



# UNIVERSIDAD DE GRANADA

**PRÁCTICA 4: Iluminación, materiales y texturas.**

**INFORMÁTICA GRÁFICA**

**Joaquín Cruz Lorenzo**

DNI: 79074896E

Grupo: B3

**Universidad de Granada**

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación  
Departamento de Lenguajes y Sistemas Informáticos

Curso 2025-2026  
Noviembre 2025

# 1. Scripts utilizados

En esta sección se describen los principales scripts implementados para el desarrollo de la práctica, detallando su propósito y las funcionalidades más relevantes. La estructura modular del código ha permitido separar la generación de geometría, las operaciones auxiliares y la gestión de materiales e interactividad.

## 1.1. `peon_revolucion.gd`

Este script es el núcleo de la práctica. Se encarga de generar los peones mediante revolución a partir de un perfil 2D definido en el plano X–Y. Durante la inicialización, el método `_ready()` invoca la función `generar_malla_por_revolucion()` del módulo `AlgoritmoRevolucion`, con la que se calculan los vértices y triángulos de la figura.

Posteriormente se crean las normales con `Utilidades.calcNormales()`, se definen las coordenadas UV y se construye la malla `ArrayMesh`. Además, el script configura materiales con sombreado *per-pixel*, permitiendo alternar entre modo color y modo textura mediante interacción de teclado:

- **T**: alterna entre el color base y la textura correspondiente a cada columna.
- **N**: muestra u oculta las normales de los vértices.

```

extends MeshInstance3D

@export var columna := 0    # 0, 1, 2 → define color y textura
@export var fila := 0       # 0, 1, 2 → define posición

var ver_normales: bool = false
var mostrar_textura := false   # empieza solo con color (sin textura)
var mesh_normales: MeshInstance3D = null
var mat_variante: StandardMaterial3D
var textura: Texture2D
var base_color: Color

func _ready():
    # ... 1. Perfil del peón ...
    var perfil_peon = PackedVector2Array([
        Vector2(0.01, 0.0),
        Vector2(0.8, 0.0),
        Vector2(0.8, 0.2),
        Vector2(0.5, 0.4),
        Vector2(0.6, 1.0),
        Vector2(0.5, 1.2),
        Vector2(0.7, 1.3),
        Vector2(0.01, 1.4),
        Vector2(0.4, 1.6),
        Vector2(0.55, 1.8),
        Vector2(0.55, 2.1),
        Vector2(0.4, 2.3),
        Vector2(0.01, 2.4)
    ])
    # ... 2. Generar revolución ...
    var vertices = PackedVector3Array()
    var triangulos = PackedInt32Array()
    AlgoritmoRevolucion.generar_malla_por_revolucion(perfil_peon, 32, vertices, triangulos)

    # ... 3. Calcular normales y UVs ...
    var normales = Utilidades.calcNormales(vertices, triangulos)
    var uvs = Utilidades.calcUV(vertices)

    # ... 4. Crear malla ...
    var tablas = []
    tablas.resize(Mesh.ARRAY_MAX)
    tablas[Mesh.ARRAY_VERTEX] = vertices
    tablas[Mesh.ARRAY_INDEX] = triangulos
    tablas[Mesh.ARRAY_NORMAL] = normales
    tablas[Mesh.ARRAY_TEX_UV] = uvs

    var array_mesh = ArrayMesh.new()
    array_mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES, tablas)
    self.mesh = array_mesh

    # ... 5. Asignar color y textura por columna ...
    match columna:
        0:
            base_color = Color(0.9, 0.4, 0.4) # rojo
            textura = preload("res://texturas/ladrillos.jpg")
        1:
            base_color = Color(0.8, 0.7, 0.4) # amarillo madera
            textura = preload("res://texturas/madera.jpg")
        2:
            base_color = Color(0.6, 0.6, 0.6) # gris piedra
            textura = preload("res://texturas/piedra.jpg")
        :
            base_color = Color(1, 1, 1)
            textura = preload("res://texturas/madera.jpg")

    # ... 6. Crear material (inicialmente SOLO color) ...
    mat_variante = StandardMaterial3D.new()
    mat_variante.albedo_color = base_color
    mat_variante.albedo_texture = null
    mat_variante.metallic = 0.2
    mat_variante.roughness = 0.4
    mat_variante.specular = 0.6
    mat_variante.shading_mode = BaseMaterial3D.SHADING_MODE_PER_PIXEL
    mat_variante.transparency = BaseMaterial3D.TRANSPARENCY_DISABLED
    mat_variante.cull_mode = BaseMaterial3D.CULL_BACK
    self.material_override = mat_variante

    # ... 8. Normales (opcional) ...
    mesh_normales = AlgoritmosVarios.generarSegmentosNormales(vertices, normales, 0.15, Color(1.0, 1.0, 1.0))
    mesh_normales.visible = ver_normales
    add_child(mesh_normales)

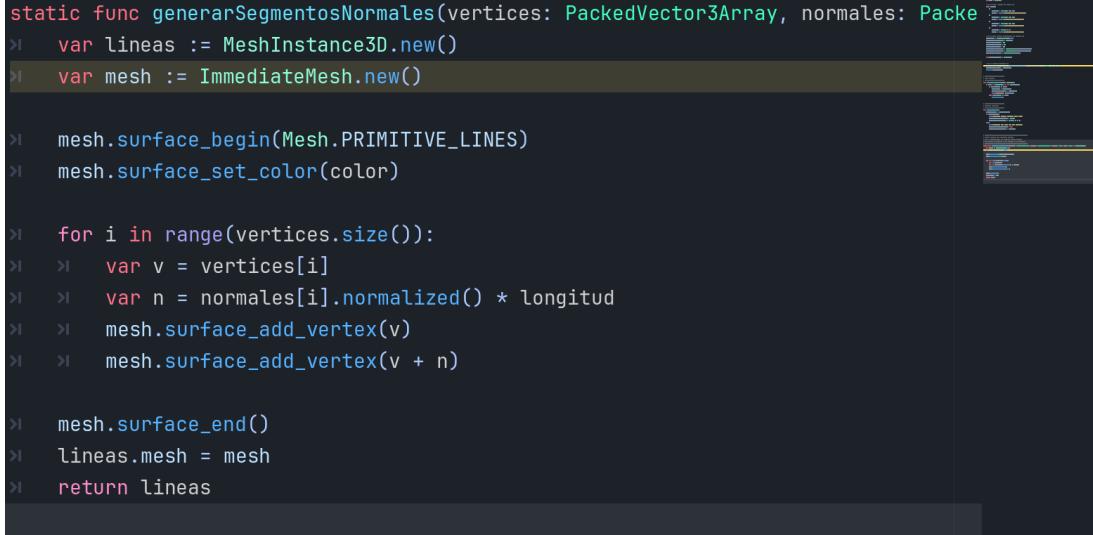
# -----
# INPUT HANDLERS
# -----
func _unhandled_key_input(event: InputEvent):
    if event is InputEventKey and not event.pressed:
        if event.keycode == KEY_N:
            ver_normales = !ver_normales
            mesh_normales.visible = ver_normales
            print("Normales:", ver_normales)
        elif event.keycode == KEY_T:
            _toggle_textura()

# -----
# FUNCIONES AUXILIARES
# -----
func _toggle_textura():
    mostrar_textura = !mostrar_textura

```

## 1.2. algoritmos\_varios.gd

El script `algoritmos_varios.gd` actúa como módulo auxiliar (autoload), proporcionando funciones reutilizables. En esta práctica se utiliza principalmente la función `generarSegmentosNormales` que construye un `MeshInstance3D` con líneas blancas para representar las normales de los vértices. Esto se realiza empleando la clase `ImmediateMesh`, agregando segmentos desde cada vértice en la dirección de su normal.



```
static func generarSegmentosNormales(vertices: PackedVector3Array, normales: PackedVector3Array):
    var lineas := MeshInstance3D.new()
    var mesh := ImmediateMesh.new()

    mesh.surface_begin(Mesh.PRIMITIVE_LINES)
    mesh.surface_set_color(color)

    for i in range(vertices.size()):
        var v = vertices[i]
        var n = normales[i].normalized() * longitud
        mesh.surface_add_vertex(v)
        mesh.surface_add_vertex(v + n)

    mesh.surface_end()
    lineas.mesh = mesh
    return lineas
```

Figura 2: Fragmento del script `algoritmos_varios.gd` mostrando la generación visual de las normales.

## 1.3. algoritmo\_revolucion.gd

Este módulo contiene el procedimiento geométrico que genera la figura 3D por revolución. La función `generar_malla_por_revolucion()` toma un perfil bidimensional (`PackedVector2Array`) y lo rota alrededor del eje Y, generando los anillos de vértices correspondientes. Posteriormente, se construyen los triángulos que conectan los vértices de cada anillo de manera continua, formando la malla completa.

El algoritmo permite además cerrar las tapas superior e inferior si los extremos del perfil están próximos al eje, garantizando un modelo sólido y visualmente coherente.

```

func generar_malla_por_revolucion(perfil: PackedVector2Array, num_copias: int, vertices: PackedVector3Array, triangulos: PackedInt32Array) -> void:
    if perfil.size() < 2 or num_copias < 3:
        print("Error: El perfil necesita al menos 2 puntos y se necesitan al menos 3 copias.")
        return

    # --- 1. Generación de Vértices ---
    var angulo_incremente = (2.0 * PI) / num_copias

    # Bucle para cada copia/sección alrededor del eje Y
    for i in range(num_copias):
        var angulo_actual = i * angulo_incremente
        var transformacion = Transform3D().rotated(Vector3(0, 1, 0), angulo_actual)

        # Bucle para cada punto del perfil original
        for punto2D in perfil:
            # Convertimos el punto 2D (x,y) a un punto 3D (x,y,0) y lo rotamos
            var punto3D = Vector3(punto2D.x, punto2D.y, 0)
            vertices.append(transformacion * punto3D)

    # --- 2. Generación de Triángulos ---
    var num_puntos_perfil = perfil.size()

    # Bucle para cada "cara" entre dos copias consecutivas
    for i in range(num_copias):
        # Bucle para cada segmento del perfil
        for j in range(num_puntos_perfil - 1):
            # Índices de los 4 vértices que forman un "quad"
            var v1 = i * num_puntos_perfil + j
            var v2 = i * num_puntos_perfil + (j + 1)
            # Usamos el módulo (%) para que la última copia conecte con la primera
            var v3 = ((i + 1) % num_copias) * num_puntos_perfil + j
            var v4 = ((i + 1) % num_copias) * num_puntos_perfil + (j + 1)

            # Creamos dos triángulos para formar el quad
            # Triángulo 1: v1, v3, v2
            triangulos.append(v1)
            triangulos.append(v3)
            triangulos.append(v2)
            triangulos.append(v3)

            # Triángulo 2: v3, v4, v2
            triangulos.append(v2)
            triangulos.append(v4)
            triangulos.append(v3)

```

Figura 3: Fragmento del script `algoritmo_revolucion.gd` que genera los vértices y triángulos por revolución.

## 1.4. Resumen de funcionalidades

En la Tabla 1 se resume la función principal de cada script y su papel dentro del proyecto.

Script	Función principal	Tipo de interacción
<code>peon_revolucion.gd</code>	Generación, materiales y gestión de entrada (T/N)	Interactivo
<code>algoritmo_revolucion.gd</code>	Construcción geométrica de la malla por revolución	Automático
<code>algoritmos_varios.gd</code>	Representación visual de normales	Auxiliar

Cuadro 1: Resumen de los scripts utilizados en la práctica.

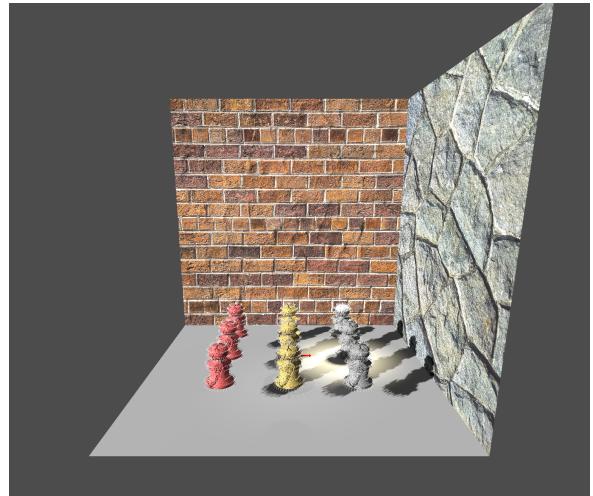
## 2. Resultados y conclusiones

### 2.1. Escenario general

La escena está formada por un conjunto de peones generados mediante revolución a partir de un perfil 2D. Los nueve peones se disponen en una cuadrícula de 3x3, donde cada columna presenta un color base distinto (rojo, amarillo y gris). El entorno incluye un suelo y tres paredes verticales, una de las cuales posee una textura aplicada como fondo decorativo. Sobre estas superficies se proyectan las sombras generadas por las fuentes de luz de la escena.



(a) Vista general del escenario con los peones y las paredes.



(b) Visualización de las normales de los peones.

Figura 4: Configuración general de la escena.

## 2.2. Materiales e interactividad

Cada peón utiliza un material `StandardMaterial3D` con sombreado por píxel (`per-pixel`), ajustado con diferentes valores de rugosidad y metalicidad para conseguir acabados más realistas.

Se implementaron dos modos visuales que pueden alternarse mediante la tecla **T**:

- **Modo color:** muestra únicamente el color base asignado a cada columna.
- **Modo textura:** sustituye el color base por una textura cargada dinámicamente (madera, ladrillos o piedra).

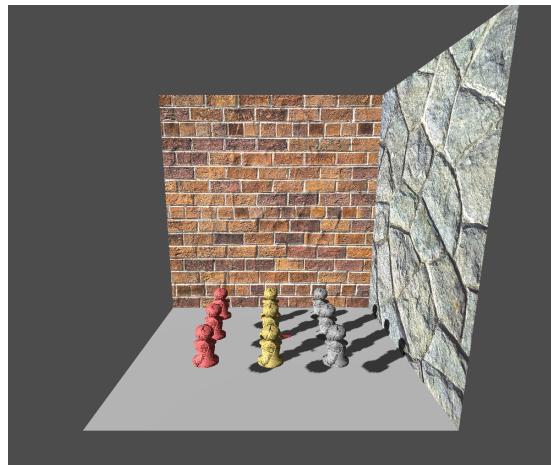
Además, mediante la tecla **N** es posible mostrar u ocultar las normales de la malla, lo que permite comprobar su correcta orientación.

## 2.3. Iluminación

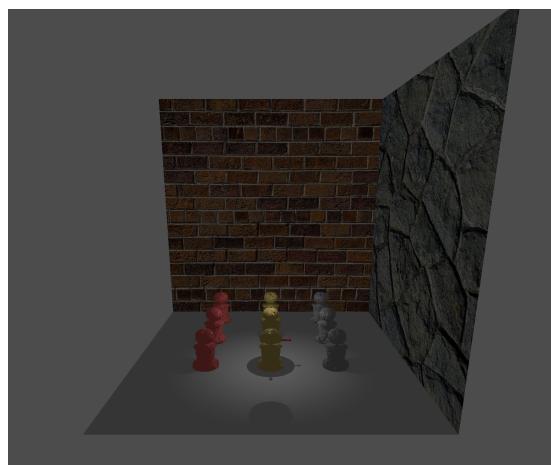
Se han utilizado tres tipos de luces para analizar su comportamiento sobre las mallas y materiales:

- **DirectionalLight3D:** ilumina toda la escena de forma uniforme, simulando la luz solar.
- **OmniLight3D:** emite luz esférica desde un punto central, iluminando los objetos en todas las direcciones.
- **SpotLight3D:** proyecta un haz de luz cónico, permitiendo destacar áreas concretas y generar sombras más marcadas.

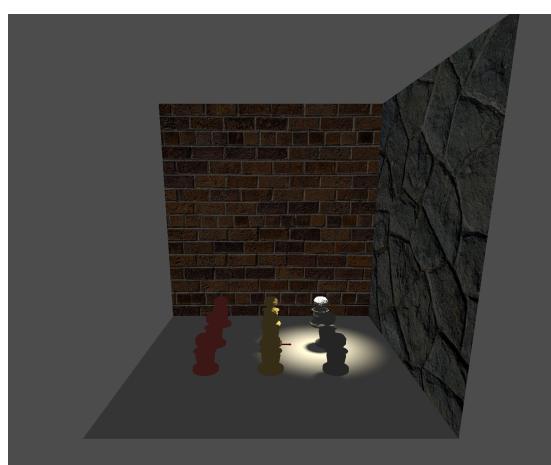
A continuación se muestran las comparativas de cada tipo de luz, tanto en el modo color como en el modo textura:



(a) DirectionalLight3D - Modo color



(b) OmniLight3D - Modo color



(c) SpotLight3D - Modo color

## 2.4. Conclusiones

La práctica ha permitido comprender el proceso de generación de geometría por revolución, la aplicación de materiales avanzados y el efecto de distintas fuentes de luz sobre los objetos 3D. Se ha comprobado que el sombreado *per-pixel* proporciona resultados más suaves y realistas en comparación con el sombreado *per-vertex*, especialmente en modelos curvos como el peón.

Asimismo, la interacción mediante teclado facilita la exploración visual de las propiedades del material, permitiendo alternar entre representaciones basadas en color y textura. Las pruebas con diferentes luces confirman la influencia de la dirección, la intensidad y la atenuación en la proyección de sombras y en la percepción del volumen.