UNIVERSIDAD DE GRANADA

Informática Gráfica

Temario

Ismael Sallami Moreno

Recursos Ingeniería Informática y Ade

Licencia

Informática Gráfica

Ismael Sallami Moreno

Índice general

Parte I

Teoría

Capítulo 1

Introducción de Ingeniería Gráfica

La Informática Gráfica se define como el área de la informática dedicada al procesamiento de información de naturaleza geométrica y visual. Sus campos de estudio más relevantes abarcan la representación de información mediante modelos geométricos, la generación de imágenes o visualización (conocida como *rendering*), la captura de información a través de la interacción y adquisición de modelos, y la computación geométrica, que incluye operaciones y cálculos sobre dichos modelos. Esta disciplina ha experimentado una notable evolución, pasando de ser una tecnología de nicho en los años 90, limitada a aplicaciones de alto rendimiento como la visualización científica o simuladores de vuelo costosos, a convertirse en una tecnología de consumo masivo presente en todos los ordenadores modernos, principalmente a través de videojuegos y entornos virtuales.

1.1 Aplicaciones gráficas interactivas y visualización

Esta sección introduce los conceptos fundamentales de las aplicaciones gráficas interactivas, describe los métodos esenciales de visualización y detalla el proceso del cauce gráfico, sentando las bases para comprender la generación de imágenes sintéticas.

1.1.1 Aplicaciones gráficas interactivas

Un programa gráfico es aquel que almacena un modelo computacional de cierta información y produce como salida una o varias imágenes. Estas imágenes suelen ser de tipo *raster*, compuestas por un arreglo discreto de píxeles, cada uno con un color RGB específico. No obstante, también existen otros formatos como las imágenes vectoriales.

Una **aplicación gráfica interactiva** se caracteriza por un ciclo continuo de retroalimentación entre el usuario y el sistema. Este tipo de aplicación:

- Visualiza una imagen en una ventana gráfica, la cual representa visualmente un modelo de datos.
- Procesa acciones del usuario, denominadas **eventos**, que se traducen en modificaciones del modelo o de los parámetros de visualización.
- Vuelve a visualizar el modelo modificado de forma inmediata, con tiempos de respuesta del orden de decenas de milisegundos, para proporcionar una retroalimentación instantánea al usuario.

Este esquema es fundamental en campos como los videojuegos, simuladores, diseño asistido por computador (CAD), realidad virtual y realidad aumentada.

El núcleo de una aplicación interactiva es el **bucle principal de gestión de eventos**, que se ejecuta de forma continua y consta de tres pasos fundamentales:

1) Esperar a que ocurra un evento y recuperar sus datos.
2) Procesar el evento, lo que generalmente implica actualizar el modelo y/o los parámetros de visualización.
3) Visualizar el modelo actualizado con los nuevos parámetros.

1.1.2 El proceso de visualización 2D y 3D

Las aplicaciones gráficas se pueden clasificar en dos grandes categorías: 2D y 3D.

Visualización 2D

Las aplicaciones 2D emplean modelos de objetos definidos en un plano bidimensional, o en múltiples planos superpuestos con un orden de prioridad. Aunque predominantemente bidimensionales, pueden incorporar elementos 3D como sombras. El proceso de visualización 2D genera una imagen a partir de:

- **Modelo 2D:** Una estructura de datos que representa elementos como puntos, líneas, polígonos, textos e imágenes.
- **Parámetros de visualización:** Valores que influyen en la imagen final, como la resolución, la posición en pantalla (*viewport*) y la zona del "mundo"2D que se desea mostrar.

Visualización 3D

Las aplicaciones 3D utilizan modelos de objetos definidos en el espacio tridimensional, incluyendo información de aspecto como materiales, texturas y fuentes de luz. El proceso de visualización 3D parte de:

- **Modelo de escena 3D:** Compuesto por un modelo geométrico (conjunto de primitivas, típicamente triángulos, que definen la forma) y un modelo de aspecto (parámetros que definen la apariencia visual).
- **Parámetros de visualización:** Incluyen elementos cruciales como la **cámara virtual**, que define la posición, orientación y ángulo de visión del observador, y el ***viewport***, que especifica la resolución y posición de la imagen en la ventana.

La finalidad es transformar este modelo 3D en una imagen 2D que se proyecta en la pantalla.

1.1.3 Rasterización versus ray-tracing

Existen dos métodos fundamentales para la síntesis de imágenes a partir de modelos geométricos: la rasterización (*rasterization*) y el trazado de rayos (*ray-tracing*).

Rasterización

La **rasterización** es un algoritmo que procesa un conjunto de primitivas de entrada para generar una imagen. El proceso, en esencia, itera sobre cada primitiva (p. ej., un triángulo) y, para cada una, determina el conjunto de píxeles que cubre en la imagen final. Luego, calcula el color correspondiente para cada uno de esos píxeles.

El pseudocódigo básico es el siguiente:

```
Inicializar todos los píxeles al color de fondo.
para cada primitiva P en la escena:
```

```
S = conjunto de píxeles cubiertos por P
para cada píxel q en S:
    c = color de P en la posición de q
    asignar color c al píxel q
```

La complejidad temporal de este algoritmo es proporcional al producto del número de primitivas y el número de píxeles, es decir, $O(n \cdot p)$.

Las Unidades de Procesamiento Gráfico (GPUs) están diseñadas específicamente para ejecutar este proceso de manera altamente eficiente, lo que convierte a la rasterización en el método preferido para aplicaciones interactivas como videojuegos y simuladores.

Ray-Tracing

El **trazado de rayos** (o *ray-tracing*) invierte el orden de los bucles de la rasterización. En lugar de iterar por primitiva, itera sobre cada píxel de la imagen a generar. Para cada píxel, lanza un rayo hacia la escena para determinar qué primitiva es la más cercana y, por lo tanto, visible en esa dirección.

El pseudocódigo básico sería:

```
Inicializar todos los píxeles.
para cada píxel q en la imagen:
    T = subconjunto de primitivas que cubren q
    // Determinar la primitiva más cercana en T
    // y calcular su color
    asignar el color resultante al píxel q
```

Aunque la complejidad básica es también $O(n \cdot p)$, el *ray-tracing* se puede optimizar significativamente mediante el uso de estructuras de datos de indexación espacial, reduciendo la complejidad a $O(p \cdot \log n)$.

Este método, aunque tradicionalmente más lento, es capaz de producir resultados más fotorrealistas al simular efectos ópticos complejos como reflexiones, refracciones y sombras precisas. Por ello, ha sido el preferido para la producción de imágenes no interactivas en cine y efectos especiales. Recientemente, la aparición de hardware dedicado en las GPUs ha permitido su uso en tiempo real en videojuegos, aunque sigue siendo computacionalmente exigente.

1.1.4 El cauce gráfico en rasterización

El **cauce gráfico** (o *graphics pipeline*) es el conjunto de etapas de cálculo que permiten la síntesis de imágenes por rasterización. Este proceso transforma una descripción matemática de los objetos en una imagen de píxeles.

Las entradas son **primitivas** (puntos, líneas o, más comúnmente, triángulos) definidas por una lista de **vértices**. Cada vértice posee, como mínimo, sus coordenadas de posición, pero puede incluir **atributos** adicionales como color, normal (para iluminación) o coordenadas de textura. La salida se escribe en el ***framebuffer***, una zona de memoria que almacena la imagen final.

Las etapas principales del cauce gráfico moderno, muchas de las cuales son programables por el usuario mediante pequeños programas llamados ***shaders***, son:

1) **Procesado de Vértices (*Vertex Processing*):** Procesa cada vértice de entrada de forma independiente. Su principal tarea es la *transformación* de las coordenadas del vértice desde su espacio local (espacio de objeto) hasta el espacio de la pantalla. Esta etapa es programable

a través del ***Vertex Shader***. En cauces más avanzados, pueden existir etapas adicionales como el *tesselation shader* o el *geometry shader*.

2) **Recortado y Ensamblaje de Primitivas (*Clipping and Primitive Assembly*):** Los vértices procesados se agrupan en primitivas (triángulos). Aquellas primitivas que están parcial o totalmente fuera del volumen de visión de la cámara son recortadas (*clipping*). También se puede realizar un cribado de caras (*face culling*) para descartar las caras traseras de los objetos y optimizar el rendimiento.
3) **Rasterización (*Rasterization*):** Convierte cada primitiva geométrica en un conjunto de **fragmentos**, que corresponden a los píxeles que cubre en la pantalla. Para cada fragmento, se interpolan los atributos de los vértices (como el color o las coordenadas de textura). Esta etapa no es programable.
4) **Procesado de Fragmentos (*Fragment Processing*):** El ***Fragment Shader*** (o *Pixel Shader*) se ejecuta para cada fragmento generado en la etapa anterior y calcula su color final. Este cálculo puede ser tan simple como asignar un color plano o tan complejo como simular la interacción de la luz con el material de la superficie, utilizando los atributos interpolados.
5) **Operaciones por Fragmento (*Per-Fragment Operations*):** Antes de escribir el color final en el *framebuffer*, se realizan varias pruebas y operaciones. La más importante es la prueba de profundidad (*Z-buffering*), que asegura que solo los fragmentos más cercanos a la cámara sean visibles, solucionando el problema de la oclusión. También se realizan operaciones de mezcla (*alpha blending*) para simular transparencias.

El paralelismo es una característica clave de este modelo. Las GPUs están diseñadas para procesar múltiples vértices y fragmentos de forma concurrente, lo que permite un rendimiento en tiempo real.

1.2 APIs y motores gráficos

1.2.1 APIs para Rasterización, Ray-tracing y GPGPU

Una **API (Interfaz de Programación de Aplicaciones) gráfica** es una especificación que define un conjunto de funciones y clases para facilitar la creación de aplicaciones gráficas. Proporciona una capa de abstracción entre la aplicación y el hardware subyacente (la GPU), permitiendo el desarrollo de software portable y eficiente.

Evolución y Tipos de APIs

Inicialmente, las aplicaciones escribían directamente en la memoria de vídeo, un método lento y no portable. El uso de APIs estandarizadas resolvió los problemas de portabilidad y coexistencia de aplicaciones, pero la eficiencia mejoró drásticamente con la llegada de las GPUs, que ejecutan el cauce gráfico y reducen el tráfico de datos con la CPU.

Las APIs han evolucionado para ofrecer mayor funcionalidad y programabilidad. Se han incorporado lenguajes de alto nivel para escribir *shaders*, como **GLSL** (OpenGL), **HLSL** (DirectX) y **MSL** (Metal). La programabilidad se ha extendido a más etapas del cauce, y las GPUs se han vuelto omnipresentes en todo tipo de dispositivos.

Las APIs se pueden clasificar en:

- **APIs tradicionales:** Como **OpenGL**, **OpenGL ES** (para sistemas embebidos como móviles) y **DirectX** (hasta la versión 11). Ofrecen un alto nivel de abstracción y son más sencillas de usar, pero con mayor sobrecarga en la CPU.

- **APIs modernas:** Como **Vulkan**, **Metal** y **DirectX 12**. Son de más bajo nivel, lo que permite un control más preciso del hardware, un mejor aprovechamiento del multiprocesamiento y una mayor eficiencia, a costa de una mayor complejidad en el desarrollo.

Para la web, existen APIs como **WebGL**, basada en OpenGL ES, y la más reciente **WebGPU**, basada en Vulkan.

APIs para Ray-Tracing y GPGPU

Además de la rasterización, las GPUs modernas ofrecen soporte para otras tareas computacionales:

- **Ray-Tracing:** Desde 2018, las APIs modernas han incorporado extensiones para acelerar el trazado de rayos por hardware, como **Vulkan Ray-Tracing**, **DirectX Raytracing (DXR)** y **Metal Ray-Tracing**.
- **GPGPU (Cómputo de Propósito General en GPUs):** Las GPUs se utilizan para acelerar tareas no gráficas como la inteligencia artificial, simulaciones científicas o minería de criptomonedas. APIs como **CUDA** (de NVIDIA) y **OpenCL** (de Khronos) están diseñadas para este propósito.

1.2.2 Motores gráficos

Un **motor gráfico** (*game engine*) es un entorno de desarrollo de software que integra un conjunto de herramientas para facilitar la creación de aplicaciones gráficas interactivas complejas, como videojuegos, simuladores o aplicaciones de realidad virtual.

Los motores gráficos proporcionan una capa de abstracción de alto nivel sobre las APIs gráficas (como OpenGL, Vulkan o DirectX). Sus componentes principales suelen incluir:

- **Motor de renderizado:** Se encarga de la visualización de escenas 2D y 3D, utilizando técnicas de rasterización y/o *ray-tracing*.
- **Editor visual:** Una interfaz gráfica (IDE) que permite diseñar escenas de forma interactiva, organizar recursos y configurar objetos sin necesidad de escribir código.
- **Grafo de escena (*Scene Graph*):** Una estructura de datos jerárquica (generalmente un árbol o un grafo acíclico dirigido) que organiza todos los elementos de una escena (objetos, luces, cámaras) y sus relaciones espaciales y de parentesco.
- **Sistema de *scripting*:** Permite a los desarrolladores programar la lógica y el comportamiento de la aplicación. Se pueden utilizar lenguajes de propósito general como C++ o C#, o lenguajes específicos del motor como **GDScript** en Godot. Muchos motores también ofrecen sistemas de *scripting* visual (*visual scripting*), que permiten programar mediante la conexión de nodos en un diagrama de flujo.
- **Otras herramientas:** Suelen incluir sistemas de animación, simulación de físicas, gestión de audio, funcionalidades de red para multijugador, etc.

Algunos de los motores gráficos más relevantes en la actualidad son **Unreal Engine** (de Epic Games), **Unity** (de Unity Technologies), y **Godot** (de código abierto). Estos motores permiten crear aplicaciones portátiles para una amplia gama de plataformas (PC, consolas, móviles, web) sin tener que lidiar con los detalles de bajo nivel de cada una de ellas.

Parte II

Práctica

Capítulo 2

Práctica 1. Escena básica y modos de visualización

2.1 Objetivos

El propósito fundamental de esta práctica es iniciar al estudiante en el entorno de desarrollo integrado (IDE) que ofrece el motor de juegos Godot. Se busca una familiarización con su sistema de nodos y los principios elementales para la construcción de escenas tridimensionales simples. Al concluir esta sesión, el alumno deberá ser competente en la creación de una escena 3D que contenga geometría básica, aplicar materiales para definir el aspecto visual de los objetos, e implementar mecanismos de interacción básicos para el control de la cámara y los modos de visualización mediante entradas de teclado y ratón.

2.2 Requisitos previos

Para la correcta ejecución de esta práctica, es imperativo disponer de una instalación funcional de Godot Engine en su versión 4.0 o superior. Aunque no se requiere experiencia previa en el ámbito de los gráficos por computador, es fundamental poseer conocimientos básicos de programación orientada a objetos. Asimismo, se deberán descargar los ficheros de script proporcionados, con extensión `.gd`, que facilitarán la implementación de funcionalidades específicas como la visualización de ejes coordinados, la generación procedural de una pirámide y el control de una cámara orbital.

2.3 Actividades

2.3.1 Crear un nuevo proyecto Godot

El primer paso consiste en la configuración de un nuevo proyecto en Godot Engine. Este proceso se inicia desde el gestor de proyectos, donde se debe seleccionar la opción "Nuevo proyecto". Es necesario asignar un nombre único al proyecto y especificar un directorio en el sistema de ficheros donde se almacenarán todos sus recursos y configuraciones. Para esta práctica, se utilizará el renderizador por defecto, **Forward+**.

Una vez creado el proyecto, se abre el editor de Godot, que presenta una interfaz para el diseño de escenas, programación de scripts y gestión de recursos. Se procederá a crear una nueva escena 3D, cuyo nodo raíz será de tipo `Node3D`, renombrado a `EscenaPrincipal` para una mejor organización jerárquica. En Godot, una escena es una estructura de datos jerárquica (un árbol) compuesta por nodos que representan los distintos elementos de la aplicación.

2.3.2 Crear un cubo en la escena

La inserción de geometría básica es un procedimiento fundamental en el modelado 3D. Se añadirá un objeto tridimensional con forma de cubo.

1) **Añadir un nodo de malla:** Como hijo del nodo raíz `EscenaPrincipal`, se debe instanciar un nodo de tipo `MeshInstance3D`. Este tipo de nodo se utiliza para visualizar una malla geométrica (`Mesh`) en una escena 3D, asignándole una transformación espacial (posición, rotación y escala).
2) **Asignar la geometría:** En el panel *Inspector*, se debe asignar un recurso de tipo `CubeMesh` a la propiedad *Mesh* del nodo. `CubeMesh` es una clase derivada de `PrimitiveMesh`, que proporciona geometrías predefinidas por el motor.
3) **Posicionar el objeto:** Se renombra el nodo a `Cubo` y se modifica su posición a las coordenadas (0, 0.5, 0) para elevarlo ligeramente sobre el plano base de la escena.

2.3.3 Añadir ejes de coordenadas

Para una mejor comprensión espacial de la escena, es útil visualizar un sistema de ejes de coordenadas. Se utilizará un script proporcionado (`ejes3D.gd`) que genera proceduralmente la geometría de los ejes.

1) Se crea un nuevo nodo de tipo `Node3D`, renombrado a `Ejes3D`.
2) A este nodo se le asocia un nuevo script, cargando el fichero existente `ejes3D.gd`. Este script generará mallas que no se ven afectadas por la iluminación de la escena, sirviendo como una referencia visual constante.

2.3.4 Añadir cámara

La visualización de una escena 3D requiere la presencia de una **cámara virtual**. Este elemento define la posición, orientación y ángulo de visión desde los cuales se renderiza la imagen final.

1) Se instancia un nodo `Camera3D` como hijo del nodo raíz. Este tipo de nodo define un punto de vista para el renderizado.
2) Se posiciona la cámara en `(1.5, 1.5, 2.0)` y se orienta para que apunte hacia el origen `(0, 0, 0)`. Esto se puede lograr mediante un script simple que, en su función `_ready()`, establece la posición y utiliza el método `look_at`.

Código 2.1: Script para posicionamiento inicial de la cámara.

```
1 extends Camera3D
2
3 func _ready():
4     position = Vector3(1.5, 1.5, 2.0)
5     look_at(Vector3(0.0, 0.0, 0.0), Vector3.UP)
```

2.3.5 Asignar materiales

El aspecto visual de un objeto se define mediante **materiales**. Un material describe cómo la superficie de un objeto interactúa con la luz, determinando propiedades como el color, el brillo o la textura.

1) Con el nodo `Cubo` seleccionado, en el panel *Inspector*, se crea un nuevo recurso `StandardMaterial3D` en la propiedad *Surface Material Override*. `StandardMaterial3D` es una clase que implementa un modelo de materiales complejo con múltiples parámetros.
2) Se modifica la propiedad **Albedo** del material, asignándole un color amarillo. El albedo representa el color base de la superficie.

Al ejecutar la escena, se observará el cubo amarillo, pero su color será plano y sin sombreado, ya que aún no hay fuentes de luz que interactúen con el material.

2.3.6 Añadir luz

La iluminación es un componente crucial para el realismo en la visualización 3D. Las fuentes de luz emiten fotones que, al interactuar con los materiales de los objetos, producen el sombreado y los reflejos que percibimos.

1) Se añade un nodo de tipo `DirectionalLight3D` a la escena. Este tipo de luz simula una fuente infinitamente lejana, como el sol, donde todos los rayos de luz son paralelos.
2) Se posiciona y rota la luz para que ilumine las caras visibles del cubo con distinta intensidad, generando así un sombreado que revela su volumen tridimensional. Un modelo de iluminación simple como el de Lambert calcula la intensidad reflejada en función del coseno del ángulo entre la normal de la superficie y la dirección de la luz, lo que provoca que las caras no orientadas directamente hacia la luz aparezcan más oscuras.

Tras añadir la luz, la ejecución mostrará el cubo con un sombreado que distingue sus diferentes caras.

2.3.7 Crear una pirámide (generación por código)

Godot permite la **generación procedural de geometría**, que consiste en crear mallas (`Mesh`) mediante código en tiempo de ejecución. Esto es útil para formas complejas o para geometrías que deben variar dinámicamente. Utilizaremos la clase `SurfaceTool` para construir la malla de una pirámide triángulo a triángulo.

1) Se crea un nuevo nodo `Node3D` llamado `Piramide` y se le asocia el script `piramide.gd`.
2) El script define la función `crear_piramide`, que utiliza un objeto `SurfaceTool` para definir la geometría.
3) Se inicializa el `SurfaceTool` para construir primitivas de tipo triángulo (`Mesh.PRIMITIVE_TRIANGLES`).
4) Se definen los vértices de la base y el ápice. Las caras laterales y la base se construyen añadiendo los vértices de cada triángulo con `add_vertex`.
5) Para cada triángulo, se calcula y asigna su vector normal con `set_normal`. La normal es un vector unitario perpendicular al plano del triángulo, esencial para los cálculos de iluminación.
6) Finalmente, `st.commit()` genera y devuelve un recurso de tipo `ArrayMesh` con la geometría definida. Este recurso se asigna a un nuevo nodo `MeshInstance3D` que se añade a la escena como hijo del nodo `Piramide`.
7) La pirámide se posiciona sobre el cubo en (0, 1, 0).

Código 2.2: Fragmento del script `piramide.gd` para la generación procedural.

```
1 func crear_piramide(h: float) -> ArrayMesh:
2     var st = SurfaceTool.new()
3     st.begin(Mesh.PRIMITIVE_TRIANGLES)
4
```

```
    # 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)

    # Caras laterales (triangulos)
    _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 triangulos)
    _add_triangulo(st, p1, p3, p2, Vector3.DOWN)
    _add_triangulo(st, p1, p4, p3, Vector3.DOWN)

    return st.commit()
```

2.3.8 Cambiar el material (colores y texturas)

De forma análoga a la geometría, los materiales también pueden ser creados y modificados mediante código. Se modificará el script de la pirámide para asignarle un material de color rojo.

1) En la función `_ready()` del script `piramide.gd`, se instancia un nuevo `StandardMaterial3D`.
2) Se modifica su propiedad `albedo_color` para asignarle un color rojizo, por ejemplo, `Color(1.0, 0.1, 0.2)`.
3) El material creado se asigna a la propiedad `material_override` de la instancia de malla de la pirámide. Esto sobreescribe cualquier material que el objeto pudiera tener asignado en el editor.

Este método permite un control dinámico y procedural sobre la apariencia de los objetos, abriendo la puerta a efectos visuales complejos y a la modificación de texturas en tiempo de ejecución.

2.3.9 Controlar una cámara orbital por teclado y ratón

Finalmente, se reemplazará la cámara estática por una cámara orbital interactiva. Este tipo de cámara gira alrededor de un punto de interés (el origen, en este caso), permitiendo al usuario observar la escena desde múltiples ángulos.

1) Se elimina el nodo `Camera3D` anterior.
2) Se crea un nuevo nodo `Camera3D`, se renombra a `Camara3DOrbital`, y se le asocia el script `camara_3d_orbital_simple.gd`.
3) El script gestiona los eventos de entrada del usuario (`_input`) para actualizar los ángulos de órbita (`dxy`) y la distancia al origen (`dz`). Los eventos de teclado (flechas, +/-) y de ratón (botón derecho arrastrado, rueda) son procesados para modificar estos parámetros.
4) La función `_actualiza_transf_vista` recalcula la transformación (`transform`) de la cámara en cada cambio. Utiliza una composición de transformaciones: una traslación a lo largo del eje Z local para establecer la distancia, seguida de rotaciones alrededor de los ejes X e Y

para definir la orientación orbital.

Capítulo 3

Carga de modelos externos y normales

3.1 Objetivos

Esta práctica profundiza en la representación de mallas poligonales y la gestión de modelos 3D en Godot. Los objetivos específicos son:

- Comprender la estructura de las mallas triangulares.
- Aprender a cargar y visualizar modelos 3D de formatos estándar como `glb` y `obj`.
- Distinguir entre los diferentes modos de sombreado que ofrece Godot y entender su impacto visual y de rendimiento.
- Implementar algoritmos para el cálculo de normales en vértices, un requisito indispensable para una correcta iluminación en superficies suaves.
- Generar mallas complejas mediante técnicas de geometría procedural, como la revolución de un perfil 2D.

3.2 Requisitos previos

Se requiere haber completado la Práctica 1 y tener configurado un proyecto base que incluya un nodo raíz, una fuente de luz y la cámara orbital implementada previamente. Es necesario disponer de los scripts `script_raiz.gd`, `utilidades.gd` y `donut.gd`.

3.3 Actividades

3.3.1 Añadir modo de visualización en alambre (wireframe)

La visualización en modo alambre, o *wireframe*, es una técnica de renderizado que muestra únicamente las aristas de los polígonos que componen una malla, en lugar de sus caras rellenas. Este modo es invaluable para la depuración de algoritmos de generación de mallas y para el análisis de la topología de un modelo 3D.

En Godot, este modo se puede activar a nivel de *viewport*. Se implementará una funcionalidad que permita alternar entre el modo de renderizado estándar y el modo *wireframe* al pulsar la tecla 'W'.

1) Se asocia el script `script_raiz.gd` al nodo raíz de la escena.
2) En la función `_init`, se habilita la generación de mallas de alambre en el servidor de renderizado con `RenderingServer.set_debug_generate_wireframes(true)`.
3) La función `_unhandled_key_input` intercepta la pulsación de la tecla 'W'.
4) Al detectar el evento, se alterna el valor de una variable booleana `dibujar_aristas`. Depen-

diendo de su estado, se establece la propiedad `debug_draw` del *viewport* actual a `Viewport.DEBUG_DRAW_WIREFRAME` o `Viewport.DEBUG_DRAW_DISABLED`.

La Figura 16 del guion de prácticas ilustra la diferencia visual entre el renderizado normal y el modo *wireframe* para un modelo de toroide (donut).

3.3.2 Cargar modelos 3D en formato GLB

Godot soporta la importación de diversos formatos de modelos 3D, entre los que se encuentra el formato `glb` (GL Transmission Format Binary). Este formato es eficiente ya que empaqueta en un único fichero binario una escena completa, incluyendo mallas, materiales, texturas y animaciones. El procedimiento de importación es el siguiente:

1) **Obtención del modelo:** Se descarga un modelo en formato `glb` desde un repositorio público como Sketchfab o Poly Pizza.
2) **Importación al proyecto:** El fichero `.glb` se copia al directorio del proyecto. Godot lo detectará automáticamente y lo mostrará en el panel del sistema de archivos. Para una mejor organización, es recomendable crear una subcarpeta `modelos_3D` para alojar los activos importados.
3) **Instanciación en la escena:** El modelo se arrastra desde el panel del sistema de archivos al árbol de la escena, como hijo de un nodo `Node3D` de organización (p.ej., `ObjetosP2`). Godot creará una nueva jerarquía de nodos que representa la estructura interna del fichero `glb`. Opcionalmente, se puede convertir esta instancia en una escena editable para modificar sus componentes de forma individual.

Es posible que sea necesario ajustar la escala del nodo importado para que su tamaño sea coherente con el resto de la escena.

3.3.3 Cargar modelos 3D en formato OBJ

El formato `obj` es otro estándar de facto para modelos 3D, especialmente popular por su simplicidad (es un formato de texto). A diferencia de `glb`, un fichero `.obj` solo contiene la geometría (vértices, normales, coordenadas de textura y definición de caras). La información de materiales se suele almacenar en un fichero `.mtl` asociado, y las texturas en ficheros de imagen separados.

El proceso de carga en Godot es ligeramente diferente:

1) Se descarga el modelo, que consistirá en varios ficheros (`.obj`, `.mtl`, imágenes de textura) y se organizan en una subcarpeta dentro del proyecto.
2) Se crea un nodo `MeshInstance3D` vacío en la escena.
3) El fichero `.obj` se arrastra desde el panel del sistema de archivos y se suelta sobre la propiedad *Mesh* del nodo `MeshInstance3D` en el *Inspector*. Godot cargará automáticamente la geometría y tratará de asociar los materiales y texturas definidos en el fichero `.mtl`.

3.3.4 Cálculo de normales de objetos suaves y tipos de sombreado en Godot

Las **normales de los vértices** son vectores unitarios perpendiculares a la superficie en la posición de cada vértice, y son un atributo fundamental para los algoritmos de iluminación. Para superficies suaves (curvas), una aproximación común y efectiva es calcular la normal de un vértice como el promedio normalizado de las normales de todas las caras adyacentes a dicho vértice. Este método, conocido como el promediado de normales de caras, asume que la malla poligonal es una aproximación de una superficie subyacente continua y diferenciable.

La función `calcNormales` proporcionada en el script `utilidades.gd` implementa este algoritmo: itera sobre todos los triángulos de la malla, calcula la normal de cada cara mediante el producto vectorial de dos de sus aristas, y acumula esta normal en los tres vértices que componen el triángulo. Finalmente, normaliza la normal acumulada en cada vértice.

Asimismo, Godot, como la mayoría de los motores de renderizado en tiempo real, distingue entre dos principales técnicas de sombreado (*shading*):

- **Sombreado por Píxel (Pixel Shading o Fragment Shading):** El cálculo de la iluminación se realiza para cada fragmento (píxel potencial) cubierto por un triángulo. Las normales de los vértices se interpolan de forma perspectiva-correcta para cada fragmento, y esta normal interpolada se utiliza en la ecuación de iluminación. Produce resultados de alta calidad, especialmente para reflejos especulares (brillos), pero es computacionalmente más intensivo.
- **Sombreado por Vértice (Vertex Shading o Gouraud Shading):** La ecuación de iluminación se calcula únicamente en cada vértice de la malla. El color resultante en cada vértice se interpola linealmente a través de la superficie del triángulo para determinar el color de cada píxel interior. Es más rápido pero puede producir artefactos visuales, como la pérdida de detalle en los reflejos especulares si la malla no es suficientemente densa.

En la práctica, se creará un toroide (donut) proceduralmente, se calcularán sus normales con el algoritmo mencionado y se comparará el resultado visual de ambos tipos de sombreado modificando la propiedad `shading_mode` del material.

3.3.5 Normales de objetos con aristas reales (no suaves)

El algoritmo de promediado de normales asume una superficie suave. Para objetos con aristas duras o reales (no suaves), como un cubo, este método produce artefactos de iluminación incorrectos, ya que suaviza visualmente las aristas que deberían ser nítidas. En un cubo, cada vértice es compartido por tres caras mutuamente perpendiculares, por lo que no existe una única normal "suave."[en] esa posición.

La solución consiste en **duplicar los vértices** en las aristas duras. Para un cubo, en lugar de un modelo con 8 vértices compartidos, se utiliza un modelo con 24 vértices. Cada esquina del cubo real corresponde a tres vértices en la misma posición geométrica en el modelo, pero cada uno de estos vértices pertenece a una sola cara (o a caras coplanares) y tiene una normal distinta, perpendicular a dicha cara. De esta manera, al no compartir vértices entre caras no coplanares, no se produce el promediado de normales a través de las aristas, preservando su dureza visual en el renderizado. Esta técnica es fundamental en el modelado de polígonos duros (*hard-surface modeling*) para asegurar una iluminación precisa.

3.3.6 Creación de mallas por revolución de un perfil (geometría procedural)

La **geometría por revolución** es una técnica de modelado procedural que genera una malla 3D al rotar un perfil 2D alrededor de un eje. El perfil es una secuencia de puntos en un plano (por ejemplo, el plano XY) que define una sección transversal del objeto.

El algoritmo a implementar tomará como entrada un perfil 2D (un array de `Vector2`) y el número de subdivisiones angulares (copias del perfil). Por cada punto del perfil, se generarán N vértices en un círculo alrededor del eje de revolución (eje Y). Estos vértices se conectarán para formar una malla de cuadriláteros (descompuestos en dos triángulos cada uno) que constituyen la superficie de revolución. El cálculo de las normales para esta malla generada puede abordarse de dos formas:

1) Aplicar el algoritmo genérico de promediado de normales sobre la malla resultante.
2) Un método más eficiente y preciso consiste en calcular la normal para cada vértice del perfil 2D original (en su plano) y luego rotar este vector de normal junto con el propio vértice alrededor del eje de revolución para obtener las normales de todos los vértices generados.

Esta técnica permite crear eficientemente objetos con simetría axial como vasos, botellas, peones de ajedrez o tornos.

Bibliografía

[1] Ismael Sallami Moreno, **Estudiante del Doble Grado en Ingeniería Informática + ADE**, Universidad de Granada, 2025.