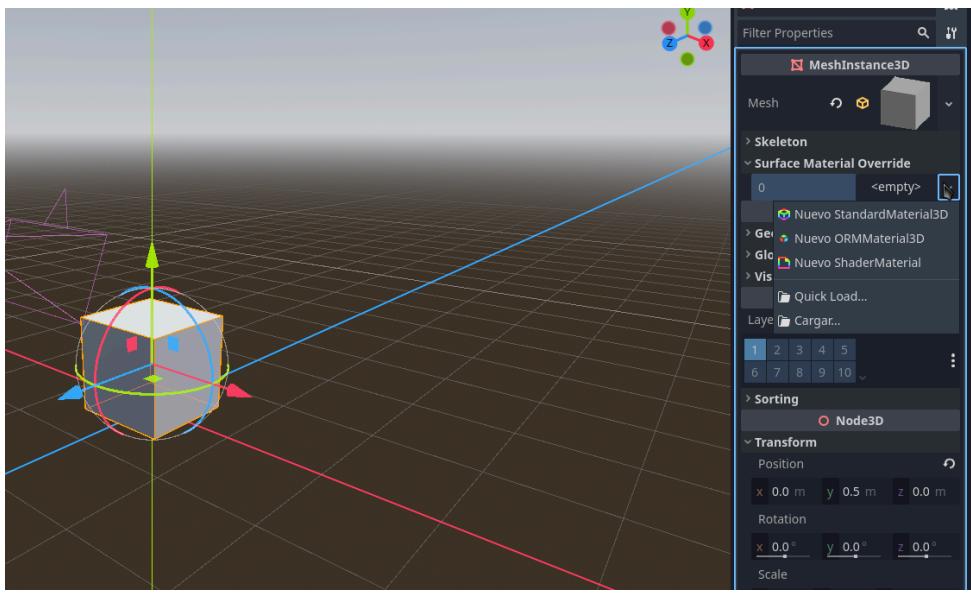


Figura 6 : Asignación de material

2. Pulsar en la esfera y modificar el color del **Albedo**, asignar un color Amarillo.

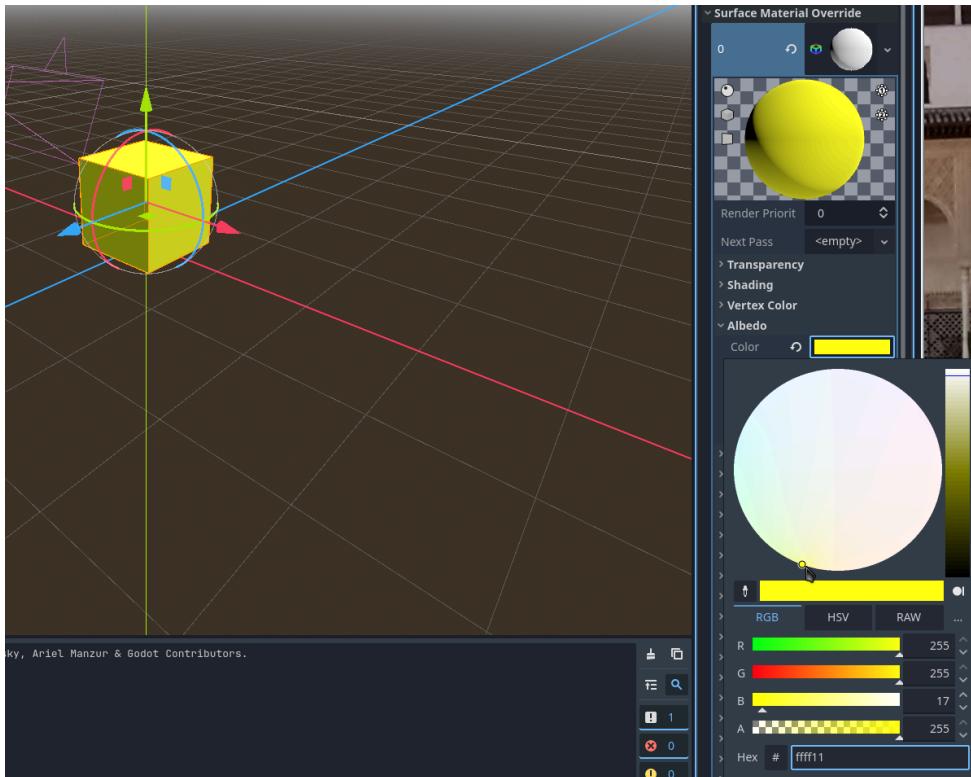


Figura 7 : Cambiando el color

3. Ejecutar.

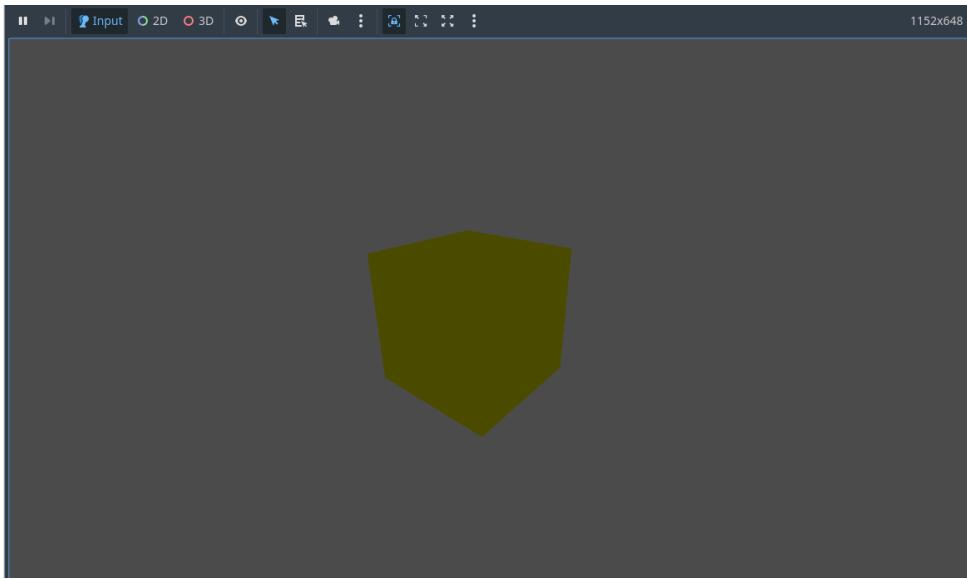


Figura 8 : Ejecución tras añadir cámara

### 1.3.6. Añadir luz

1. Añadir un nodo `DirectionalLight3D` como hijo del nodo raíz.
2. Colocarlo fuera del origen para distinguir su representación de la del cubo
3. Rotarlo para que ilumine el cubo de forma que las tres caras tengan distinta tonalidad.

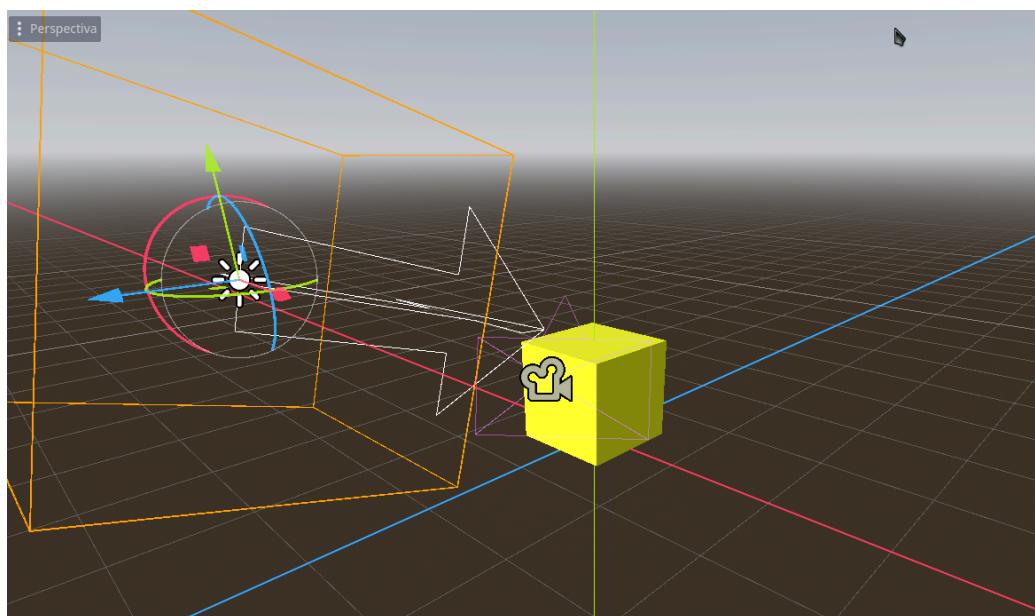


Figura 9 : Fuente de luz direccional añadida a la escena

4. Ejecutar.

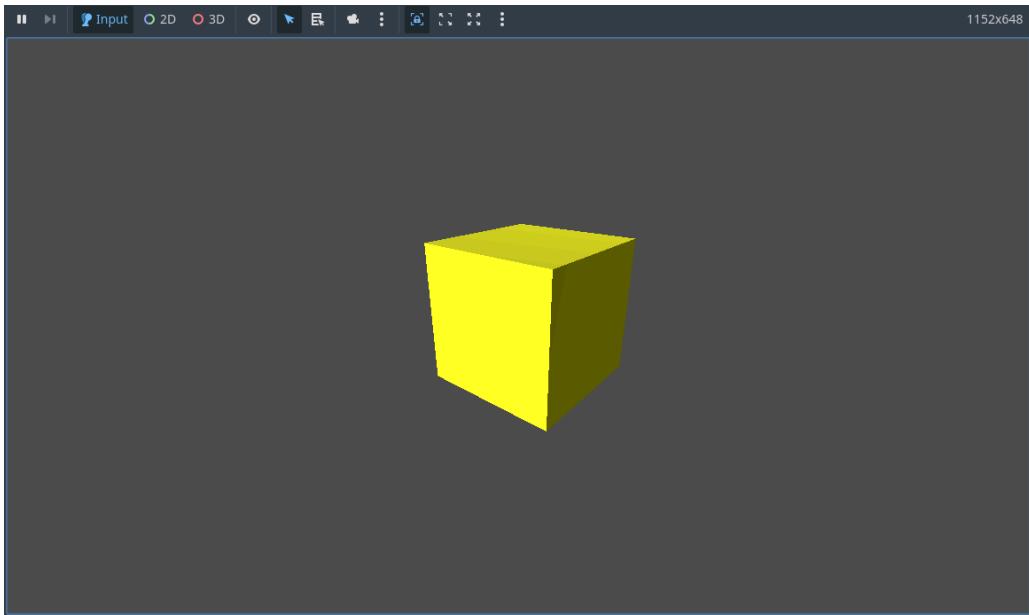


Figura 10 : Ejecución con cámara y luz direccional

### 1.3.7. Crea una pirámide

Vamos a crear una pirámide de base cuadrada añadiendo la geometría mediante código.

1. Crear un nodo hijo de tipo `Node3D`.
2. Renombra el nodo como `Piramide`.
3. Asóciale el script con el código que aparece en la figura 11, y que tienes disponible en el archivo `piramide.gd`.

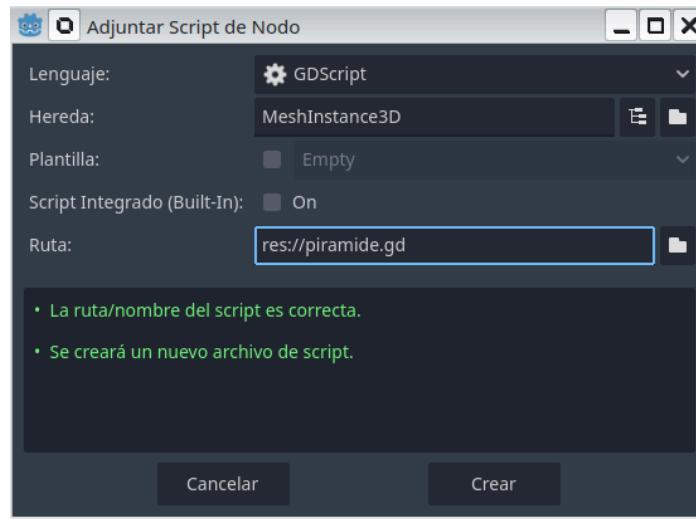


Figura 12 : Creación de un script para la pirámide

4. Posiciona la pirámide sobre el cubo, en las coordenadas `(0, 1, 0)`.
5. Modifica el parámetro `altura` en el `Inspector` asignándole valor `0.5`.
6. Ejecuta.

```

extends Node3D

@export var altura: float = 1.5

func _ready():
    var piramide = crear_piramide(altura)
    var mesh_instance = MeshInstance3D.new()
    mesh_instance.mesh = piramide
    add_child(mesh_instance)

func crear_piramide(h: float) -> ArrayMesh:
    var st = SurfaceTool.new()
    st.begin(Mesh.PRIMITIVE_TRIANGLES)

    # Coordenadas de la base (cuadrado centrado en el origen, lado 1)
    var p1 = Vector3(-0.5, 0, -0.5)
    var p2 = Vector3( 0.5, 0, -0.5)
    var p3 = Vector3( 0.5, 0, 0.5)
    var p4 = Vector3(-0.5, 0, 0.5)

    var apex = Vector3(0, h, 0) # cúspide

    # Caras laterales (triángulos)
    _add_triangulo(st, p1, p2, apex)
    _add_triangulo(st, p2, p3, apex)
    _add_triangulo(st, p3, p4, apex)
    _add_triangulo(st, p4, p1, apex)

    # Base (dos triángulos)
    _add_triangulo(st, p1, p3, p2, Vector3.DOWN)
    _add_triangulo(st, p1, p4, p3, Vector3.DOWN)

    return st.commit()

# Añade triángulo y calcula normal automáticamente
func _add_triangulo(st: SurfaceTool,
                      a: Vector3, b: Vector3, c: Vector3,
                      normal_override: Vector3 = Vector3.ZERO):
    var normal = normal_override
    if normal == Vector3.ZERO:
        normal = Plane(a, b, c).normal
    st.set_normal(normal)
    st.add_vertex(a)
    st.add_vertex(b)
    st.add_vertex(c)

```

**Figura 11 :** Código del script para la pirámide de la práctica 1 (en el archivo **piramide.gd**)

**Figura 13 :** Ejecución con pirámide sobre el cubo

### 1.3.8. Cambiar el material para que se vea rojo.

Se le cambiará el material a la pirámide para que se vea de color rojo uniforme.

1. Edita la función `_ready()` del script anterior para añadir la descripción del material:

```
func _ready():
    var piramide = crear_piramide(altura)
    var mesh_instance = MeshInstance3D.new()
    var material = StandardMaterial3D.new()
    mesh_instance.mesh = piramide
    material.albedo_color = Color(1.0, 0.1, 0.2) # Naranja claro
    mesh_instance.material_override = material
    add_child(mesh_instance)
```

2. Ejecuta

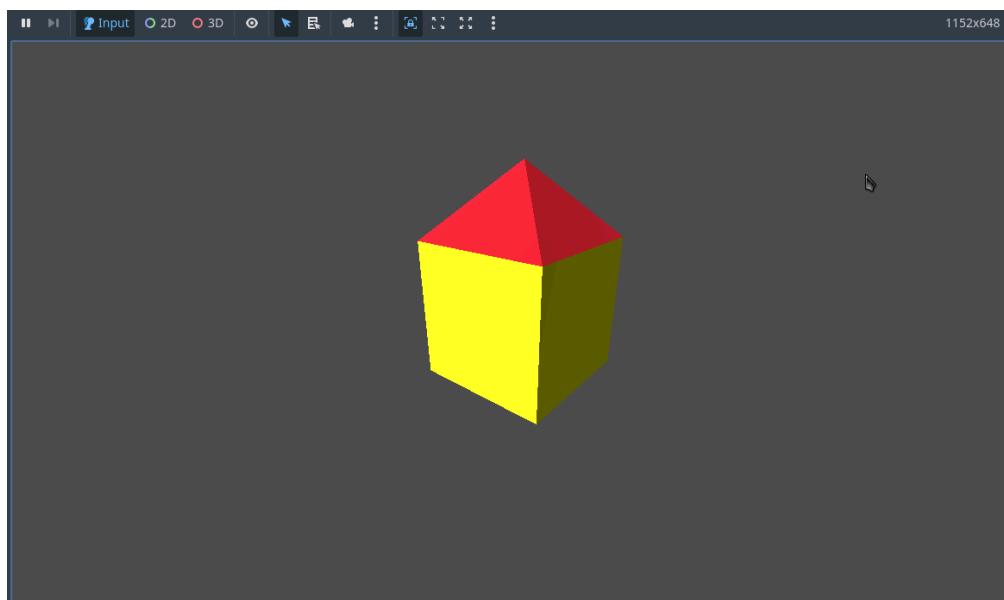


Figura 14 : Ejecución con pirámide sobre el cubo

### 1.3.9. Controlar una cámara orbital por teclado y ratón

Vamos a hacer que la cámara pueda girar alrededor del origen para mostrar la escena desde diferentes posiciones, y que se pueda controlar con ratón además del teclado.

1. Borra el nodo `Camera3D` que se añadió antes. Debes borrar el nodo y el script asociado.
2. Añade un nuevo nodo `Camera3D` como hijo del nodo raíz, y renómbarlo como `Camara3DOrbital`.
3. Añade al nuevo nodo el script que está en el archivo `camara_3d_orbital_simple.gd`, que contiene el código de la figura 15.

Comprueba que puedes controlar la cámara con el teclado y el ratón. Puedes usar:

- Teclas de cursor para orbitar horizontal y verticalmente.
- Teclas +/-, o teclas de página anterior/siguiente o rueda del ratón para acercar/alejar la cámara.

```

extends Camera3D

const at    := 2.5    ## ángulo de rot. con teclas (en grados x pulsación)
const ar    := 0.5    ## ángulo de rot. con ratón (en grados x pixel)
var bdrp := false ## botón derecho del ratón presionado sí/no
var dz    := 3.0    ## distancia en Z de la cámara al origen
var dxy   := Vector2( 0.0, 0.0 ) ## ángulos hor. y vert. (en grados)

func _actualiza_transf_vista( ) -> void :
    var ahr  := ((45.0+float(dxy.x))*2.0*PI)/360.0 ## ang.horiz.radi.
    var avr  := ((30.0+float(dxy.y))*2.0*PI)/360.0 ## ang.vert.radi.
    var tras := Transform3D().translated( Vector3( 0.0, 0.0, dz ))
    var rotx := Transform3D().rotated( Vector3.RIGHT, -avr )
    var roty := Transform3D().rotated( Vector3.UP, ahr )
    transform = roty*rotx*tras    ## actualiza transform del nodo cámara

func _ready() -> void : ## se ejecuta una vez al inicio
    _actualiza_transf_vista() ## inicializa la vista al inicio

func _input( event : InputEvent ): ## procesa evento de entrada
    var av : bool = true ## actualizar vista sí/no

    if event is InputEventKey and event.pressed: ## teclas
        match event.keycode:
            KEY_UP:    dxy += Vector2( 0, -at )
            KEY_DOWN:  dxy += Vector2( 0, +at )
            KEY_RIGHT: dxy += Vector2( -at, 0 )
            KEY_LEFT:  dxy += Vector2( at, 0 )
            KEY_MINUS, KEY_PAGEDOWN, KEY_KP_SUBTRACT: dz *= 1.05
            KEY_PLUS, KEY_PAGEUP, KEY_KP_ADD: dz = max( dz/1.05, 0.1 )
            _: av = false
    elif event is InputEventMouseButton: ## botón ratón o rueda
        match event.button_index:
            MOUSE_BUTTON_RIGHT: bdrp = event.pressed ; av = false
            MOUSE_BUTTON_WHEEL_DOWN: dz *= 1.05
            MOUSE_BUTTON_WHEEL_UP:   dz = max( dz/1.05, 0.1 )
            _: av = false
    elif event is InputEventMouseMotion and bdrp: ## movim. ratón
        dxy += ar * Vector2( -event.relative.x, event.relative.y )
    else: ## otros tipos de eventos (no hace nada)
        av = false ## (no actualizar transf)

    if av: _actualiza_transf_vista( )

```

**Figura 15 :** Código del script para la cámara orbital (en el archivo `camara_3d_orbital_simple.gd`)

- Botón derecho del ratón presionado y arrastrar para orbitar con el ratón.

## 1.4. Entrega de la práctica

- Subir la carpeta del proyecto **godot** con todos los archivos necesarios comprimido en un zip.
- Analiza los scripts y contesta el test de la práctica en Prado.

## Práctica 2.

# Carga de modelos externos y normales.

### 2.1. Objetivos

El objetivo de esta práctica es:

- Comprender la representación de mallas triangulares.
- Aprender a cargar y visualizar modelos 3D externos en Godot.
- Entender la diferencia entre los tipos de sombreado de Godot (por vértice y por pixel).
- Aprender a generar normales mediante scripts cuando el modelo no las incluye.
- Crear una malla mediante revolución de un perfil 2D.

### 2.2. Requisitos previos

- Haber realizado la práctica 1.
- Crear un proyecto nuevo, creando el nodo raiz, una fuente de luz y la cámara orbital creada en la práctica anterior.
- Disponer de los mismos archivos `.gd` descritos en los requisitos de la práctica 1.
- Disponer de los archivos `script_raiz.gd`, `utilidades.gd` y `donut.gd` que se necesitan para la práctica y se mencionan en las actividades.

### 2.3. Actividades

En las siguientes subsecciones se detallan las actividades a realizar, son las siguientes:

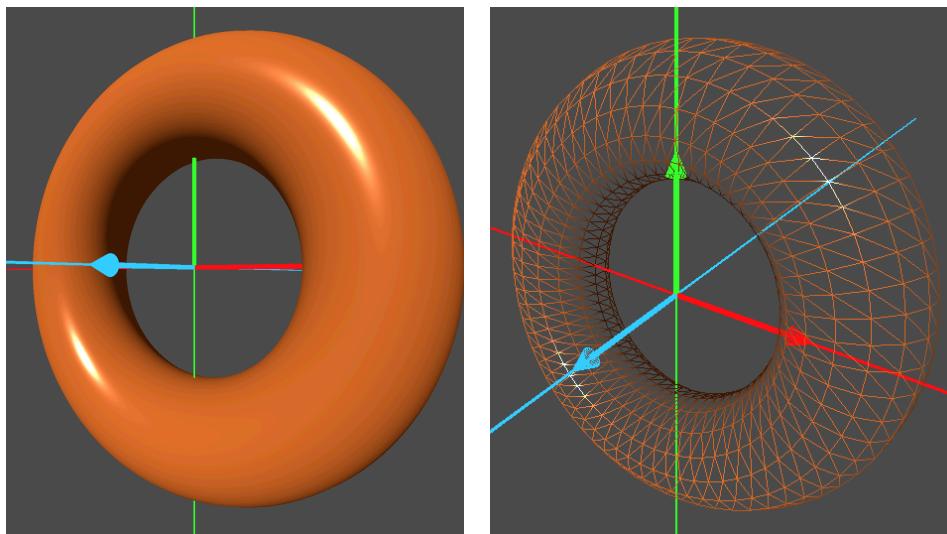
1. Añadir modo de visualización en alambre (*wireframe*)
2. Cargar modelos 3D en formato *glb*.
3. Cargar modelos 3D en formato *obj*.
4. Cálculo de normales de objetos suaves y tipos de sombreado de Godot.
5. Normales de objetos con aristas reales (no suaves).
6. Creación de mallas por revolución de un perfil.

#### 2.3.1. Añadir modo de visualización en alambre (*wireframe*)

Cuando se trabaja con algoritmos de generación de mallas, es muy útil poder ver las aristas de los triángulos que forman la malla para depurar los algoritmos. Godot permite activar un modo de visualización en alambre (*wireframe*) que muestra las aristas de todos los objetos 3D de la escena, en lugar de los triángulos llenos. En esta práctica se activará o desactivará al pulsar la tecla W.

A modo de ejemplo, en la figura 16 se muestra un modelo 3D (el donut que se menciona más adelante en este guión) en modo normal (izquierda) y en modo *wireframe* (derecha).

Da estos pasos para incorporar esta funcionalidad al proyecto:



**Figura 16 :** Donut en modo normal (izquierda) y en modo *wireframe* (derecha).

1. Añade un script al nodo raíz de la escena, al añadirlo usa el archivo `script_raiz.gd` que tienes en los materiales de prácticas. Ese script ya tiene el código necesario para activar o desactivar el modo *wireframe* al pulsar la tecla W. El código se ve en la figura 17.

### 2.3.2. Cargar modelos 3D en formato *glb*

El formato **glb** permite almacenar escenas completas con diferentes componentes, incluyendo sus texturas. Este formato es la versión en binario del formato 3D **glTF** (cuyos archivos son JSON). Godot permite importar archivos *glb* de forma simple, y permite editar individualmente sus partes tras usar la opción para *convertir a escena editable*.

```
extends Node3D

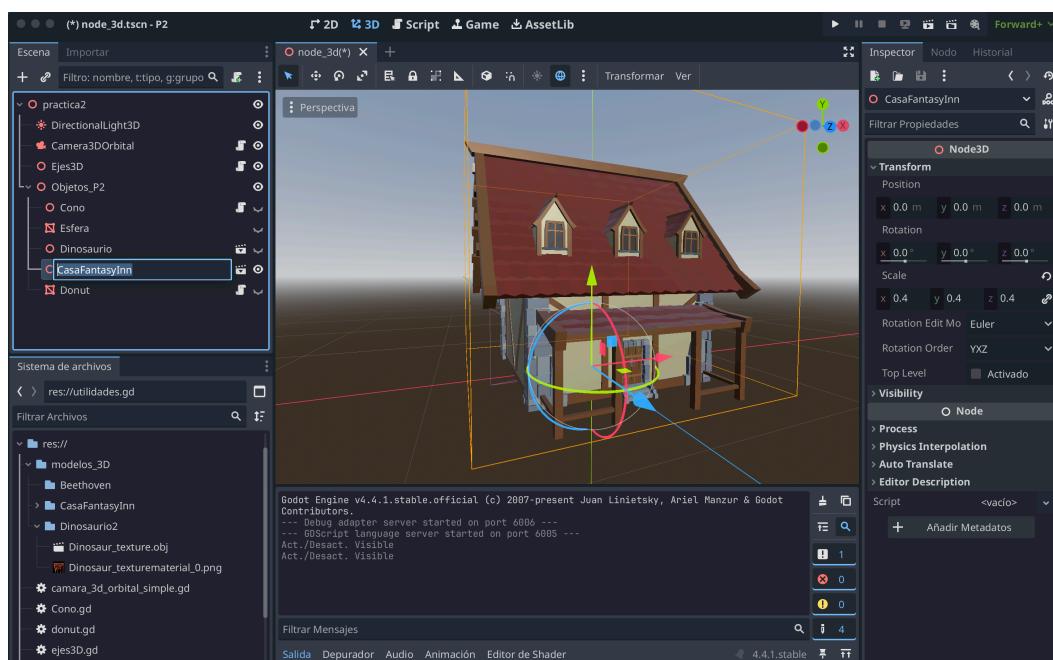
var dibujar_aristas : bool = false

func _init():
    RenderingServer.set_debug_generate_wireframes(true)

func _unhandled_key_input( key_event ):
    if key_event.keycode == KEY_W and not key_event.pressed :
        dibujar_aristas = not dibujar_aristas
        var viewport = get_viewport()
        if dibujar_aristas:
            viewport.debug_draw = Viewport.DEBUG_DRAW_WIREFRAME
            print("Dibujar en modo aristas: activado")
        else:
            viewport.debug_draw = Viewport.DEBUG_DRAW_DISABLED
            print("Dibujar en modo aristas: desactivado")
```

**Figura 17 :** Código para poder activar el modo *wireframe* con una tecla

1. Descargar al menos un modelo en formato *glb*. Puedes hacerlo desde cualquier repositorio abierto como:
  - Sketchfab: <https://sketchfab.com>,
  - Kenney.nl: <https://kenney.nl/assets> o
  - Poly Pizza: <https://poly.pizza>.
2. Para importar el modelo en Godot, cópialo al directorio del proyecto. El modelo aparecerá en el panel *Sistema de archivos* (abajo a la derecha). Para que la carpeta del proyecto no aparezca con muchos archivos de modelos diferentes, crea una subcarpeta de ella llamada **modelos\_3D**, dentro de esa carpeta, crea una subcarpeta distinta para cada modelo que descargues y ponle un nombre descriptivo del mismo. En cada subcarpeta de cada modelo aparecerán varios archivos relacionados con dicho modelo, los que descargues y los que crea Godot al importarlo.
3. Para organizar mejor el árbol de escena, crea un nodo de tipo **Node3D** como hijo del nodo raíz, y renómbralo como **ObjetosP2**. Este nodo servirá para tener como hijos todos los objetos de esta práctica y así distinguirlos fácilmente de los ejes, la cámara y la fuente de luz. A medida que vayas añadiendo objetos debajo de ese nodo, los verás todos ellos juntos en la escena. Para poder apreciar los objetos por separado, puedes poner como invisibles todos ellos menos uno, que será el que quieras ver en cada momento. Para cambiar la visibilidad de un nodo, haz click en el ícono con forma de ojo que hay a la derecha del nombre del nodo en el panel del árbol de escena.
4. Para incorporar el modelo *glb* arrástralolo desde el panel *Sistema de archivos* hasta el nodo **ObjetosP2**. Después verifica que la escala del modelo es apropiada (no aparece muy grande o muy pequeño en relación a los ejes), y si es necesario, edita su transformación y añádele un escalado adecuado.



**Figura 18 : Ejemplo de carga de modelo con formato *glb***

### 2.3.3. Cargar modelos 3D en formato *obj*

En esta actividad podemos probar a añadir a la escena uno o varios modelos en formato *obj*, otro formato para modelos 3D muy extendido.

1. Descargar al menos un modelo en formato *obj*.
2. Descomprímelo y muevelo a su propia subcarpeta dentro de la carpeta `modelos_3D` (ponle a la carpeta un nombre descriptivo). Debe haber un archivo `.obj`, que contiene la geometría, un archivo `.mtl`, de definición de materiales, y una o varias texturas.
3. Crea un nodo `MeshInstance3D` en la escena, como nodo hijo de `ObjetosP2` y ponle un nombre descriptivo del modelo que has descargado.
4. Arrastra el archivo `.obj` desde el panel *Sistema de archivos* sobre el nuevo nodo.

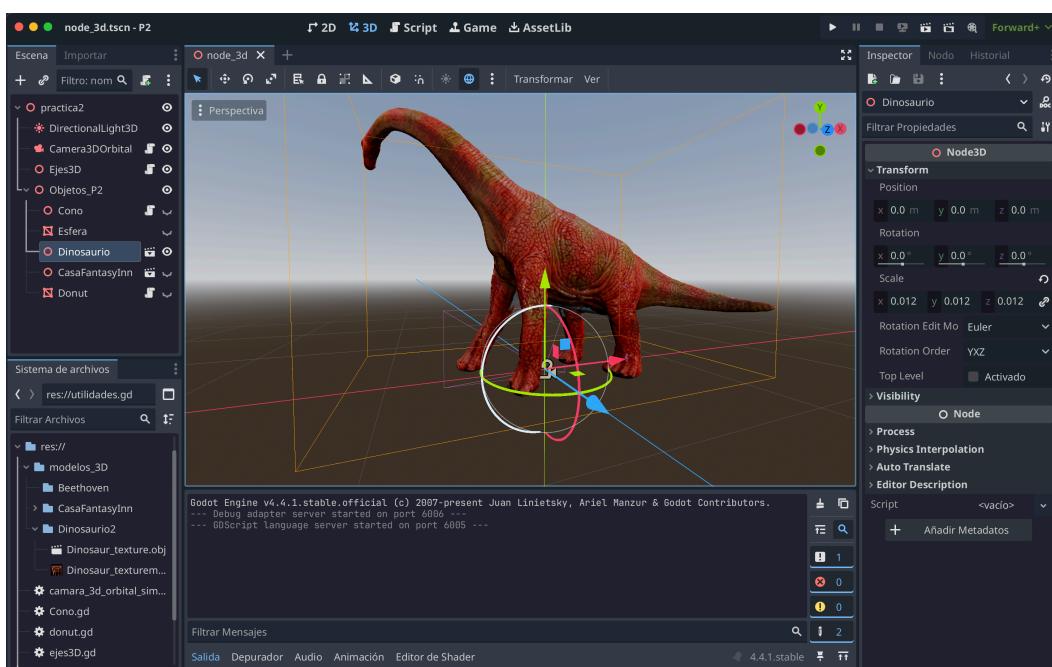


Figura 19 : Ejemplo de carga de modelo con formato *obj*

### 2.3.4. Cálculo de normales de objetos suaves y tipos de sombreado de Godot.

En esta actividad veremos el uso de un algoritmo para calcular las normales de los vértices de una malla indexada de triángulos que aproxima una superficie suave (es decir, sin discontinuidades en la normal). En este tipo de mallas muchas veces se dispone de las coordenadas de los vértices, pero no de las normales, que son necesarias la iluminación.

Para calcular las normales, y puesto que sabemos que la malla aproxima una superficie suave, podemos asignar a cada vértice la normal promedio (normalizada) de las caras adyacentes a dicho vértice. El algoritmo que hace eso se encuentra implementado en la función `calcNormales` que tienes disponible en el archivo `utilidades.gd` (puedes ver el código en la figura 21). A modo de ejemplo, cuando se ejecuta el algoritmo para un objeto con forma de donut, las normales que produce se pueden observar en la figura 22.

```

extends MeshInstance3D

func _ready() -> void:

## Crear las tablas de vértices y triángulos de un Donut
var vertices := PackedVector3Array([])
var triangulos := PackedInt32Array([])
Utilidades.generarDonut( vertices, triangulos )

var normales := Utilidades.calcNormales( vertices, triangulos )

## inicializar el array con las tablas
var tablas : Array = [] ## tabla vacía incialmente
tablas.resize( Mesh.ARRAY_MAX ) ## redimensionar al tamaño adecuado
tablas[ Mesh.ARRAY_VERTEX ] = vertices
tablas[ Mesh.ARRAY_INDEX ] = triangulos
tablas[ Mesh.ARRAY_NORMAL ] = normales

mesh = ArrayMesh.new() ## crea malla en modo diferido, vacía
mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, tablas )

## crear un material
var mat := StandardMaterial3D.new()
mat.albedo_color = Color( 1.0, 0.5, 0.2 )
mat.metallic = 0.3
mat.roughness = 0.2
mat.shading_mode = BaseMaterial3D.SHADING_MODE_PER_PIXEL

material_override = mat

```

**Figura 20 :** Código del *script* para el donut, en el archivo **donut.gd**

Además de los anterior, también veremos el efecto que tiene el **tipo de sombreado** en gráficos, término que se refiere a la forma de hacer el cálculo de la iluminación de una escena. Godot incorpora dos posibilidades:

- Calcular la iluminación en cada pixel donde se proyecta cada triángulo (usando la normal interpolada de los vértices). Se denomina **sombreado por pixel**.
- Calcular la iluminación en cada vértice de la malla, y luego interpolar el color resultante en cada pixel de cada triángulo. Se denomina **sombreado por vértice**. Es más rápido en comparación con el anterior (para mallas que no tengan muchísimos vértices), pero produce resultados menos realistas.

Para ilustrar este algoritmo, usaremos un objeto con forma de toroide (un donut), el cual se puede generar con un script sin las normales. Una vez generado probaremos a calcular las normales con el citado algoritmo, y luego probaremos también a cambiar el tipo de sombreado.

Las actividades a realizar son las siguientes:

```

func calcNormales( verts : PackedVector3Array,
                   tris   : PackedInt32Array ) -> PackedVector3Array :

    var nv : int = verts.size() ## número de vértices
    var nt : int = tris.size()/3 ## número de triángulos

    ## Inicializa normales a cero
    var normales := PackedVector3Array([])
    for iv in nv:
        normales.append( Vector3.ZERO )

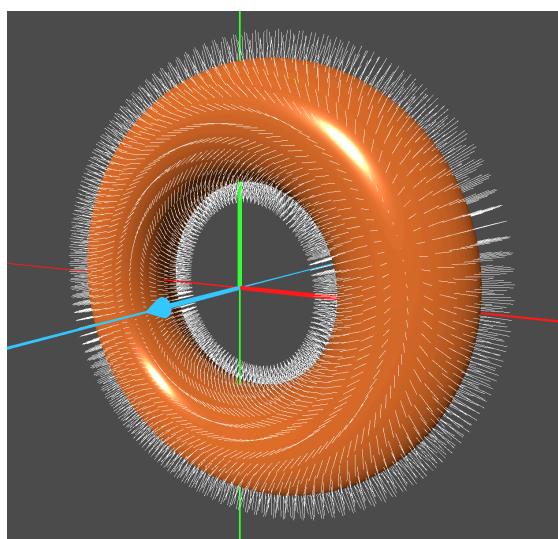
    ## Para cada triángulo, calcular su normal y sumarla
    ## a las normales de sus tres vértices.
    for it in nt :
        var t := Vector3i( tris[3*it+0], tris[3*it+1], tris[3*it+2] )
        var a := verts[t[0]] ;
        var b := verts[t[1]] ;
        var c := verts[t[2]] ;
        var normal_tri := (c-a).cross(b-a).normalized()
        for ivt in 3 :
            normales[t[ivt]] += normal_tri

    # Normalizar todos los vectores normales
    for iv in nv:
        normales[iv] = normales[iv].normalized()

    # Hecho
    return normales

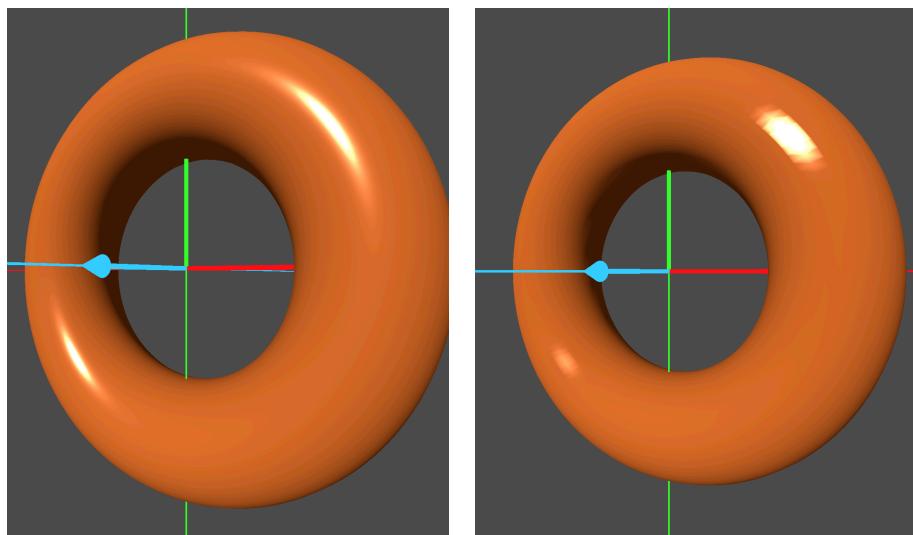
```

**Figura 21 :** Función que calcula las normales (en `utilidades.gd`)



**Figura 22 :** Donut con las normales calculadas por el algoritmo.

1. Crear un archivo global de scripts con funciones que podrás llamar desde cualquier otro script. Lo puedes hacer desde el menú de Godot, en *Proyecto* → *Configuración del proyecto* → *Globales*. Ahí añade un nuevo objeto *autoload*, con nombre **Utilidades** y con el archivo asociado **utilidades.gd** (que tienes disponible en el material de la asignatura). Ese archivo tiene, entre otras cosas, el código de la función **calcNormales**, que implementa el citado algoritmo de calcular normales, y que puedes ver en la figura 21.
2. Crear un nodo hijo de tipo **MeshInstance3D**. Renómbralo como **Donut**.
3. Asóciale un script con las declaraciones y la función **\_ready** que vemos en la figura 20 (está disponible en el archivo **donut.gd** que hay en los materiales de la asignatura). Compara este código con el de la práctica 1: ahora estamos definiendo un array de vértices (**vertices**) y un array de caras (**triangulos**), para poder calcular las normales en los vértices (vector **normales**). Ahí se fija el tipo de sombreado a *sombreado por pixel*.
4. Ejecuta. El donut aparece como se observa en la figura 23 (izquierda). Puedes observar que el material produce unos brillos especulares, que se ven correctos.
5. Cambia el modo de sombreado en el script del donut, usa sombreado por vértice en lugar de por pixel. Para ello modifica en el script el correspondiente atributo del material. El donut tendrá ahora el aspecto de la figura 23 (derecha). Puedes ver como ahora no se reproducen bien esos brillos. Razona a qué se debe esto.



**Figura 23 :** Donut con sombreado por pixel (izquierda) y por vértice (derecha).

### 2.3.5. Normales de objetos con aristas reales (no suaves).

En algunos casos queremos asignar normales a objetos que no son *suaves*, en el sentido de que tienen aristas reales, un ejemplo simple es un cubo real, o cualquier paralelepípedo real. Si usamos para estos objetos el algoritmo de cálculo de normales, obtendremos resultados inesperados, ya que el algoritmo asume que las aristas son suaves y no tiene en cuenta las discontinuidades de la normal en las aristas.

Para usar el algoritmo en estos objetos, será necesario modelar las aristas reales con aristas del modelo, pero en estos casos se hace necesario replicar los vértices, de forma que tengamos más de un vértice en una misma posición (en los extremos de algunas aristas), pero cada réplica del vértice tiene una normal distinta y es adyacente únicamente a triángulos con dicha normal.

Para el ejemplo del cubo, en principio podemos pensar que se puede modelar con un modelo de 8 vértices (correspondientes a las 8 esquinas reales de un cubo de 6 caras) y 12 triángulos (2 por cara), pero entonces el algoritmo de cálculo de normales no funcionará bien, como se ha descrito. Para que el **mismo algoritmo de generación de normales** funcione bien, necesitaremos un modelo con 12 triángulos (igual que antes) pero con 24 vértices, en concreto con 3 vértices en cada esquina real del cubo. Cada uno de esos tres vértices en la misma esquina (con la misma coordenada) tendrá asignada una normal perpendicular a una de las 3 caras del cubo que confluyen en dicha esquina (y será adyacente uno o dos triángulos coplanares en dicha cara).

Lo anterior se ilustra en la figura 24, donde se muestra un cubo con sus normales y sombreado. A la izquierda vemos el cubo de 8 vértices y a la derecha el de 24 vértices. En el de 8 la iluminación es incorrecta, ya que las normales se han generado en las direcciones de las 8 diagonales, lo cual no representa bien al objeto real. En el cubo de 24 vértices, sin embargo, vemos que en cada esquina del cubo hay 3 normales, cada una perpendicular a una de las 6 caras de dicho cubo.

Las actividades a realizar son las siguientes:

1. Crea un nodo hijo de `ObjetosP2`, de tipo `MeshInstance3D`. Renómbralo como `Cubo8vertices`.
2. Añade un script de forma que en el método `_ready` se definan los arrays de vértices y triángulos de un cubo de 8 vértices, se calculen las normales con la función que ya dispones, y se cree la malla y su material. Usa sombreado por pixel. Puedes basarte en el script del donut, pero definiendo los arrays de vértices y triángulos del cubo.

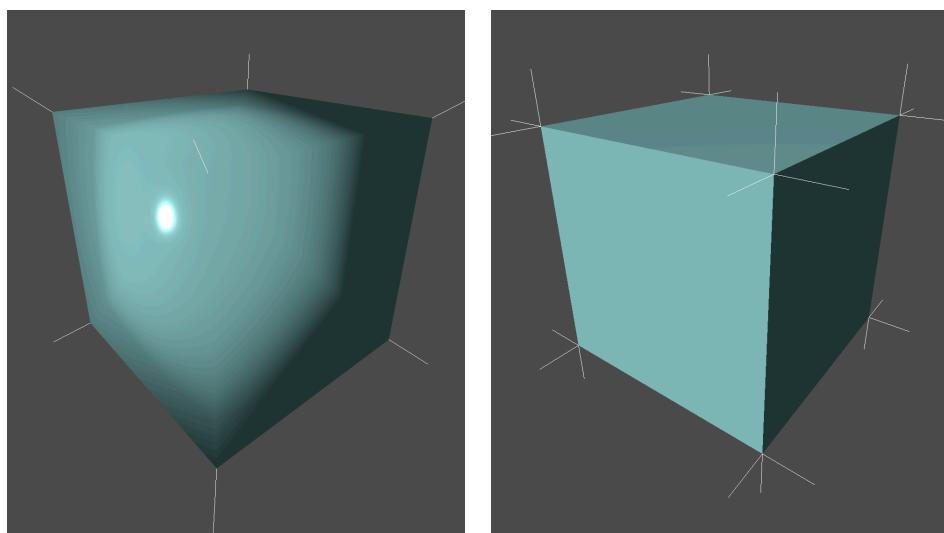


Figura 24 : Cubo de 8 vértices (izquierda) y de 24 vértices (derecha).

3. Crea un segundo nodo hijo de `ObjetosP2`, de tipo `MeshInstance3D`. Renómbralo como `Cubo24vertices`.
4. Añade un script de forma que en el método `_ready` se definan los arrays de vértices y triángulos de un cubo de 24 vértices, e igualmente se calculen las normales con la función que ya dispones, y se cree la malla y el material.
5. Compara los resultados de ambos objetos.

### 2.3.6. Creación de mallas por revolución de un perfil.

El objetivo de esta actividad es ejercitarse las posibilidades de generación procedural de geometría 3D. Para ello se pide implementar un algoritmo el cual, a partir de un perfil 2D (es un array de vectores de 2 componentes, tipo `Vector2`, que representa una secuencia de puntos en el plano  $Z=0$ ).

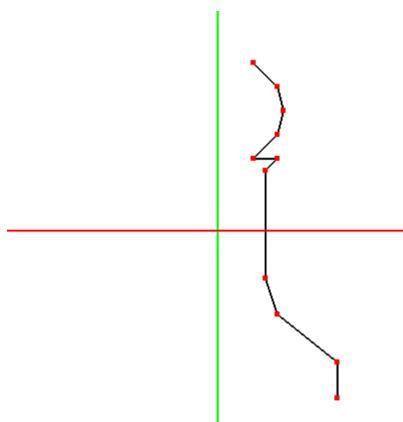
En la figura 25 se muestra un ejemplo de perfil que, al girar alrededor del eje Y, genera un modelo 3D con forma de peón de ajedrez. La coordenada X de los vértices (eje rojo) debe ser siempre positiva. Suponemos que un vértice del perfil tiene como coordenadas  $(x, y)$ , esas coordenadas se interpretan como un punto del plano  $z = 0$ , con coordenadas  $(x, y, 0)$ . Después se rotarán un ángulo  $\theta$  alrededor del eje Y, produciendo unas coordenadas  $(x', y', z)$ , donde ya  $z \neq 0$ .

El perfil se puede crear explícitamente dando la secuencia de puntos del mismo, o bien proceduralmente, mediante algoritmos que generan el array con dicha secuencia.

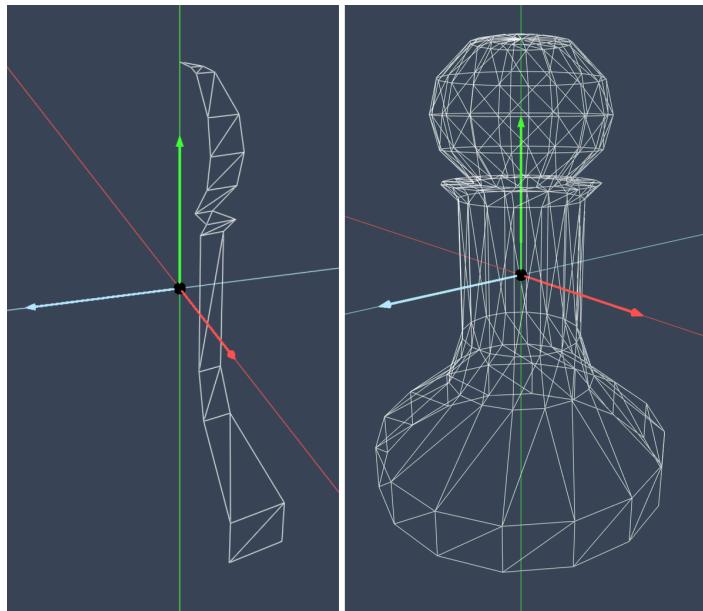
En la figura 26 se muestran (a la izquierda) las aristas de las dos primeras copias del perfil ya en 3D, y (a la derecha) todas las aristas del modelo 3D.

Las actividades a realizar son las siguientes:

1. Implementa una función global que genere la malla por revolución alrededor del eje Y. La puedes situar en un archivo con definiciones globales que añades al proyecto, igual que has hecho con `utilidades.gd`, pero con otro script. La función aceptará como parámetros:



**Figura 25 :** Ejemplo de una perfil que por revolución produce un modelo de un peón de ajedrez.



**Figura 26 : Aristas del objeto de revolución de un perfil.**

- El perfil 2D de entrada (un array de `Vector2`, de tipo `Array[Vector2]` o bien `PackedVector2Array`).
  - El número número de copias del perfil que se generan al dar una vuelta completa.
  - Dos arrays de salida (vacíos al llamar) que se llenarán con los vértices y triángulos resultantes. Son de tipos `PackedVector3Array` y `PackedInt32Array`, respectivamente.
2. Crea un nodo hijo de `ObjetosP2`, de tipo `MeshInstance3D`. Renómbralo con un nombre descriptivo.
  3. Añade un script a ese objeto. En el método `_ready`:
    1. Define un perfil 2D como lista de puntos en el plano X-Y.
    2. Invoca la función que has implementado para generar la malla por revolución.
    3. Invoca la función de cálculo de normales para obtener las normales de los vértices. Alternativamente, puedes pensar como calcular las normales de los vértices del perfil original y luego como *rotarlas* para el resto de los vértices en las copias del perfil.
    4. Crea el objeto `mesh` y su material, de forma semejante al script del donut.
  4. Ejecuta y observa el resultado.

Experimenta con diferentes perfiles: un cilindro, un cono, una esfera, un vaso, una botella, etc... Compara el aspecto usando sombreado por pixel y por vértice.

## 2.4. Entrega de la práctica

- Subir la carpeta del proyecto `godot` con todos los archivos necesarios comprimido en un zip. El proyecto debe incluir todos los pasos realizados en la práctica.
- Incluir en la entrega un breve documento explicando la generación de objetos de revolución y como se le han asignado las normales.

## Práctica 3.

# Grafos de escena.

### 3.1. Objetivos

Los objetivos de esta práctica son:

- Aprender a diseñar e implementar modelos jerárquicos de objetos articulados.
- Aprender a crear el grafo de escena.
- Aprender el funcionamiento de la pila de transformaciones.
- Aprender a modificar interactivamente parámetros del modelo.
- Aprender a implementar animaciones sencillas.

### 3.2. Requisitos previos y recomendaciones

Los requisitos previos para realizar esta práctica son:

- Haber realizado la práctica 2.
- Disponer de los archivos de scripts y modelos 3D que se han usado en las prácticas anteriores, tanto los proporcionados en el material de prácticas como los que hayas creado tú.

Para crear el proyecto debes de usar el mismo nodo raíz, la misma cámara orbital, la misma fuente de luz, y el objeto con los ejes visibles que ya se han usado en las otras dos prácticas.

Además de esto, respecto del árbol de escena:

- Usa nombres descriptivos para los nodos del árbol, por ejemplo usa `PelotaTenis` en lugar de `MeshInstance5`, eso facilita buscar objetos en el árbol.
- Recuerda usar la misma estructura del árbol de escena usado en la práctica 2, es decir, el nodo raíz de la escena principal debe tener como hijos:
  - ▶ Un nodo con la cámara orbital.
  - ▶ Un nodo con una fuente de luz (o varias).
  - ▶ Un nodo con el objeto que contiene los ejes visibles.
  - ▶ Un nodo padre de todos los objetos que forman el modelo articulado, lo puedes llamar `ObjetosP3` o `GrafoP3`.

Respecto a los archivos del proyecto, se recomienda organizar los archivos en sub-carpetas dentro de la misma carpeta del proyecto (la carpeta que contiene el archivo `.project`) para que sea fácil buscar y evitar colisiones, en concreto:

- Como se indicó en el guión de la práctica 2, usa un carpeta `modelos_3D` para los modelos 3D que hayas descargado. Dentro de esa carpeta, usa una sub-carpeta para cada modelo distinto, y ponle nombres descriptivos a esas sub-carpetas. Dentro de la

subcarpeta de cada modelo, sitúa todos los archivos relacionados con el mismo, tanto archivos de modelo, como archivos de materiales, imágenes de texturas, etc...

- Usa una carpeta llamada `scripts` para situar en ella todos los scripts (archivos `.gd`). Dentro de esa carpeta, usa una subcarpeta de nombre `nodos` para los scripts asociados a los nodos del grafo, y otra llamada `globales` para los scripts globales, creados como objetos *autoload*, como por ejemplo es el script `utilidades.gd` que ya has usado.

### 3.3. Actividades

En las siguientes subsecciones se describen las actividades que debes realizar para completar la práctica, son las siguientes:

1. Diseñar el grafo de escena.
2. Implementar el modelo en Godot.
3. Generar una animación del modelo.
4. Activar y desactivar la animación.

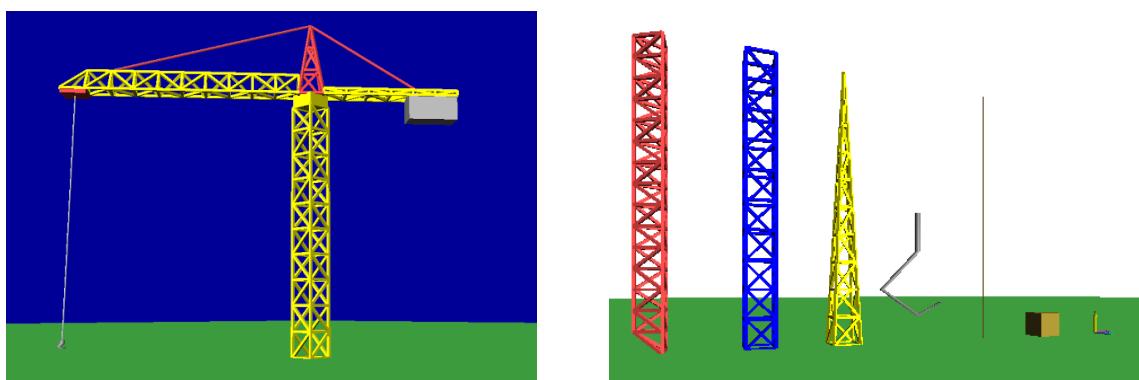
#### 3.3.1. Diseñar el grafo de escena

En esta práctica se debe diseñar el modelo de un objeto articulado con al menos tres articulaciones (que se deben controlar con giros y desplazamientos). Al menos dos de las articulaciones deben ser dependientes. Se puede tomar como ejemplo el diseño de una grúa semejante a la de la figura 27, que tiene tres grados de libertad: ángulo de giro del brazo, desplazamiento del gancho en el brazo y altura del gancho. Se pueden ver otros tipos de objetos en la figura 28.

1. Elije el objeto que vas a modelar.

El modelo puede incluir como nodos terminales:

- Objetos predefinidos de Godot.
- Objetos creados en la práctica 1.
- Modelos importados siguiendo el procedimiento visto en la práctica 2.



**Figura 27 :** Ejemplo de modelo jerárquico (izquierda), y los objetos usados (derecha).



**Figura 28 : Ejemplos de diversos objetos jerárquicos.**

- Objetos de revolución creados con el código de la práctica 2, y otros modelos procedurales que construyeras en la práctica.

## 2. Diseña el grafo de escena.

El diseño del modelo se debe materializar en un grafo de escena que se entregará en un archivo PDF.

El grafo debe incluir todas las transformaciones, indicaciones sobre los parámetros o grados de libertad, y referencias a los objetos usados como nodos terminales.

### 3.3.2. Crear el modelo en Godot

1. Implementa el modelo en Godot siguiendo el grafo de escena diseñado.
2. Comprueba que las articulaciones están definidas correctamente modificando las transformaciones geométricas en el editor.

### 3.3.3. Generar una animación del modelo

Para animar el modelo añade un script a los nodos que contienen las transformaciones geométricas asociadas a los grados de libertad que haga que en cada frame se modifique la transformación.

Por ejemplo para modificar el ángulo de la rotación respecto a Y se puede usar:

```
extends Node3D

@export var rotation_speed_deg := 10.0 # grados por segundo

func _process(delta):
    # Rotación continua en Y
    rotation.y += deg_to_rad(rotation_speed_deg * delta)
```

Incluye script de animación para todos los grados de libertad.

### 3.3.4. Activar y desactivar la animación

Modifica los script para poder activar y desactivar la animación de cada articulación.

```
extends Node3D

@export var activar := "activar_cabeza"
var activa := true
var angulo := 0.0

func _process(delta):
    if Input.is_action_just_pressed(activar):
        activa = !activa
    if activa:
        rotation.y += deg_to_rad(rotation_speed_deg * delta)
```

En *Proyecto → Configuración del proyecto → Mapa de entrada*, define las acciones usadas. En este ejemplo **activar\_cabeza**.

Usa las teclas numéricas 1,..,9 para activar los movimientos.

## 3.4. Entrega de la práctica

- Subir la carpeta del proyecto **godot** con todos los archivos comprimidos en un zip.
- Incluir en la entrega un breve documento con el grafo de escena y explicando los scripts de interacción.

## Práctica 4. Iluminación, materiales y texturas.

### 4.1. Objetivos

- Comprender el modelo de iluminación local.
- Aplicar materiales con distintas propiedades visuales.
- Incorporar y ajustar texturas sobre superficies.
- Añadir y configurar múltiples fuentes de luz en una escena 3D.

### 4.2. Requisitos previos

- Haber realizado la práctica 3.
- Crear un proyecto nuevo, creando el nodo raíz, una fuente de luz y la cámara orbital creada en la práctica anterior.
- Disponer de los mismos archivos `.gd` descritos en los requisitos de las prácticas 2 y 3.
- Disponer de los archivos `script_raiz.gd`, `utilidades.gd` y `donut.gd` que se necesitan para la práctica y se mencionan en las actividades.

### 4.3. Actividades

En las siguientes sub-secciones se detallan las actividades de la prácticas, que son las siguientes:

1. Creación de la escena
2. Iluminación
3. Materiales
4. Texturas

#### 4.3.1. Creación de la escena

En este paso se creará una escena con 9 copias de un objeto de revolución.

1. Usa una copia del proyecto de la práctica 2.
2. Crea una nueva escena 3D, con un nodo de tipo `Node3D` como raíz.
3. Añade el objeto de revolución creado en la práctica 2.
4. Copia el objeto para generar una distribución de 3 x 3 copias sobre el plano.
5. Asegúrate de que la cámara enfoque al conjunto y añádele un `script` para moverla como cámara orbital.
6. Añade un suelo por debajo del conjunto.

La escena aparecerá como se ilustra en la figura 29, al no haber fuentes de luz, los objetos se ven de un color gris oscuro plano.



Figura 29 : Escena de partida

#### 4.3.2. Iluminación

En este paso se añadirán y configurarán varias fuentes de luz en la escena.

1. Añade tres luces:

- Una luz `DirectionalLight3D` como fuente principal iluminando desde una esquina.
- Una luz `OmniLight3D` colocada sobre el conjunto.
- Una luz `SpotLight3D` sobre el objeto de otra esquina.

Las luces deben quedar situadas aproximadamente como se muestra en la figura 30.

2. Ejecuta y comprueba el efecto de apagar y encender cada fuente de luz. Puedes hacerlo en el panel de la escena mientras la aplicación se ejecuta.
3. Modifica las intensidad de las fuentes mientras ejecutas la aplicación. Busca un equilibrio entre las luces para que se aprecie el efecto de todas sin que aparezcan zonas veladas.
4. Haz una captura de pantalla de la escena (para incluirla en tu documentación).
5. Configura las fuentes para que se muestren las sombras arrojadas.

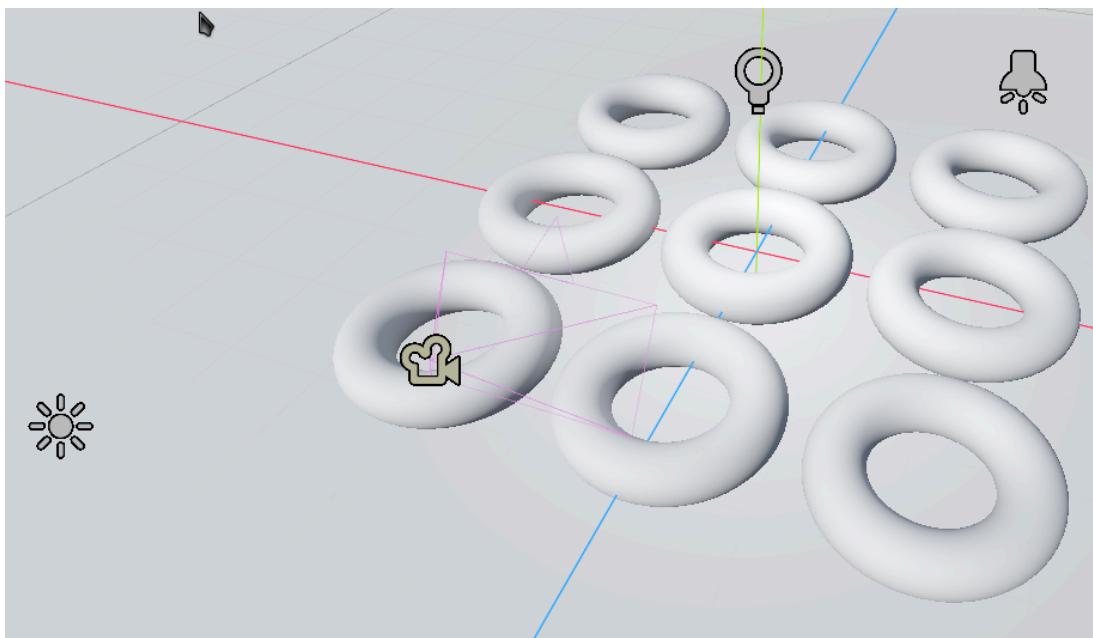
#### 4.3.3. Materiales

En este paso se le asignarán materiales distintos a los objetos de la escena.

1. Asóciale materiales (de tipo `StandardMaterial3D`) a cada uno de los objetos y al suelo.

Puedes crearlos para cada uno de esos objetos en sus propiedades. También puedes crear materiales en el panel inspector. Estos materiales se salvan y se pueden reutilizar, simplemente arrastrándolos sobre los objetos. Para poder modificar algún parámetro del material hay que hacer copia del material para el objeto (se debe usar la opción *Convertir en Único*).

2. Haz que cada fila de objetos tenga el mismo color (es la propiedad *Albedo Color*)



**Figura 30 : Luces**

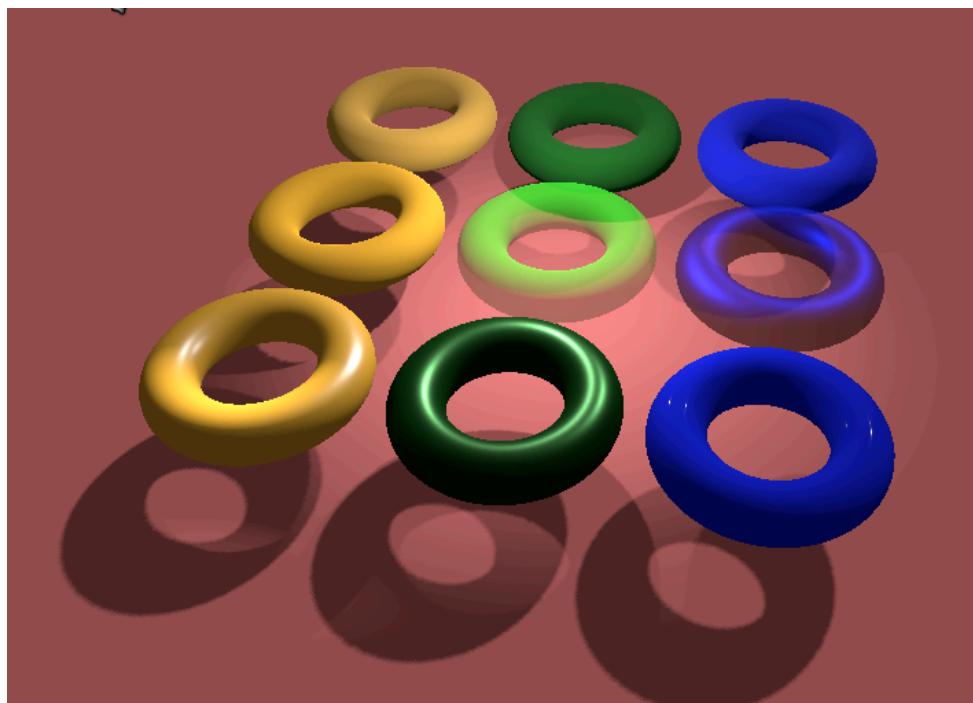
3. Modifica el aspecto de los materiales de los objetos en las columnas cambiando sus propiedades *Metallic* y *Roughness*.
4. Modifica la transparencia de los materiales, para ello usa las propiedades de transparencia, en concreto debes editar el *modo de transparencia* de cada material (propiedad de nombre *Transparency*), por defecto está en modo deshabilitado (valor *Disabled*), es decir, los objetos son opacos, pero se puede activar fijando su valor a *Alpha*. Esto hace que la transparencia del material sea igual a la componente *A* (componente alpha) del color *Albedo Color*, componente que puedes fijar en el editor a un valor entre 0 (totalmente transparente) y 255 (totalmente opaco). Para ello usa el deslizador etiquetado como *A* en el selector del color correspondiente a *Albedo Color* (debajo de los tres deslizadores *R,G,B* o *H,S,V*). Puedes ver que cuando activas la transparencia de esta forma, el objeto no proyecta sombras.
5. Haz una captura de pantalla de la escena (para incluirla en tu documentación).

En la figura 31 se muestra un ejemplo de cómo puede quedar la escena tras aplicar distintos materiales a los objetos y al suelo.

#### 4.3.4. Texturas

El objetivo de este paso es aplicar imágenes como texturas sobre las superficies de los objetos.

1. Añade una imagen de textura (por ejemplo, ladrillos o madera) al proyecto.
2. Crea dos cuadrados usando el script que aparece en las figuras 33 y 34.
3. En el **StandardMaterial3D** de los objetos, activa la opción *Albedo* → *Texture* y selecciona la imagen.
4. Usa coordenadas de textura diferentes para los dos cuadrados.



**Figura 31 : Materiales**

5. Descarga una mapa de normales y aplícaselo.

Puedes ver un ejemplo de cómo queda la escena con texturas aplicadas en la figura 32.

#### 4.3.5. Textura en el objeto de revolución.

En este paso se pretende añadir texturas al objeto de revolución creado en la práctica 2. Para ello es necesario asignar coordenadas de textura a los vértices del objeto.



**Figura 32 : Texturas**

```

extends Node3D

@export var textura: Texture2D

func _ready():
    crear_plano_uv(3, 1.0, 90,0,Vector3(0,1.5,-4)) # Textura completa
    crear_plano_uv(3, 4,0,90,Vector3(4,1.5,0)) # Textura repe. 4 veces

func crear_plano_uv( size: float, escala_uv: float,
                      rx:float, rz:float, pos : Vector3 ):

    var st = SurfaceTool.new()
    st.begin(Mesh.PRIMITIVE_TRIANGLES) ## comienza a crear la malla

    var normal = Vector3.UP

    # Coordenadas de los vértices (plano en XZ, centrado en 'centro')
    var p0 = Vector3(-size, 0, -size)
    var p1 = Vector3(size, 0, -size)
    var p2 = Vector3(size, 0, size)
    var p3 = Vector3(-size, 0, size)

    # UVs multiplicados por escala para repetir textura
    var uv0 = Vector2(0, 0)
    var uv1 = Vector2(escala_uv, 0)
    var uv2 = Vector2(escala_uv, escala_uv)
    var uv3 = Vector2(0, escala_uv)

    # Primer triángulo
    st.set_normal(normal) ; st.set_uv(uv0) ; st.add_vertex(p0)
    st.set_normal(normal) ; st.set_uv(uv1) ; st.add_vertex(p1)
    st.set_normal(normal) ; st.set_uv(uv2) ; st.add_vertex(p2)

```

**Figura 33 :** Código del script para texturas (parte 1)

1. Modifica el script de generación de tu objeto de revolución para que asigne coordenadas de textura a los vértices para ello modifica el código de la función ready añadiendo las instrucciones que aparecen en la figuras 35 y crea una nueva función (en `utilidades.gd`) para calcular las coordenadas de textura siguiendo el esquema de la figura 36.
2. Aplicale texturas a tus objetos.

## 4.4. Entrega de la práctica

- Subir la carpeta del proyecto Godot con todos los archivos comprimidos en un zip.
- Incluir en la entrega un breve documento con las capturas de pantalla realizadas explicando la configuración de cada una y las capturas de pantalla realizadas.

```

# Segundo triángulo
st.set_normal(normal) ; st.set_uv(uv2) ; st.add_vertex(p2)
st.set_normal(normal) ; st.set_uv(uv3) ; st.add_vertex(p3)
st.set_normal(normal) ; st.set_uv(uv0) ; st.add_vertex(p0)

var mesh = st.commit() ## termina de crear la malla
var mi = MeshInstance3D.new()
mi.mesh = mesh

# Crear material con textura
var mat = StandardMaterial3D.new()
mat.albedo_texture = textura
mat.uv1_offset = Vector3.ZERO
mat.uv1_scale = Vector3(1, 1, 1)
mi.material_override = mat
mi.rotation.x = deg_to_rad(rx)
mi.rotation.z = deg_to_rad(rz)
mi.position = pos
add_child(mi)

```

**Figura 34 :** Código del script para texturas (parte 2)

```

var uvs := Utilidades.calcUV(vertices)
tablas[ Mesh.ARRAY_TEX_UV ] = uvs

```

**Figura 35 :** Código a insertar en la función ready)

```

# Utilidades.gd

static func calcUV(vertices: PackedVector3Array) -> PackedVector2Array:
    var uvs := PackedVector2Array()
    var max_u = 1.0
    var max_v = 1.0

    for v in vertices:
        # 1. Calcular el valor del parámetro u
        var phi = atan2(v.z, v.x)
        var u = max_u*((phi / (2*PI)+0.5))

        # 2. Calcular el valor del parámetro v

        #Inserta tu código aquí
        #Puedes calcularlo en función del desplazamiento en el perfil
        # o de forma aproximada en función de y

        var uv_coords = Vector2(u, v)

        uvs.append(uv_coords)

    return uvs

```

**Figura 36 :** Función de calculo de coordenadas de textura

## Práctica 5.

# Interacción con ratón y selección de objetos.

## 5.1. Objetivos

- Comprender cómo detectar e interpretar la interacción del usuario mediante el ratón.
- Aprender a seleccionar objetos en la escena 3D mediante raycasting.
- Visualizar información o cambiar propiedades de objetos seleccionados.
- Leer posiciones 3D.

## 5.2. Requisitos previos

- Haber realizado la práctica 3.
- Copiar el proyecto de la práctica.
- Disponer de los mismos archivos .gd descritos en los requisitos de las prácticas 2 y 3.
- Disponer de los archivos `script_raiz.gd`, `utilidades.gd` y `donut.gd` que se necesitan para la práctica y se mencionan en las actividades.

## 5.3. Actividades

### 5.3.1. Creación de escena base

En primer lugar crearemos una escena sencilla con los objetos visibles de la escena:

- Si tu escena no tiene plano de suelo añádeselo.
- Crea varias primitivas sencillas (por ejemplo cubos, esferas, etc....) colocadas en distintas posiciones sobre el suelo.

A modo de ejemplo, en la figura 37 se muestra una posible escena base para la práctica.

### 5.3.2. Añadir colisionadores

Vamos a añadir un nodo *colisionador* (es decir, un nodo de tipo `CollisionShape3D`) para cada objeto visible que hayas añadido en el paso anterior. Cada nodo colisionador tiene una propiedad **Shape** que tendrá un objeto de alguna clase derivada de `Shape3D`, clase con una forma que se debe corresponder con uno de los objetos visibles. Puedes usar, por ejemplo, las clases `BoxShape3D` o `SphereShape3D`. Los nodos colisionadores no son visibles pero sí son detectables por el algoritmo de intersección rayo-objeto que se usa en la selección.

Para cada uno de los objetos añadidos en la creación de la escena base, debe haber tres nodos asociados:

- Un nodo `MeshInstance3D`, es el nodo con la geometría visible al renderizar la escena.

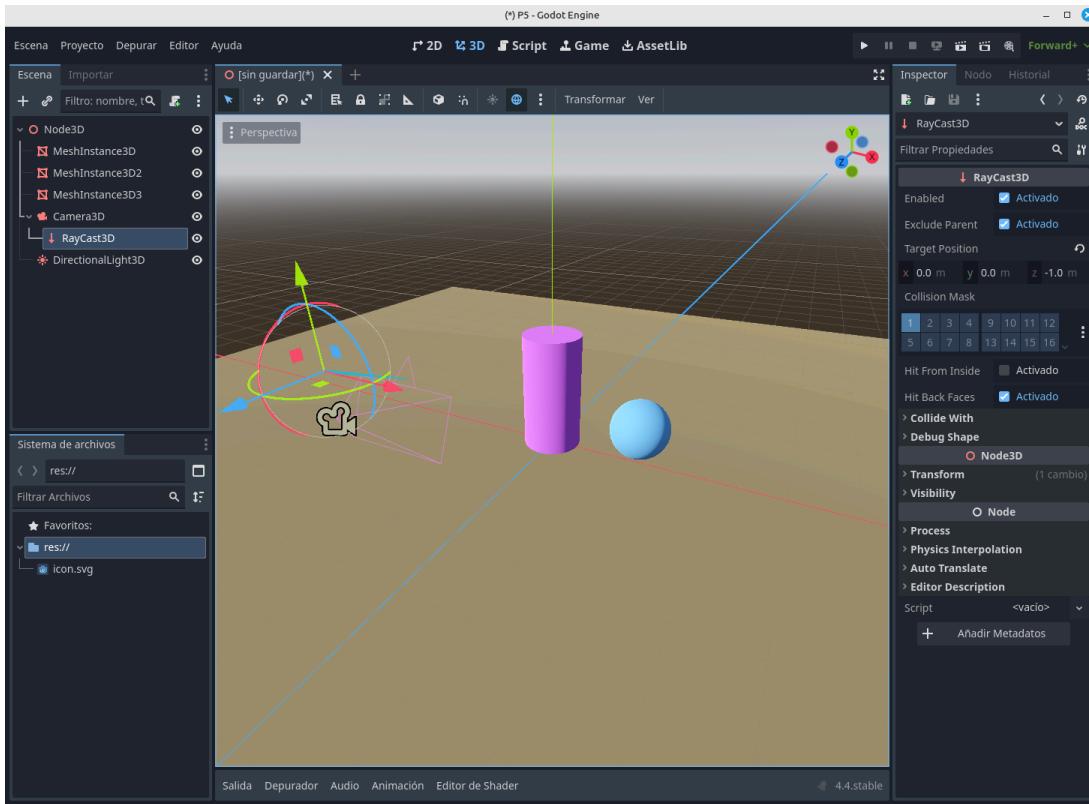


Figura 37 : Escena de partida

- Un nodo `Area3D` o `StaticBody3D`, es el nodo que se llama **nodo colisionador (collider)**,
- Un nodo `CollisionShape3D` que define la geometría del coisionador. Debe ser siempre **hijo del colisionador**

La jerarquía (disposición de estos tres nodos en el árbol de escena) puede ser de alguna de estas dos formas:

- `MeshInstance3D`  $\Rightarrow$  `Area3D` o `StaticBody3D`  $\Rightarrow$  `CollisionShape3D`
- o bien el `StaticBody3D` (o `Area3D`) como padre y los otros dos como hermanos.

Esta jerarquía puede crearse creando los nodos uno a uno o bien creando el nodo `MeshInstance3D`, definir su geometría y después seleccionar el nodo y pulsar en el menú *Malla* (sobre la vista 3D del editor) y seleccionar la opción *crear forma de colisión* con la opción *hijo de cuerpo estático*.

### 5.3.3. Creación de un nodo `RayCast3D`.

Para hacer la selección, una posibilidad es usar el editor y añadir un nodo de tipo `RayCast3D` a nuestra escena:

1. Añade un nodo de la clase `RayCast3D` como **hijo del nodo cámara**.
2. Orienta el rayo hacia el centro de la pantalla, para ello selecciona el nodo, y en el panel de propiedades, actualiza la propiedad *Target position*, y ponle el valor `(0, 0, -1)`, de

```

extends Camera3D

@onready var ray = $RayCast3D # == nodo `RayCast3D` hijo de la cámara

func _input( event ):
    var mouse_pos = get_viewport().get_mouse_position()
    var from = project_ray_origin( mouse_pos )
    var to = from + project_ray_normal( mouse_pos ) * 1000.0
    if event is InputEventMouseButton and event.pressed \
        and event.button_index == MOUSE_BUTTON_LEFT:
        ray.global_position = from
        ray.look_at(to)
        ray.target_position = Vector3(0, 0, -1000) # Apunta en su -Z local
        ray.force_raycast_update()
        if ray.is_colliding():
            var objeto = ray.get.collider()
            print("Seleccionado:", objeto.name)

```

**Figura 38 :** Código de selección con ratón en el nodo cámara

forma que la dirección del rayo siempre se corresponda con la dirección de la cámara (eje Z negativo en el marco de coordenadas local de la cámara).

- Si vas a usar **Areas3D**, activa la colisión con áreas en el inspector. Para ello selecciona el nodo **RayCast3D** y en el panel de la derecha, busca la propiedad *Collide with* y activa la opción **Areas**.

Comprueba que la dirección del rayo es correcta seleccionando la cámara.

#### 5.3.4. Selección de objetos

Para poder seleccionar objetos debemos añadir una función al script de la cámara para detectar los eventos del ratón. Para eso accedemos al nodo **RayCast3D** desde dicha función y lo usamos para detectar colisiones. El código a añadir se puede ver en la figura 38, en este código, si hay un objeto colisionador donde se ha hecho *click*, se imprime su nombre.

El script escribe el nombre del objeto (concretamente el del nodo **Area3D** o **StaticBody3D**), asegúrate de ponerles nombres diferentes y significativos.

#### 5.3.5. Lectura de posiciones con el ratón y creación de objetos

Además de detectar objetos existentes, podemos usar el **RayCast3D** para determinar dónde en el plano (o cualquier otra superficie) ha hecho clic el usuario y **crear nuevos objetos en ese punto**.

Suponiendo que se ha puesto nombre “Suelo” al plano del suelo, podemos crear, por ejemplo, un objeto de tipo **BoxMesh** (u otros) en cada lugar donde se haga click de ese suelo. Sustituye en el script anterior el último **if** por el código de la figura 39.

```

if ray.is_colliding():
    var objeto = ray.get_collider()
    if objeto.name == "Suelo" :
        crear_cubo_en(ray.get_collision_point())
        print("Creado cubo en:", ray.get_collision_point())

```

**Figura 39 :** Creación de un objeto en el punto de click

```

func crear_cubo_en( pos ):

    var nuevo_cubo  = MeshInstance3D.new()
    nuevo_cubo.mesh = BoxMesh.new()

    # Elevarlo para que no atraviese el suelo
    nuevo_cubo.position = pos + Vector3.UP * 0.5

    var mat = StandardMaterial3D.new()
    mat.albedo_color = Color(randf(), randf(), randf())

    nuevo_cubo.material_override = mat

    get_tree().get_current_scene().add_child(nuevo_cubo)

```

**Figura 40 :** Código de la función `crear_cubo_en`

La función `crear_cubo_en` crea un cubo en la posición dada. El código de dicha función se puede observar en la figura 40.

### 5.3.6. Interacción con el modelo jerárquico

Extiende el proyecto de tu modelo jerárquico para poder seleccionar las diferentes piezas del mismo, y permitir que se muevan mientras están seleccionadas.

Puedes usar teclas para cambiar la articulación seleccionada o el desplazamiento del ratón. Redacta un documento explicando como has diseñado la interacción.

## 5.4. Entrega de la práctica

- Subir la carpeta del proyecto Godot con todos los archivos comprimidos en un zip.
- Incluir en la entrega un breve documento PDF explicando el diseño de la interacción usado en el punto 5.