UNIVERSIDAD DE GRANADA

Informática Gráfica

Temario

Ismael Sallami Moreno

Recursos Ingeniería Informática y Ade

Licencia

Informática Gráfica

Ismael Sallami Moreno

https://elblogdeismael.github.io

Índice general

Parte I

Teoría

Capítulo 1

Introducción a los Fundamentos de la Informática Gráfica

1.1 Definición y Alcance de la Disciplina

La Informática Gráfica no debe reducirse conceptualmente a la mera generación de imágenes digitales. Desde una perspectiva académica y rigurosa, se define como la rama de la ingeniería informática dedicada al procesamiento integral de la información geométrica y visual. Esta disciplina se sustenta sobre cuatro pilares fundamentales que estructuran el conocimiento impartido en la asignatura:

- **Modelado Geométrico:** Comprende las técnicas matemáticas y algorítmicas necesarias para representar entidades, tanto reales como abstractas, mediante estructuras de datos computables (vértices, aristas, polígonos y superficies).
- **Visualización (Rendering):** Abarca los procesos de síntesis de imagen a partir de los modelos geométricos definidos previamente. Es la etapa de conversión de datos vectoriales tridimensionales en mapas de bits bidimensionales.
- **Interacción:** Estudio de los mecanismos que permiten al usuario modificar los modelos o los parámetros de visualización en tiempo real, cerrando el ciclo de realimentación hombre-máquina.
- **Computación Geométrica:** Conjunto de operaciones matemáticas (álgebra lineal, cálculo diferencial) aplicadas sobre los modelos para realizar simulaciones, deformaciones o análisis espacial.

1.2 Paradigmas de Visualización y Algoritmia

En el desarrollo de aplicaciones gráficas interactivas, la gestión del tiempo de cómputo es crítica. El sistema opera bajo un bucle continuo, conocido como *Game Loop*, donde se procesan eventos, se actualiza el estado lógico de la simulación y se renderiza el fotograma resultante.

Existen dos filosofías algorítmicas predominantes para resolver el problema de la visibilidad y el coloreado:

1.2.1 Rasterización

Es el estándar predominante en la industria de los gráficos en tiempo real debido a su eficiencia. Su lógica se centra en la proyección de primitivas geométricas sobre el plano de imagen.

$$\text{Complejidad} \approx O(n) \tag{1.1}$$

Donde n es el número de primitivas geométricas. Este método prioriza la velocidad, iterando sobre la geometría para determinar qué píxeles son cubiertos por cada triángulo.

1.2.2 Trazado de Rayos (Ray Tracing)

Históricamente reservado para la generación de imágenes *offline* (cine, arquitectura) debido a su alto coste computacional, aunque recientemente viable en tiempo real mediante hardware especializado (RTX).

$$\text{Complejidad} \approx O(p \cdot \log n) \tag{1.2}$$

Donde p es el número de píxeles. Este algoritmo simula el comportamiento físico de la luz trazando la trayectoria inversa de los fotones desde la cámara hacia la escena, permitiendo efectos realistas como reflexiones, refracciones y sombras suaves de manera natural.

1.3 Arquitectura del Cauce Gráfico (Graphics Pipeline)

El cauce gráfico representa la arquitectura de flujo de datos dentro de la Unidad de Procesamiento Gráfico (GPU). Se divide en etapas secuenciales, algunas de las cuales son programables mediante *shaders*:

1) **Procesamiento de Vértices (Vertex Shader):** Etapa programable donde se transforman los vértices desde el espacio local del objeto hasta el espacio de pantalla proyectado.
2) **Ensamblado de Primitivas y Rasterización:** Etapa de hardware fijo donde los vértices transformados se agrupan en geometrías (triángulos) y se discretizan en fragmentos (candidatos a píxeles).
3) **Procesamiento de Fragmentos (Fragment Shader):** Etapa programable crítica donde se determina el color final de cada píxel, aplicando cálculos de iluminación, texturizado y sombras.
4) **Operaciones de Raster (Framebuffer):** Etapa final donde se realizan pruebas de profundidad (*Z-Buffer*) y mezcla de colores (*Blending*) antes de escribir en la memoria de vídeo.

1.4 Ecosistema Tecnológico: APIs y Motores

Para interactuar con el hardware gráfico, la ingeniería de software moderna utiliza capas de abstracción jerárquicas.

1.4.1 Interfaces de Programación de Aplicaciones (APIs)

Constituyen el nivel bajo de abstracción, permitiendo la comunicación directa con el controlador de la GPU.

- **OpenGL:** API histórica, caracterizada por su alta compatibilidad y relativa sencillez, aunque con menor control sobre la gestión de memoria.
- **Vulkan / DirectX 12 / Metal:** APIs modernas de bajo nivel (*low-level*), diseñadas para reducir la sobrecarga de la CPU y permitir paralelismo masivo, a costa de una complejidad de desarrollo significativamente mayor.

1.4.2 Motores Gráficos (Game Engines)

Representan un nivel alto de abstracción, construidos sobre las APIs mencionadas. Proveen entornos integrados de desarrollo (IDE) con herramientas para física, audio, inteligencia artificial y gestión de escenas.

- **Godot Engine:** Herramienta vehicular seleccionada para el curso. Es un motor de código abierto (*Open Source*) bajo licencia MIT, ligero y versátil. Utiliza una arquitectura basada en nodos y árboles de escena, y su lenguaje de scripting principal es GDScript, sintácticamente similar a Python pero optimizado para la arquitectura del motor.

1.5 Metodología de Evaluación

La asignatura establece un criterio de evaluación mixto y riguroso para garantizar la adquisición de competencias teóricas y prácticas. La calificación final se compone de dos bloques equiponderados:

$$Nota_{Final} = (0{,}5 \cdot Nota_{Teoría}) + (0{,}5 \cdot Nota_{Prácticas}) \tag{1.3}$$

Es condición indispensable obtener una calificación mínima del 40 % (4 puntos sobre 10) en cada una de las partes de forma independiente para poder realizar el promedio. La parte teórica se evalúa mediante examen escrito, mientras que la práctica requiere la defensa de implementaciones en el laboratorio utilizando el motor Godot.

Capítulo 2

El Motor Godot y la Gestión de Mallas Geométricas

2.1 Arquitectura del Motor Godot

El desarrollo de aplicaciones gráficas interactivas mediante el motor Godot se fundamenta en una arquitectura jerárquica y modular. La comprensión de su estructura interna es esencial para la implementación eficiente de algoritmos gráficos.

2.1.1 El Árbol de Escena (Scene Tree)

El núcleo organizativo de Godot es el *Scene Tree*. Toda aplicación se estructura como una jerarquía de nodos, donde la clase base `Node` define el comportamiento genérico.

- **Node2D:** Clase base para elementos en el espacio bidimensional (coordenadas x, y).
- **Node3D:** Clase base para elementos en el espacio tridimensional (coordenadas x, y, z).

La composición de escenas complejas se realiza mediante la anidación de estos nodos, permitiendo la propagación de transformaciones geométricas (traslación, rotación, escalado) de padres a hijos.

2.1.2 El Bucle Principal y GDScript

El flujo de ejecución sigue el paradigma del *Game Loop*. La clase `SceneTree` gestiona el ciclo de vida de la aplicación, invocando métodos críticos como `_process(delta)` en cada fotograma para la actualización lógica y el renderizado.

El lenguaje de scripting nativo, GDScript, aunque sintácticamente similar a Python, está optimizado para la arquitectura del motor. Desde una perspectiva de ingeniería de software, se enfatiza el uso de **tipado estático** (ej. `var x: float`) frente al dinámico, permitiendo optimizaciones en tiempo de compilación y mayor robustez en el código.

2.2 Fundamentos de las Mallas (Meshes)

La unidad atómica de representación geométrica en Godot es la clase `Mesh`. Una malla no es más que una colección estructurada de primitivas geométricas que la GPU puede procesar.

2.2.1 Primitivas Gráficas

El hardware gráfico procesa secuencias de vértices interpretadas según la topología definida:

1) **Puntos y Líneas:** Utilizados principalmente para depuración visual (*debugging*) y representación de estructuras auxiliares (ej. normales, esqueletos).
2) **Triángulos (`PRIMITIVE_TRIANGLES`):** La primitiva fundamental para superficies sólidas.

3) **Tiras de Triángulos (`TRIANGLE_STRIP`):** Optimización topológica donde cada nuevo vértice forma un triángulo con los dos inmediatamente anteriores, reduciendo la redundancia de datos.

2.2.2 Atributos del Vértice

Un vértice en informática gráfica es una estructura de datos compleja que excede la mera posición espacial. Incluye atributos necesarios para el sombreado (*shading*):

- **Posición:** Coordenadas espaciales ($P \in \mathbb{R}^3$).
- **Normal:** Vector unitario ($N \in \mathbb{R}^3$) perpendicular a la superficie, crítico para los cálculos de iluminación.
- **Color:** Valor RGBA interpolado para el sombreado de Gouraud o similares.
- **Coordenadas UV:** Par ordenado (u, v) para el mapeo de texturas.

Nota sobre Arquitectura de Memoria: Godot implementa un diseño de **Estructura de Arrays (SOA)** en lugar de Array de Estructuras (AOS). Esto implica que los atributos se almacenan en arrays contiguos independientes (un array solo para posiciones, otro solo para normales), optimizando la localidad de caché de la GPU.

2.3 Estrategias de Optimización y Renderizado

La eficiencia en el envío de datos del procesador (CPU) a la tarjeta gráfica (GPU) es un cuello de botella habitual que se mitiga mediante técnicas específicas.

2.3.1 Indexación de Vértices

Para evitar la redundancia de datos en mallas conectadas (donde múltiples triángulos comparten un mismo vértice), se utiliza la técnica de **Mallas Indexadas**:

- **Tabla de Vértices (V):** Almacena únicamente los vértices únicos con sus atributos pesados.
- **Tabla de Índices (I):** Array de enteros ligeros que referencian la posición en V para definir la conectividad.

Esta técnica reduce drásticamente el consumo de VRAM y aprovecha la caché de vértices de la GPU.

2.3.2 Modos de Envío: Inmediato vs. Diferido

- **Modo Inmediato (`ImmediateMesh`):** Los datos geométricos se envían al bus gráfico en cada fotograma. Flexible pero computacionalmente costoso. Su uso se restringe a geometría dinámica simple que cambia constantemente.
- **Modo Diferido (`ArrayMesh`):** Los datos se transfieren una única vez a la memoria de vídeo (VRAM). Es el estándar para modelos estáticos y personajes, ofreciendo el máximo rendimiento.

2.4 Generación Procedural: SurfaceTool

Para la construcción algorítmica de geometría compleja desde código, Godot provee la clase `SurfaceTool`. Esta herramienta abstrae la gestión manual de arrays, funcionando como una máquina de estados que permite definir atributos vértice a vértice y generar automáticamente tangentes, normales y la estructura final de la `ArrayMesh` optimizada.

Capítulo 3

Espacios y Transformaciones Afines

3.1 Espacios Afines y Marcos de Referencia

La fundamentación matemática de la informática gráfica reside en la comprensión profunda de las estructuras geométricas que permiten modelar el espacio y las entidades que en él habitan. Es imperativo distinguir formalmente entre vectores y puntos, así como establecer las relaciones algebraicas que los gobiernan.

3.1.1 Estructuras de Espacio Vectorial y Espacio Afín

Un **Espacio Vectorial** de dimensión es un conjunto de elementos denominados vectores libres, denotados comúnmente como . Estos entes representan desplazamientos, direcciones o magnitudes físicas carentes de posición absoluta. En se definen dos operaciones fundamentales:

- **Suma de vectores:** Dados , existe un único vector . Esta operación es conmutativa, asociativa, posee un elemento neutro () y elemento opuesto.
- **Producto por escalar:** Dado y , existe , modificando la magnitud y/o sentido del vector original.

Por contraposición, un **Espacio Afín** sobre es un conjunto de elementos denominados puntos, denotados como , que representan posiciones en el espacio. No existe la suma de puntos en sentido estricto, pero se define la operación de suma de punto y vector:

$$\forall \dot{p} \in A_n, \forall \vec{v} \in V_n, \exists! \dot{q} \in A_n : \dot{q} = \dot{p} + \vec{v} \tag{3.1}$$

De esta definición se deduce la operación de sustracción de puntos, cuyo resultado es un vector: .

Es posible definir rectas en el espacio afín mediante ecuaciones paramétricas. Una recta que pasa por y posee un vector director se expresa como el conjunto de puntos , donde . Asimismo, se definen las **combinaciones afines** de puntos , válidas únicamente si (interpretado como un punto en la recta que los une) o si (interpretado como un vector).

3.1.2 Marcos Afines y Coordenadas Homogéneas

Para representar computacionalmente estos elementos abstractos, es necesario establecer un sistema de referencia. Una **base** de es un conjunto ordenado de vectores linealmente independientes . Un **marco afín** (o sistema de coordenadas) en se constituye por una base de y un punto origen :

$$\mathcal{R} = (\vec{e}_0, \vec{e} * 1, \ldots, \vec{e} * n - 1, \dot{o}) \tag{3.2}$$

Cualquier punto o vector puede expresarse de manera única respecto a utilizando **coordenadas homogéneas**, las cuales son tuplas de componentes .

- Para vectores, : .
- Para puntos, : .

Esta notación unificada es crucial en informática gráfica pues permite tratar transformaciones lineales y traslaciones mediante una única multiplicación matricial.

3.1.3 Producto Escalar y Vectorial

La introducción de nociones métricas (distancia, ángulos) requiere una base especial formada por versores (vectores unitarios) ortogonales entre sí.

- **Producto Escalar:** Operación que resulta en un real. Permite calcular la norma y el ángulo entre vectores . Dos vectores son ortogonales si su producto escalar es nulo.
- **Producto Vectorial (en 3D):** Operación que resulta en un vector perpendicular a ambos operandos. Es anticonmutativo y distributivo.

Un marco se denomina **cartesiano** si su base es ortonormal (vectores unitarios y perpendiculares) y su orientación es a derechas (determinante positivo de la matriz de cambio de base respecto a).

3.2 Transformaciones Afines y Matrices

Una transformación geométrica es una aplicación entre espacios afines. Se denomina **transformación afín** si conserva las combinaciones afines, lo que implica que mapea líneas rectas en líneas rectas y conserva el paralelismo.

$$T(\dot{q}) - T(\dot{p}) = T(\vec{v}) \quad \text{si} \quad \vec{v} = \dot{q} - \dot{p} \tag{3.3}$$

Las transformaciones afines son lineales respecto a vectores: . Se clasifican en singulares (no invertibles) y no singulares (biyectivas).

3.2.1 Representación Matricial

Fijado un marco , toda transformación afín tiene asociada una matriz de dimensiones . Las columnas de corresponden a las coordenadas en de las imágenes de los vectores de la base y del origen:

$$M_T = \begin{pmatrix} | & | & & | \\ T(\vec{e}_0)_{\mathcal{R}} & T(\vec{e}_1)_{\mathcal{R}} & \dots & T(\dot{o})_{\mathcal{R}} \\ | & | & & | \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.4}$$

La aplicación de la transformación se realiza mediante el producto matricial , donde son las coordenadas homogéneas.

- **Composición:** La aplicación sucesiva de transformaciones seguida de se representa mediante la multiplicación de matrices $M * T_2 M_{T_1}$.
- **Cambio de Coordenadas:** Si un punto tiene coordenadas en un marco y se desea expresarlo en un marco , se utiliza la matriz de cambio de base tal que .

3.2.2 Clasificación de Transformaciones

- **Isométricas:** Conservan las distancias (). La submatriz lineal es ortonormal con determinante . Incluyen traslaciones, rotaciones y reflexiones.
- **Rígidas:** Son isometrías que además conservan la orientación (determinante). Incluyen

traslaciones y rotaciones.

3.3 Transformaciones Usuales en Informática Gráfica

3.3.1 Traslaciones

Desplazan puntos mediante la suma de un vector . No afectan a los vectores libres. Su matriz en coordenadas homogéneas es la identidad con el vector de traslación en la última columna.

3.3.2 Escalados y Reflexiones

El escalado multiplica las coordenadas por factores . Si los factores son iguales, es uniforme (conserva ángulos); si no, es no uniforme. Una reflexión es un caso particular de escalado donde alguno de los factores es negativo (comúnmente -1), invirtiendo la orientación. Para reflexiones respecto a planos arbitrarios definidos por una normal , se utilizan matrices de Householder.

3.3.3 Cizallas (Shearing)

Transformaciones que desplazan una coordenada en función de otra (ej.). No son rígidas ni conservan ángulos (salvo casos particulares), pero preservan volúmenes. Su matriz es la identidad con elementos fuera de la diagonal principal.

3.3.4 Rotaciones

Transformaciones rígidas que giran puntos alrededor de un centro (2D) o un eje (3D).

- **En 2D:** Matriz ortonormal con funciones trigonométricas ().
- **En 3D (Ejes principales):** Matrices similares a la 2D aplicadas sobre los planos perpendiculares a los ejes o .
- **En 3D (Eje arbitrario):** Se utiliza la fórmula de Rodrigues para rotar alrededor de un vector unitario . Alternativamente, se emplean **Cuaterniones** (), que ofrecen ventajas computacionales y evitan problemas como el bloqueo de cardán (*gimbal lock*) presente en los ángulos de Euler.

3.4 Transformaciones en Godot

El motor Godot implementa estos conceptos matemáticos mediante clases específicas en su lenguaje GDScript.

3.4.1 Estructuras de Datos

- **Tuplas:** ‘Vector2‘ y ‘Vector3‘ almacenan coordenadas reales (precisión simple). Permiten operaciones algebraicas directas (suma, producto escalar ‘dot‘, producto vectorial ‘cross‘).
- **Matrices:** ‘Transform2D‘ (matriz , fila implícita) y ‘Transform3D‘ (matriz , fila implícita). Almacenan la base transformada y el origen.

3.4.2 Aplicación en Nodos

Los nodos ‘Node2D‘ y ‘Node3D‘ poseen una propiedad ‘transform‘ que almacena la matriz de transformación relativa al padre. Esta propiedad puede modificarse directamente o mediante métodos

auxiliares como ‘translate()‘, ‘rotate()‘ o ‘scale()‘, los cuales componen la transformación actual con la nueva operación. La correcta manipulación de estas matrices es esencial para el posicionamiento y movimiento de objetos en la escena gráfica.

Capítulo 4

Modelos de Objetos y Representación mediante Mallas Indexadas

4.1 Introducción a los Modelos Geométricos

En el ámbito de la Informática Gráfica, la representación digital de entes espaciales constituye el pilar fundamental sobre el que se sustentan los procesos de visualización y simulación. Un **modelo geométrico** se define formalmente como una abstracción matemática diseñada para representar un objeto que reside en un espacio afín, típicamente de dos () o tres dimensiones ().

La condición *sine qua non* para cualquier modelo computacional es que debe permitir la visualización del objeto representado mediante algoritmos. Históricamente y en la práctica actual, distinguimos varias categorías principales de representación:

- **Modelos de Fronteras (B-Rep):** Representan la superficie que delimita al objeto, separando el interior del exterior. La implementación más ubicua de este paradigma son las mallas de polígonos, específicamente las mallas de triángulos.
- **Enumeración Espacial (Vóxeles):** Aproximación volumétrica donde el espacio se discretiza en celdas regulares (vóxeles), clasificando cada una como interior o exterior al objeto. Es análogo al píxel en pero extendido a .
- **Modelos Implícitos y Procedurales:** Se basan en definiciones matemáticas continuas, como las Funciones de Distancia con Signo (*Signed Distance Functions* o SDF).

4.1.1 Formalización Matemática

Desde una perspectiva teórica rigurosa, un objeto se modela como un subconjunto de puntos en un espacio euclídeo . Por ejemplo, una esfera de radio y centro se define como:

$$S = \mathbf{p} \in E \mid |\mathbf{p} - \mathbf{c}| \leq r \tag{4.1}$$

Estos conjuntos deben ser cerrados (contienen a su frontera), acotados (extensión finita) y poseer una superficie diferenciable. Dado que la memoria de un computador es finita y discreta, es imposible representar el conjunto infinito de puntos de de manera exacta en todos los casos, lo que obliga a recurrir a aproximaciones computacionales (mallas o vóxeles) o representaciones algorítmicas.

4.1.2 Modelos Algorítmicos: SDFs

Los modelos procedurales no almacenan geometría explícita, sino que codifican el objeto mediante una función evaluable en cualquier punto del espacio . Distinguimos dos variantes:

1) **Función de Pertenencia:** Devuelve un valor booleano indicando si está dentro del objeto.
2) **Función de Distancia con Signo (SDF):** Devuelve la distancia euclídea mínima desde hasta la superficie del objeto. El signo indica si el punto es interior (negativo) o exterior (positivo).

Las SDFs son cruciales en la visualización moderna (especialmente en *Ray Marching* y redes neuronales profundas para geometría) debido a su capacidad para representar topologías complejas y fractales con precisión arbitraria.

4.2 Modelos de Fronteras: Mallas de Polígonos

Una malla de polígonos (*Polygon Mesh*) es una colección de vértices, aristas y caras que define la forma de un objeto poliédrico. Este modelo aproxima superficies curvas mediante un conjunto finito de facetas planas.

4.2.1 Elementos Topológicos y Geométricos

Es imperativo distinguir entre la **geometría** (la posición espacial de los puntos) y la **topología** (la conectividad entre dichos puntos).

- **Vértice:** Entidad fundamental compuesta por una posición en el espacio afín y, crucialmente, un índice identificador único entero (a). El índice permite abstraer la conectividad de la posición geométrica.
- **Cara:** Superficie plana delimitada por un polígono. Se define topológicamente como una secuencia ordenada de índices de vértices.
- **Arista:** Segmento de recta que conecta dos vértices. Se define por un par de índices de vértices.

4.2.2 Propiedades de las 2-Variedades (2-Manifolds)

Para garantizar la consistencia en algoritmos de renderizado y simulación, las mallas suelen restringirse a ser **2-variedades**. Esto implica que la vecindad local de cualquier punto de la superficie es homeomorfa a un disco abierto en . Las condiciones discretas para que una malla sea una 2-variedad incluyen:

1) Cada arista es compartida por, como máximo, dos caras.
2) Las caras incidentes a un vértice forman un .ªbanicoçontinuo (o un ciclo completo si es un punto interior).
3) No existen vértices aislados ni singularidades topológicas (como dos conos unidos únicamente por el ápice).

Las mallas pueden ser **cerradas** (sin fronteras, encierran un volumen) o **abiertas** (poseen bordes). Una malla es cerrada si y solo si todas sus aristas son adyacentes a exactamente dos caras.

4.2.3 Orientación y Cribado (Culling)

La orientación de una cara se determina por el orden de recorrido de sus vértices. Esto define un vector normal a la superficie. En visualización, es fundamental la coherencia en la orientación (típicamente antihoraria o *Counter-Clockwise* para la cara frontal). El **cribado de caras traseras**

(*back-face culling*) es una técnica de optimización que descarta el renderizado de caras cuya normal apunta en dirección opuesta al observador, asumiendo que son ocultadas por la parte delantera del objeto cerrado.

4.3 Atributos de la Malla

Más allá de la posición espacial, los vértices y caras portan información adicional necesaria para el modelo de iluminación y texturizado:

- **Normales:** Vectores unitarios perpendiculares a la superficie.
  - *Normal de la cara:* Calculada mediante el producto vectorial de dos aristas no colineales del polígono. Es constante para toda la cara (sombreado plano).
  - *Normal del vértice:* Promedio ponderado de las normales de las caras adyacentes. Fundamental para el sombreado de Gouraud o Phong, permitiendo simular curvatura en geometría facetada.
- **Coordenadas de Textura:** Mapean puntos de la superficie 3D a un espacio 2D de imagen (espacio UV).
- **Colores:** Valores RGB asignados a vértices para interpolación.

Es importante notar que las discontinuidades en la superficie (bordes duros) requieren la duplicación de vértices en la misma posición espacial pero con diferentes normales, rompiendo la continuidad topológica para preservar la discontinuidad geométrica visual.

4.4 Estructuras de Datos y Representación en Memoria

La eficiencia del procesamiento gráfico depende críticamente de cómo se organizan estos datos en memoria. Analizamos las estructuras principales:

4.4.1 Triángulos Aislados (Triangle List)

Es la representación más simple. Se almacena una lista lineal de vértices, donde cada terna consecutiva define un triángulo.

$$V = v_0, v_1, v_2, v_3, v_4, v_5, \dots \tag{4.2}$$

Desventaja: Alta redundancia. Un vértice compartido por triángulos se almacena y procesa veces. No codifica topología explícita.

4.4.2 Tiras de Triángulos (Triangle Strips)

Estructura optimizada donde cada nuevo vértice, junto con los dos anteriores, define un nuevo triángulo.

$$\text{Triángulo } i = v_i, v_{i+1}, v_{i+2} \tag{4.3}$$

Reduce la redundancia geométrica, pero impone restricciones en la construcción de la malla y no elimina completamente la duplicación.

4.4.3 Mallas Indexadas (Indexed Meshes)

Es el estándar *de facto* en la industria. Se compone de dos estructuras separadas:

1) **Tabla de Vértices:** Array conteniendo las coordenadas y atributos de cada vértice único.
2) **Tabla de Índices (o Caras):** Array de enteros que referencian a la tabla de vértices.

Esta separación desacopla la topología de la geometría.

- **Eficiencia:** Los vértices se almacenan una sola vez (ahorro de memoria).
- **Cómputo:** La GPU puede cachear los vértices transformados, evitando recálculos.
- **Topología:** La conectividad es explícita a través de los índices compartidos.

4.4.4 Aristas Aladas (Winged-Edge)

Para operaciones que requieren un recorrido eficiente de la topología (como subdivisión o simplificación de mallas), las mallas indexadas son insuficientes (consultas de adyacencia costosas). La estructura de aristas aladas almacena explícitamente relaciones de vecindad en una tabla de aristas. Cada entrada de arista contiene:

- Índices de los vértices extremos (inicial y final).
- Índices de las caras adyacentes (izquierda y derecha).
- Índices de las aristas predecesoras y sucesoras en el ciclo de cada cara.

Esto permite consultas de adyacencia en tiempo constante , a costa de un mayor consumo de memoria y complejidad de mantenimiento.

4.5 Formatos de Archivo

La persistencia de estos modelos se realiza mediante formatos estandarizados:

- **PLY (Polygon File Format):** Flexible, puede ser ASCII o binario. Estructura simple basada en listas de elementos (vértices, caras).
- **OBJ (Wavefront):** Formato de texto ampliamente soportado. Permite índices independientes para posición, normales y texturas (), lo cual optimiza el almacenamiento pero requiere procesamiento para convertirlo a una estructura de malla indexada estricta (donde un índice apunta a un conjunto único de atributos).
- **glTF/GLB:** Estándar moderno de Khronos Group. Diseñado para la transmisión eficiente (JSON para jerarquía + Binario para datos), soportando materiales PBR y escenas complejas.

4.6 Análisis de Complejidad y Eficiencia

El análisis asintótico del almacenamiento revela diferencias significativas. Para una malla cerrada triangular con topología de esfera, si es el número de vértices, el número de caras es y el de aristas (consecuencia de la característica de Euler).

- **Mallas Indexadas:** El coste de memoria es proporcional a . Dado que los índices son enteros (menor tamaño que vectores flotantes), esta es una compresión significativa respecto a los triángulos aislados.
- **Triángulos Aislados:** El coste es . Al ser grande, la redundancia de datos geométricos es masiva (factor de comparado con vértices únicos).

En conclusión, la elección de la estructura de datos (Malla Indexada vs. Tiras vs. Aristas Aladas) representa un compromiso clásico en ingeniería entre eficiencia espacial, velocidad de renderizado y capacidad de manipulación topológica.

Capítulo 5

Modelos Jerárquicos y Grafos de Escena

5.1 Introducción a los Modelos Jerárquicos

En el ámbito de la Informática Gráfica, la gestión de la complejidad geométrica y espacial es un desafío fundamental. Los modelos jerárquicos constituyen una solución estructural a este problema, definiéndose como estructuras de datos organizadas en forma de grafo que representan las relaciones espaciales y lógicas entre los diversos componentes de una aplicación interactiva o una simulación visual.

Esta aproximación permite descomponer objetos complejos en componentes más simples y reutilizables. Un componente se define como un objeto geométrico básico (2D o 3D), tal como una malla poligonal, o una agrupación lógica de otros componentes. Las ventajas inherentes a este enfoque incluyen:

- **Abstracción y Modularidad:** Facilita el diseño de sistemas complejos mediante la composición de entidades primitivas.
- **Reutilización:** Permite instanciar múltiples veces una misma definición geométrica en diferentes contextos espaciales.
- **Colaboración:** Habilita el desarrollo concurrente, donde distintos ingenieros pueden trabajar en componentes aislados sin interferencias destructivas.

5.2 Teoría de Grafos de Escena

El *Scene Graph* o Grafo de Escena es la implementación concreta del modelo jerárquico. Matemáticamente, se modela como un Grafo Dirigido Acíclico (DAG, por sus siglas en inglés), aunque frecuentemente se simplifica a una estructura de árbol.

5.2.1 Tipología de Nodos

Los elementos constitutivos del grafo se clasifican en dos categorías principales:

1) **Nodos Terminales (Hojas):** Representan los objetos geométricos fundamentales que no se subdividen más, típicamente asociados a mallas poligonales con vértices definidos en un espacio local.
2) **Nodos No Terminales (Intermedios):** Actúan como agrupadores lógicos o contenedores de transformaciones que afectan a sus subgrafos descendientes (nodos hijos).

Existe una relación jerárquica estricta donde cada nodo, a excepción del nodo raíz, posee al menos un nodo padre. Las aristas del grafo, que conectan padres con hijos, tienen asociadas transformaciones geométricas afines.

5.2.2 Instanciación y Transformaciones Afines

La renderización de la escena implica el recorrido del grafo. Un objeto instanciado es, en esencia, una réplica de la geometría asociada a un nodo, modificada por una cadena de transformaciones.

Cada nodo N define un **Marco de Referencia Local** ($\mathcal{N}$). Las coordenadas de los vértices almacenados en dicho nodo son relativas a este marco. Para situar estos vértices en el contexto global de la escena, se debe aplicar una transformación afín T. Si denotamos el marco del nodo padre como $\mathcal{P}$, la relación entre ambos marcos se define mediante una matriz de transformación M_N tal que:

$$\mathcal{P}M_N = \mathcal{N} \tag{5.1}$$

La posición final de un vértice en el **Marco de la Escena** (o Espacio de Mundo) se obtiene mediante la composición de las transformaciones acumuladas a lo largo del camino desde la raíz hasta el nodo en cuestión. Si un nodo S_k es descendiente de una cadena de nodos con transformaciones $T_1, T_2, \ldots, T_k$, la transformación global T_{global} aplicada a la geometría de S_k es:

$$T_{global} = T_1 \circ T_2 \circ \cdots \circ T_k \tag{5.2}$$

Esto implica que las operaciones se aplican de izquierda a derecha en términos de composición de funciones, o equivalentemente, multiplicando las matrices de transformación en el orden correspondiente descendente.

5.3 Arquitectura de Grafos de Escena en Godot Engine

El motor gráfico Godot implementa estos conceptos teóricos a través de una arquitectura basada en Nodos, Escenas y Árboles de Escena.

5.3.1 Estructura Fundamental

- **Proyecto:** Unidad contenedora de todos los recursos y configuraciones.
- **Escena:** Un árbol de nodos que puede ser guardado en disco (archivos `.tscn`). Una escena posee siempre un nodo raíz.
- **Nodo:** La unidad atómica de construcción. Todo nodo debe pertenecer a una escena y poseer un único padre (salvo la raíz).
- **Árbol de Escena (SceneTree):** La estructura en tiempo de ejecución que contiene la jerarquía completa de nodos activos.

5.3.2 Tipos de Nodos y Gestión de Memoria

Godot clasifica los nodos según su funcionalidad espacial:

- **Node2D (CanvasItem):** Base para objetos bidimensionales.
- **Node3D:** Base para objetos tridimensionales (anteriormente conocido como Spatial).
- **Instancia de Escena:** Un mecanismo de composición que permite incluir una escena completa dentro de otra como si fuera un único nodo. Esto es crucial para la instanciación múltiple de objetos complejos.

Desde la perspectiva de la ingeniería de software y la eficiencia computacional, es imperativo evitar

la duplicación de datos geométricos pesados. Godot utiliza el patrón de diseño *Flyweight* mediante la clase `MeshInstance` (2D o 3D). Estos nodos no contienen la geometría en sí, sino una referencia a un recurso de tipo `Mesh`. Dado que `Mesh` hereda de `RefCounted`, el motor gestiona automáticamente la memoria, liberando el recurso solo cuando el contador de referencias desciende a cero.

5.4 Matemática de las Transformaciones en Godot

Cada nodo espacial (N) mantiene una propiedad `transform` que codifica la matriz M_N responsable de mapear las coordenadas locales al espacio del padre.

5.4.1 Transformaciones en el Espacio 2D

En 2D, la transformación se representa mediante la clase `Transform2D`, una matriz de 3×2 (asumiendo coordenadas homogéneas implícitas). Los atributos que componen esta matriz son:

- **Position:** Vector de traslación (t_x, t_y).
- **Rotation:** Escalar θ (radianes).
- **Scale:** Vector de escalado (s_x, s_y).
- **Skew:** Ángulo de distorsión o cizalladura.

El orden de composición de estas operaciones es crítico para la consistencia matemática. En Godot, la matriz resultante equivale a aplicar las operaciones en el siguiente orden (de derecha a izquierda en notación matricial sobre vectores columna):

$$M_{2D} = T_{raslacion} \cdot R_{otacion} \cdot Skew \cdot S_{cale} \tag{5.3}$$

5.4.2 Transformaciones en el Espacio 3D

En 3D, se utiliza la clase `Transform3D` (matriz 4×3 o 4×4 conceptualmente). A diferencia del caso 2D, no se incluye el *skew* por defecto, y la rotación es vectorial (ángulos de Euler o Cuaterniones). El orden de composición estándar es:

$$M_{3D} = T_{raslacion} \cdot R_{otacion} \cdot S_{cale} \tag{5.4}$$

Las rotaciones se aplican intrínsecamente en el orden Y-X-Z (Euler), aunque esto es configurable.

5.4.3 Manipulación Programática de Transformaciones

La actualización de la matriz de transformación puede realizarse mediante asignación directa de propiedades o mediante métodos de composición. Es vital distinguir entre:

- **Composición por la Izquierda (Global/Padre):** Métodos como `rotate()` o `translate()` aplican la transformación respecto al marco de referencia del padre. Matemáticamente: $M'_N = M_{operacion} \cdot M_N$.
- **Composición por la Derecha (Local):** Métodos como `rotate_object_local()` aplican la transformación respecto al marco local del objeto. Matemáticamente: $M'_N = M_N \cdot M_{operacion}$.

5.5 Gestión Dinámica del Árbol de Escena

La ingeniería de aplicaciones interactivas requiere la manipulación del grafo en tiempo de ejecución (runtime). Godot proporciona una API robusta para este fin mediante *GDScript*.

5.5.1 Ciclo de Vida de los Nodos

1) **Instanciación:** Se utiliza el método `.new()` de la clase correspondiente. El nodo se crea en un estado "huérfano".
2) **Vinculación:** Se inserta en el grafo mediante `add_child()`. Solo entonces se hace activo y visible en la escena.
3) **Desvinculación:** El método `remove_child()` desconecta el nodo del árbol, devolviéndolo al estado huérfano pero manteniéndolo en memoria.
4) **Destrucción:** Para liberar la memoria, se invoca `queue_free()`. Este método no elimina el nodo inmediatamente, sino que agenda su destrucción segura al finalizar el procesamiento del *frame* actual, evitando errores de referencia colgante durante la ejecución de la lógica.

5.5.2 Búsqueda y Acceso

El acceso a nodos dentro de la jerarquía se realiza mediante rutas relativas o absolutas utilizando `get_node(ruta/al/nodo")`. La nomenclatura de los nodos (`name`) actúa como identificador único entre hermanos dentro del mismo padre.

5.6 Diseño de Grafos Parametrizados

Un concepto avanzado en el modelado jerárquico es la **parametrización**. Esto implica diseñar el grafo de tal manera que las transformaciones de sus nodos dependan de variables externas (parámetros o grados de libertad), en lugar de valores estáticos.

5.6.1 Aplicaciones

- **Animación Procedural:** Vincular parámetros de transformación (rotación, escala) al tiempo (t). Por ejemplo, una rotación continua se define como $\theta(t) = \theta_0 + \omega \cdot t$.
- **Variación de Diseño:** Generar múltiples variantes de un objeto (e.g., un edificio) alterando parámetros como altura o anchura, los cuales propagan cambios de escala a través de la jerarquía.
- **Interacción:** Permitir que las entradas del usuario modifiquen directamente los parámetros de la matriz de transformación.

5.6.2 Implementación Lógica

En la práctica, esto se traduce en scripts que sobrescriben el método `_process(delta)`. En cada ciclo de renderizado, se recalculan las matrices de transformación basándose en el estado actual de los parámetros y el tiempo delta transcurrido, asegurando una animación suave e independiente de la tasa de fotogramas (frame-rate independent).

Capítulo 6

Transformación de Vértices

Este capítulo aborda el conjunto de operaciones matemáticas y algorítmicas necesarias para transformar la geometría de una escena tridimensional, definida mediante vértices, en una representación bidimensional adecuada para su visualización en una pantalla. Este proceso es fundamental en el cauce gráfico (*graphics pipeline*) implementado en hardware por las Unidades de Procesamiento Gráfico (GPUs).

6.1 Espacios de Coordenadas y Transformaciones

El proceso de transformación de vértices implica la conversión secuencial de las coordenadas de los vértices a través de diversos espacios o sistemas de referencia. Cada transición entre espacios se realiza mediante una matriz de transformación específica.

6.1.1 El Cauce Gráfico y sus Etapas

El cauce gráfico de rasterización, utilizado por librerías como OpenGL, DirectX y motores como Godot, procesa los vértices para proyectarlos en el plano de visión. Las etapas principales relacionadas con la geometría son:

1) **Transformación de Coordenadas:** Cálculo de la posición proyectada de cada vértice en la pantalla.
2) **Recortado (*Clipping*):** Eliminación de la geometría que se encuentra fuera del volumen de visión visible.
3) **Rasterización y Eliminación de Partes Ocultas (EPO):** Determinación de los píxeles cubiertos por las primitivas y resolución de visibilidad (típicamente mediante *Z-buffer*).
4) **Iluminación y Texturización:** Cálculo del color final de cada píxel (*shading*).

6.1.2 Secuencia de Sistemas de Coordenadas

Se define una secuencia de seis sistemas de coordenadas fundamentales para el procesamiento de vértices:

1) **Coordenadas de Objeto (OC - *Object Coordinates*):** Coordenadas locales relativas al sistema de referencia propio de cada objeto o nodo.
2) **Coordenadas de Mundo (WC - *World Coordinates*):** Coordenadas globales relativas a un sistema de referencia único para toda la escena.
3) **Coordenadas de Vista u Ojo (EC - *Eye Coordinates*):** Coordenadas relativas a la cámara virtual. El observador se sitúa en el origen de este sistema.
4) **Coordenadas de Recortado (CC - *Clip Coordinates*):** Coordenadas homogéneas resultantes de la proyección. Los vértices visibles se encuentran dentro de un rango normalizado, aunque la componente w puede ser distinta de 1.
5) **Coordenadas Normalizadas de Dispositivo (NDC - *Normalized Device Coordina-***

***tes*):** Obtenidas tras la división por la componente w $(x/w, y/w, z/w)$. El volumen visible es un cubo centrado en el origen con coordenadas en el rango $[-1, +1]$.

6) **Coordenadas de Dispositivo (DC - *Device Coordinates*):** Coordenadas finales en unidades de píxeles (o fragmentos) sobre la ventana de visualización.

6.1.3 Matrices de Transformación

La conversión entre estos sistemas se gestiona mediante matrices de 4×4:

- **Matriz de Modelado (N):** Transforma de OC a WC.
- **Matriz de Vista (V):** Transforma de WC a EC. La composición de N y V se denomina a menudo matriz *Model-View*.
- **Matriz de Proyección (P):** Transforma de EC a CC.
- **Matriz de Viewport (D):** Transforma de NDC a DC (dependiente de la resolución de salida).

6.2 Transformación de Vista

La transformación de vista reorienta la geometría de la escena para alinearla con el sistema de referencia de la cámara virtual.

6.2.1 Definición del Marco de Vista

El marco de referencia de la cámara, $\mathcal{V}$, se define mediante un sistema cartesiano $\{\hat{x}_{ec}, \hat{y}_{ec}, \hat{z}_{ec}, \dot{o}_{ec}\}$. Este marco se construye habitualmente a partir de tres parámetros:

- **Posición del observador ($\dot{o}_{ec}$):** Punto focal de la proyección (PRP).
- **Punto de atención ($\dot{a}$):** Punto hacia el cual apunta la cámara (*Look-at point*). Define, junto con la posición, el eje óptico o eje Z negativo del marco de cámara.
- **Vector hacia arriba ($\vec{u}$):** Vector que indica la orientación vertical de la cámara (*View-up vector*, VUP).

El eje Z del marco ($\hat{z}_{ec}$) se alinea con la dirección opuesta a la visión (vector normal $\vec{n} = \dot{o}_{ec} - \dot{a}$). El eje X ($\hat{x}_{ec}$) se obtiene mediante el producto vectorial normalizado de $\vec{u}$ y $\vec{n}$. Finalmente, el eje Y ($\hat{y}_{ec}$) se obtiene como el producto vectorial de $\hat{z}_{ec}$ y $\hat{x}_{ec}$.

6.2.2 Construcción de la Matriz de Vista

La matriz de vista V es la inversa de la matriz que sitúa la cámara en el mundo. Dado que el marco es ortonormal, V se puede calcular como la composición de una rotación (la transpuesta de la matriz de rotación de la cámara) y una traslación (que lleva la posición del observador al origen).

6.2.3 Implementación en Motores Gráficos

En entornos como Godot, la clase `Camera3D` encapsula esta transformación. La propiedad `transform` del nodo cámara almacena la matriz inversa de vista (V^{-1}), que posiciona la cámara en el mundo. Métodos como `look_at` permiten configurar intuitivamente la orientación de la cámara hacia un objetivo específico.

6.3 Transformación de Proyección

La transformación de proyección convierte el volumen de visión (*view frustum*) en un volumen canónico de coordenadas normalizadas, preparando la geometría para el recorte y la rasterización.

6.3.1 Tipos de Proyección

Existen dos modelos fundamentales de proyección:

- **Proyección Ortográfica (Paralela):** Los proyectores son paralelos entre sí. Mantiene el tamaño de los objetos independientemente de su distancia a la cámara. El volumen de visión es un paralelepípedo rectangular.
- **Proyección Perspectiva:** Los proyectores convergen en un centro de proyección (el ojo). Los objetos más lejanos aparecen más pequeños (escorzo). El volumen de visión es una pirámide truncada (*frustum*).

6.3.2 Parámetros del View-Frustum

El volumen de visión se define mediante seis planos de recorte: izquierdo (l), derecho (r), inferior (b), superior (t), cercano (n, *near*) y lejano (f, *far*). En la proyección perspectiva, estos parámetros determinan la apertura del campo de visión (*Field of View*, FOV) y la relación de aspecto (*aspect ratio*).

6.3.3 La Matriz de Proyección Perspectiva

La matriz de proyección perspectiva Q transforma coordenadas de cámara (EC) a coordenadas de recorte (CC). Una característica distintiva de esta transformación es que modifica la componente homogénea w. Para un punto $(x_{ec}, y_{ec}, z_{ec}, 1)$, la coordenada transformada w_{cc} suele tomar el valor $-z_{ec}$. Esto prepara la posterior división por w (división perspectiva) que normaliza las coordenadas y genera el efecto de perspectiva. La matriz Q también reescala los valores de profundidad Z para que se mapeen al rango $[-1, 1]$ (o $[0, 1]$ dependiendo de la API) de manera no lineal, preservando el orden de profundidad para la eliminación de partes ocultas.

6.4 Recortado y Transformación de Viewport

6.4.1 Recortado (Clipping) y División Homogénea

En el espacio de coordenadas de recortado (CC), se identifican y descartan las primitivas (o partes de ellas) que yacen fuera del volumen canónico de visión. Tras el recorte, se realiza la división por la componente w ($x_{ndc} = x_{cc}/w_{cc}$, etc.) para obtener las Coordenadas Normalizadas de Dispositivo (NDC).

6.4.2 Transformación de Viewport

Finalmente, las coordenadas NDC (en el rango $[-1, 1]$) se transforman linealmente a coordenadas de dispositivo (DC), correspondientes a los píxeles de la ventana o *viewport*. Esta transformación implica un escalado y una traslación determinados por las dimensiones (w, h) y la posición (x_l, y_b) del *viewport* en la ventana. La componente Z también se escala a un rango adecuado para el *buffer* de profundidad (usualmente $[0, 1]$).

Capítulo 7

Fundamentos Avanzados de Iluminación Computacional y Renderizado

7.1 Introducción a la Radiometría y Percepción Visual

El estudio de la síntesis de imágenes realistas requiere una comprensión profunda de la física de la luz y su interacción con la materia, así como de la psicofísica de la percepción humana.

7.1.1 Naturaleza de la Radiación Electromagnética

La luz visible comprende una franja estrecha del espectro electromagnético, específicamente longitudes de onda (λ) entre aproximadamente 390 nm y 750 nm. Aunque la física cuántica describe la luz mediante una dualidad onda-partícula, en la informática gráfica, y específicamente en la óptica geométrica, adoptamos principalmente el **modelo de partículas**.

Bajo este modelo, la luz se conceptualiza como un flujo de fotones que viajan en trayectorias rectilíneas. La magnitud fundamental para cuantificar este flujo es la **Radiancia** (L), definida como la densidad de flujo de energía radiante por unidad de tiempo, por unidad de área proyectada y por unidad de ángulo sólido. Matemáticamente, la radiancia en un punto p en la dirección $\vec{v}$ se denota como $L(\lambda, p, \vec{v})$.

7.1.2 El Sistema Visual Humano y la Teoría del Color

La percepción del color es el resultado de la respuesta espectral de los fotorreceptores en la retina. El sistema visual humano reduce la distribución espectral de potencia continua de la luz entrante a tres valores discretos (tristímulos), correspondientes a la sensibilidad de los conos S, M y L.

Esta reducción dimensional permite modelar el color mediante un espacio vectorial tridimensional. En computación, utilizamos el modelo **RGB**, donde un color se representa mediante la mezcla aditiva de tres primarios: Rojo, Verde y Azul. Matemáticamente, la percepción de una distribución de radiancia L se aproxima mediante una función lineal $f(L) \approx (r, g, b)$. Es crucial notar que el espacio RGB es dependiente del dispositivo; una misma tupla (r, g, b) puede resultar en estímulos psicofísicos diferentes dependiendo de las características del hardware de visualización (monitor o proyector).

7.2 La Ecuación de Renderizado

El comportamiento global de la luz en una escena se describe formalmente mediante la **Ecuación de Renderizado** (Kajiya, 1986). Esta ecuación integral expresa la conservación de la energía radiante en equilibrio. La radiancia saliente L_o desde un punto p en la dirección $\vec{v}$ es la suma de la radiancia emitida por el propio punto (si es una fuente de luz) y la radiancia reflejada proveniente de todas las direcciones incidentes sobre el hemisferio Ω:

$$L_o(p, \vec{v}) = L_{em}(p, \vec{v}) + \int_{\Omega} f_r(p, \vec{v}, \vec{\omega}_{in}) L_{in}(p, \vec{\omega}_{in})(\vec{n} \cdot \vec{\omega}_{in}) d\vec{\omega}_{in} \tag{7.1}$$

Donde:

- L_{em} es la radiancia emitida.
- L_{in} es la radiancia incidente desde la dirección $\vec{\omega}_{in}$.
- f_r es la Función de Distribución de Reflectancia Bidireccional (BRDF).
- $(\vec{n} \cdot \vec{\omega}_{in})$ es el factor de atenuación geométrica (ley del coseno de Lambert).

Esta ecuación integral de Fredholm de segunda especie no tiene solución analítica general, lo que obliga a utilizar métodos numéricos (como Monte Carlo) o modelos simplificados para su resolución.

7.3 El Modelo de Iluminación Local de Phong

Debido a la complejidad computacional de la ecuación de renderizado completa, históricamente se han adoptado modelos empíricos simplificados. El modelo de Phong es el estándar clásico en la rasterización por hardware. Este modelo descompone la interacción de la luz en tres componentes independientes: ambiental, difusa y especular.

7.3.1 Suposiciones y Simplificaciones

El modelo asume fuentes de luz puntuales, ignora las inter-reflexiones complejas (iluminación global) y trata los materiales como opacos. La radiancia total se calcula como:

$$L(p, \vec{v}) = \sum_{i=1}^{N} S_i [f_{am} + f_{dl} + f_{sp}] \tag{7.2}$$

7.3.2 Componente Ambiental

Aproxima la iluminación indirecta global mediante un término constante, evitando que las zonas en sombra sean completamente negras:

$$f_{am} = k_a \cdot C(p) \tag{7.3}$$

Donde k_a es el coeficiente ambiental y $C(p)$ el color base del objeto.

7.3.3 Componente Difusa (Lambertiana)

Modela la reflexión en superficies mate ideales, donde la luz se dispersa uniformemente en todas direcciones. Depende exclusivamente de la posición de la luz respecto a la normal de la superficie $\vec{n}$, no de la posición del observador:

$$f_{dl} = k_d \cdot C(p) \cdot \text{máx}(0, \vec{n} \cdot \vec{l}) \tag{7.4}$$

Donde $\vec{l}$ es el vector unitario hacia la fuente de luz.

7.3.4 Componente Especular (Phong y Blinn-Phong)

Simula los reflejos brillantes característicos de materiales pulidos.

- **Modelo original de Phong:** Se basa en el ángulo entre el vector de reflexión perfecta $\vec{r}$ y el vector hacia el observador $\vec{v}$.
$$f_{ph} = k_s \cdot (\vec{r} \cdot \vec{v})^e \tag{7.5}$$
Donde $\vec{r} = 2(\vec{n} \cdot \vec{l})\vec{n} - \vec{l}$.
- **Modelo de Blinn-Phong:** Una optimización computacional y visualmente más robusta que utiliza el vector medio ("halfway vector") $\vec{h}$, definido como la bisectriz entre $\vec{l}$ y $\vec{v}$:
$$\vec{h} = \frac{\vec{l} + \vec{v}}{||\vec{l} + \vec{v}||}, \quad f_{bp} = k_s \cdot (\vec{n} \cdot \vec{h})^e \tag{7.6}$$

El exponente e controla la "nitidez"del brillo especular (la suavidad de la superficie).

7.4 Modelado Realista y la Función BRDF

Para alcanzar el fotorrealismo, es necesario sustituir las aproximaciones empíricas por modelos basados en la física (Physically Based Rendering - PBR). La pieza central es la BRDF (f_r), que cuantifica la relación entre irradiancia incidente y radiancia reflejada.

7.4.1 Leyes de Fresnel

En la interfaz entre dos medios con diferentes índices de refracción, la luz se divide en un componente reflejado y otro refractado (transmitido). Las ecuaciones de Fresnel dictan esta proporción. Para materiales dieléctricos, la reflectancia especular aumenta drásticamente a medida que el ángulo de incidencia se aproxima a la rasante (90 grados). En computación gráfica, a menudo se utiliza la aproximación de Schlick para el término de Fresnel (F):

$$F(\theta) \approx F_0 + (1 - F_0)(1 - \cos\theta)^5 \tag{7.7}$$

Donde F_0 es la reflectancia en incidencia normal.

7.4.2 Teoría de Microfacetas

Los modelos modernos (como Cook-Torrance o GGX) asumen que, a nivel microscópico, las superficies rugosas están compuestas por pequeños espejos perfectos (microfacetas) orientados aleatoriamente. La BRDF especular se modela como:

$$f_{micro}(\vec{l}, \vec{v}) = \frac{D(\vec{h})G(\vec{l}, \vec{v}, \vec{h})F(\vec{l}, \vec{h})}{4(\vec{n} \cdot \vec{l})(\vec{n} \cdot \vec{v})} \tag{7.8}$$

Donde:

- **D (Distribución de Normales):** Describe la probabilidad estadística de que una microfaceta esté orientada hacia el vector medio $\vec{h}$. La distribución GGX (Trowbridge-Reitz) es el estándar actual por su capacidad para modelar colas especulares largas.
- **G (Geometría/Enmascaramiento-Sombreado):** Modela la auto-oclusión de las microfa-

cetas (shadowing) y el bloqueo de la luz reflejada (masking).
- **F (Fresnel):** La reflectancia física de las microfacetas individuales.

7.4.3 Rugosidad y Anisotropía

El parámetro de rugosidad (α) controla la dispersión de la distribución D. Si $\alpha_x \neq \alpha_y$, el material es anisotrópico (como el metal cepillado), variando su apariencia al rotar la superficie alrededor de su normal.

7.5 Modelos de Fuentes de Luz

La iluminación de una escena depende críticamente de la tipología de las fuentes emisoras.

7.5.1 Fuentes Puntuales y Direccionales

- **Puntual (Point Light):** Emite desde una posición q en todas direcciones. La intensidad decae cuadráticamente con la distancia ($1/r^2$), aunque en motores gráficos se suelen usar funciones de atenuación modificadas para control artístico.
- **Direccional (Distant Light):** Simula una fuente en el infinito (como el sol). Los rayos son paralelos y no hay atenuación por distancia.

7.5.2 Fuentes Avanzadas

- **Spot Light (Foco):** Una fuente puntual restringida a un cono. La intensidad se atenúa angularmente desde el eje central del cono hacia los bordes, a menudo modelado mediante funciones coseno elevadas a una potencia (similar a Phong).
- **Luces de Área:** Emiten luz desde una superficie geométrica (disco, rectángulo). Son esenciales para generar sombras suaves realistas, aunque su coste computacional es elevado.
- **Luces Fotométricas:** Utilizan perfiles de intensidad tabulados (como archivos IES) medidos de luminarias reales, permitiendo patrones de emisión complejos y físicamente exactos.

Capítulo 8

Texturas, Sombreado y Materiales

8.1 Introducción Teórica a las Texturas

En el ámbito de la Informática Gráfica, la representación de superficies realistas requiere ir más allá de la geometría poligonal pura. Los objetos reales presentan variaciones de color, rugosidad y reflectividad a una escala microscópica que sería computacionalmente prohibitivo modelar mediante polígonos individuales. Para resolver esto, introducimos el concepto de **textura**.

Una textura se define formalmente como una función T que mapea un dominio bidimensional D, usualmente normalizado en el intervalo $[0, 1] \times [0, 1]$, a un espacio de atributos del Modelo de Iluminación Local (MIL).

$$T : D \subseteq \mathbb{R}^2 \to \mathcal{A} \tag{8.1}$$

Donde $\mathcal{A}$ representa el conjunto de parámetros modificables, siendo el color difuso ($C(p)$) el más común, aunque también se aplica a coeficientes de especularidad, vectores normales y transparencia.

Existen dos modalidades fundamentales para representar esta función T:

- **Texturas de Imagen (Image Textures):** La función se discretiza en una matriz de elementos denominados *texels* (texture elements).
- **Texturas Procedurales:** La función $T(s)$ se define algorítmicamente mediante un subprograma, permitiendo resolución infinita y patrones matemáticos complejos sin consumo de memoria de almacenamiento de imagen.

8.1.1 Coordenadas de Textura y Mapeo

Para aplicar una función de textura sobre una superficie tridimensional arbitraria, es imperativo establecer una correspondencia biyectiva entre los puntos de la superficie $S \subset \mathbb{R}^3$ y el dominio de la textura D. Sea $p = (x, y, z)$ un punto en la superficie, debe existir una función de proyección f:

$$(u, v) = f(x, y, z) \tag{8.2}$$

Donde el par (u, v) constituye las coordenadas de textura. Esta función suele descomponerse en componentes escalares $u = f_u(p)$ y $v = f_v(p)$.

Estrategias de Asignación

La implementación de la función f se realiza mediante dos estrategias principales:

1) **Asignación Explícita:** Las coordenadas (u, v) se almacenan como atributos directos de los vértices en la malla poligonal. Durante la etapa de rasterización, estas coordenadas se interpolan linealmente a través de la superficie del polígono. Esta técnica es fundamental en

el modelado CAD y requiere atención especial en la continuidad de las aristas (topología de la malla) para evitar artefactos visuales o discontinuidades en el mapeo.

2) **Asignación Procedural:** Las coordenadas se calculan dinámicamente mediante una función matemática basada en la posición espacial del punto. Esto puede realizarse a nivel de vértice (interpolando posteriormente) o a nivel de fragmento (per-pixel) para mayor precisión en superficies no lineales.

Funciones de Proyección Procedural

Se definen diversas topologías de proyección para envolver objetos geométricos:

- **Proyección Planar (Lineal):** Se proyecta el punto p sobre un plano definido por un punto de anclaje q y dos vectores ortonormales base e_u, e_v.

$$u = (p - q) \cdot e_u, \quad v = (p - q) \cdot e_v \tag{8.3}$$

- **Coordenadas Esféricas:** Se utiliza una conversión a coordenadas polares, ideal para objetos con topología esférica. Dado un punto (x, y, z), los ángulos α (azimut) y β (elevación) determinan (u, v):

$$u = \frac{1}{2} + \frac{\text{atan2}(z, x)}{2\pi}, \quad v = \frac{1}{2} + \frac{\text{atan2}(y, \sqrt{x^2 + z^2})}{\pi} \tag{8.4}$$

- **Coordenadas Cilíndricas:** Se proyecta radialmente sobre un cilindro, utilizando el ángulo para u y la altura y normalizada para v.

8.1.2 Filtrado y Consulta de Texels

Dado que la proyección de un píxel de pantalla sobre el espacio de textura raramente coincide exactamente con un texel, se requieren algoritmos de muestreo:

- **Vecino más cercano (Nearest Neighbor):** Selecciona el texel cuyo centroide está más próximo a (u, v). Computacionalmente económico pero propenso a aliasing (pixelación).
- **Interpolación Bilineal:** Calcula el promedio ponderado de los cuatro texels adyacentes a (u, v), suavizando las transiciones y mejorando la calidad visual en primeros planos.

8.2 Teoría del Sombreado (Shading)

El sombreado se refiere al proceso de interpolación de la iluminación sobre las superficies poligonales. En el pipeline gráfico, la evaluación del Modelo de Iluminación Local (MIL) puede ocurrir en diferentes etapas, determinando el costo computacional y la calidad visual.

8.2.1 Sombreado Plano (Flat Shading)

El MIL se evalúa una única vez por polígono, generalmente en su centroide o primer vértice. Se utiliza una única normal n_p para toda la cara.

- **Ventaja:** Alta eficiencia computacional.
- **Desventaja:** Facetado visible y discontinuidades de iluminación en las aristas.
- **Fenómeno Psico-visual:** Exacerba el efecto de las *Bandas de Mach*, una ilusión óptica donde el contraste lateral de la retina exagera los límites entre zonas de intensidad constante.

8.2.2 Sombreado de Gouraud (Vertex Shading)

El MIL se evalúa en cada vértice de la malla, utilizando normales promediadas de las caras adyacentes para simular curvatura. Los colores resultantes C_{vert} se interpolan linealmente en el interior del polígono durante la rasterización.

- **Limitación Crítica:** La interpolación lineal de colores pierde componentes de alta frecuencia, como los brillos especulares (highlights), si estos caen en el interior de un polígono grande y no coinciden con un vértice.

8.2.3 Sombreado de Phong (Pixel Shading)

El MIL se evalúa para cada fragmento (píxel) generado. En lugar de interpolar colores, se interpolan los vectores normales desde los vértices.

- **Ventaja:** Captura brillos especulares precisos y produce gradientes suaves, eliminando casi totalmente las bandas de Mach geométricas.
- **Costo:** Requiere evaluar la ecuación de iluminación millones de veces por frame, aunque es el estándar actual en hardware gráfico moderno.

8.3 Implementación en Motores Gráficos: Godot

El motor Godot implementa estos conceptos teóricos mediante una arquitectura de nodos y un modelo de materiales basado en física (PBR - Physically Based Rendering).

8.3.1 Fuentes de Iluminación

Las luces se modelan como nodos en el árbol de escena (`Light3D`), interactuando con las superficies mediante sus normales y propiedades materiales.

- **DirectionalLight3D:** Simula fuentes en el infinito (como el sol). Los rayos son paralelos y la intensidad no se atenúa con la distancia. Su orientación se define por un vector local, típicamente alineado con el eje Z negativo.
- **OmniLight3D:** Fuente puntual isotrópica. La atenuación sigue la ley del inverso del cuadrado de la distancia:
$$I(r) \propto \frac{1}{r^e} \tag{8.5}$$
Donde $e = 2$ corresponde a la realidad física, aunque se permite su modificación artística.
- **SpotLight3D:** Fuente cónica restringida por un ángulo de apertura θ_{max}. La intensidad decae tanto por distancia como por desviación angular respecto al eje principal del foco.

Adicionalmente, se soporta Iluminación Basada en Imágenes (IBL) mediante `WorldEnvironment` y `PanoramaSkyMaterial`, permitiendo que texturas esféricas de alto rango dinámico iluminen la escena de manera global y difusa.

8.3.2 El Modelo de Material Estándar

La clase `StandardMaterial3D` en Godot encapsula los parámetros del modelo PBR. La ecuación fundamental que determina el color final I de un fragmento es una combinación lineal ponderada por la "metalicidad"del material:

$$I = e + a + (1 - m)(b \cdot d + s \cdot p_\alpha) + m \cdot b \cdot p_\alpha \tag{8.6}$$

Donde:

- e: Emisión (luz propia del material).
- a: Luz ambiental reflejada (proveniente del mapa de entorno).
- m: Factor *metallic* (0.0 a 1.0). Distingue entre dieléctricos y conductores.
- b: Color base (*Albedo*), modulado por texturas.
- d: Reflectividad difusa (Lambertiana o Burley).
- s: Factor especular para no metales (*metallic_specular*).
- p_α: Lóbulo especular (BRDF), dependiente de la rugosidad α (*roughness*).

Análisis de los extremos del modelo

- **Dieléctricos ($m = 0$):** El término predominante es $(b \cdot d + s \cdot p_\alpha)$. El color base afecta a la componente difusa, pero el brillo especular es blanco (o del color de la luz), gobernado por el índice de fresnel implícito en s.
- **Metales ($m = 1$):** La ecuación se simplifica a $b \cdot p_\alpha$. No existe componente difusa. Toda la luz reflejada es especular y está tintada por el color base del material (el albedo define el color del reflejo metálico).

8.3.3 Transparencia y Renderizado

Godot gestiona la transparencia mediante la propiedad `transparency` del material o el canal Alfa del color base. A diferencia de la óptica física real, en el modo de renderizado estándar (rasterización), los objetos transparentes no suelen generar refracción compleja ni proyectar sombras parciales, a menos que se utilicen técnicas avanzadas de ray-tracing o shaders específicos de refracción en espacio de pantalla.

Capítulo 9

Fundamentos de la Interacción en Sistemas Gráficos

9.1 Introducción a los Sistemas Gráficos Interactivos

Un Sistema Gráfico Interactivo (SGI) se define como una arquitectura de software diseñada para mantener un ciclo continuo de retroalimentación con el usuario. A diferencia de los sistemas de procesamiento por lotes, un SGI debe responder a las acciones del operador —típicamente en un intervalo de tiempo imperceptible, del orden de décimas de segundo— y presentar dicha respuesta mediante una visualización gráfica bidimensional o tridimensional.

La arquitectura subyacente de estos sistemas opera mediante un bucle infinito que gestiona una estructura de datos residente en memoria (el modelo). En cada iteración de este ciclo, el sistema espera o detecta una acción externa, captura los datos característicos de dicha acción, modifica el estado interno del modelo en consecuencia y, finalmente, renderiza una nueva imagen que refleja el cambio de estado.

9.1.1 Interactividad frente a Tiempo Real

Es crucial establecer una distinción taxonómica entre sistemas interactivos y sistemas de tiempo real, aunque a menudo convergen:

- **Sistemas Interactivos:** El requisito primordial es una latencia lo suficientemente baja para mantener la percepción de causalidad entre la acción del usuario y la respuesta del sistema.
- **Sistemas de Tiempo Real:** Se rigen por restricciones deterministas donde la latencia debe ser estrictamente menor o igual a un umbral predefinido. Superar este límite constituye un fallo del sistema. Ejemplos críticos incluyen simuladores de vuelo y aplicaciones de realidad virtual (VR) donde la latencia induce cinetosis.

9.1.2 Taxonomía de Dispositivos y Modos de Entrada

La interacción se facilita mediante dispositivos físicos (hardware como teclados, ratones, digitalizadores) y dispositivos lógicos (abstracciones software como un puntero en pantalla o un reconocedor de gestos). La gestión de estos dispositivos se clasifica en tres modos operativos fundamentales:

1) **Modo de Muestreo (Sampling/Polling):** La aplicación interroga activamente el estado del dispositivo en instantes arbitrarios. Es eficiente en memoria pero puede omitir cambios de estado breves si la frecuencia de muestreo es insuficiente.
2) **Modo de Petición (Request):** El sistema detiene su ejecución esperando explícitamente a que ocurra un evento específico. Garantiza la captura del evento pero bloquea el flujo de

ejecución.

3) **Modo de Cola de Eventos (Event Queue):** El sistema operativo o el controlador del dispositivo acumula los cambios de estado en una cola FIFO (First In, First Out). La aplicación procesa esta cola de manera asíncrona, garantizando que no se pierdan eventos sin necesidad de bloqueo ni muestreo constante.

9.2 Arquitectura de Eventos en Godot Engine

El motor Godot implementa un sistema de gestión de entrada basado primordialmente en eventos, encapsulados en la clase base `InputEvent`. Esta arquitectura permite desacoplar la lógica del juego de los detalles del hardware.

9.2.1 Jerarquía y Tipología de Eventos

Los eventos son objetos que transportan información sobre cambios de estado. La jerarquía de clases incluye:

- `InputEventFromWindow`: Eventos originados en una ventana o viewport.
- `InputEventWithModifiers`: Subclase para entradas que pueden alterarse mediante teclas modificadoras (Ctrl, Alt, Shift), abarcando `InputEventKey` (teclado), `InputEventMouse` (ratón) e `InputEventGesture` (pantallas táctiles).
- `InputEventAction`: Eventos abstractos definidos semánticamente en el mapa de entradas.

9.2.2 Flujo de Propagación de Eventos

El procesamiento de eventos en el árbol de escena sigue un orden de prioridad estricto. Cuando se instancia un evento, el motor invoca secuencialmente los siguientes métodos virtuales en los nodos activos:

1) `_input()`: Primer método en ejecutarse. Procesa eventos generales.
2) `_shortcut_input()`: Específico para atajos de teclado y eventos de control.
3) `_unhandled_key_input()`: Captura eventos de teclado no consumidos previamente.
4) `_unhandled_input()`: El último recurso para eventos no procesados, ideal para la lógica del mundo del juego (p.ej., movimiento de cámara) que no debe interferir con la interfaz de usuario (GUI).

Un concepto fundamental es el consumo del evento. Un nodo puede marcar un evento como "manejado"mediante `set_input_as_handled()`, deteniendo su propagación hacia otros nodos en la jerarquía.

9.2.3 Abstracción mediante el Mapa de Entradas (Input Map)

El `InputMap` actúa como una capa de indirección que asocia entradas físicas (teclas específicas, botones de joystick) con acciones semánticas (p.ej., "saltar", "mover_adelante"). Esto permite a los desarrolladores programar lógica basada en acciones (`InputEventAction`) en lugar de hardware específico, facilitando la reconfiguración de controles por parte del usuario final.

9.3 Interfaz de Usuario (GUI) y el Patrón Observador

La interfaz gráfica de usuario en motores gráficos se distingue de la proyección de la escena 3D/2D. En Godot, se construye mediante nodos derivados de la clase `Control`, los cuales poseen propiedades de posicionamiento, dimensionamiento y estilo (temas).

9.3.1 Elementos de Control y Contenedores

La construcción de interfaces complejas se basa en la composición jerárquica de controles.

- **Controles Básicos:** Incluyen `Label` (texto), `Button` (interacción binaria), `TextureRect` (imágenes) y rangos numéricos como `HSlider` o `SpinBox`.
- **Contenedores:** Clases derivadas de `Container` (como `VBoxContainer`, `GridContainer`, `MarginContainer`) que administran automáticamente la disposición espacial de sus hijos, adaptándose a resoluciones dinámicas.

9.3.2 Sistema de Señales

Godot implementa el patrón de diseño Observador a través del sistema de *Signals*. Una señal es un mecanismo de comunicación desacoplado donde un objeto emisor notifica un cambio de estado sin conocer a sus receptores.

- **Definición y Emisión:** Los nodos pueden definir señales personalizadas (keyword `signal`) y emitirlas ante eventos específicos.
- **Conexión:** Los objetos interesados se suscriben (conectan) a dichas señales, vinculándolas a funciones de respuesta (*callbacks*). Esto elimina el acoplamiento fuerte entre componentes lógicos y de interfaz, permitiendo, por ejemplo, que una barra de vida se actualice cuando el jugador recibe daño sin que el jugador tenga referencia directa a la barra de vida.

9.4 Selección y Picking en Entornos Tridimensionales

La selección (*picking*) es el proceso de traducir una interacción 2D (clic del ratón en coordenadas de pantalla) en la identificación de un objeto 3D dentro de la escena.

9.4.1 Técnicas de Implementación

Existen dos paradigmas principales para resolver este problema:

1) **Buffer de Selección (Color Picking):** Renderización de la escena en un framebuffer oculto donde cada objeto se dibuja con un color único que codifica su ID. Al hacer clic, se lee el color del píxel correspondiente. Es preciso a nivel de píxel pero requiere una pasada de renderizado adicional.
2) **Lanzamiento de Rayos (Ray Casting):** Cálculo analítico de la intersección entre un rayo proyectado desde la cámara y la geometría de la escena. Es el método estándar en motores modernos debido a su flexibilidad.

9.4.2 Implementación en Godot mediante Ray Casting

El proceso de selección en Godot utiliza el sistema de física para realizar consultas espaciales:

- **Generación del Rayo:** Se utilizan las funciones de la cámara (`project_ray_origin` y `project_ray_normal`) para desproyectar la posición 2D del ratón hacia un rayo en el espacio 3D.
- **Colisionadores:** La intersección no se calcula contra la malla visual compleja, sino contra versiones simplificadas invisibles denominadas *colliders* (`StaticBody3D` con `CollisionShape3D`). Esto optimiza el coste computacional.
- **Consulta al Espacio de Estado:** Se emplea el objeto `PhysicsRayQueryParameters3D` para configurar la consulta y el método `intersect_ray()` del `PhysicsDirectSpaceState` para obtener el primer objeto interceptado.

9.4.3 Fundamento Matemático: Intersección Rayo-Triángulo

En el nivel más bajo, la selección por rayos implica resolver si una semirrecta $R(t) = O + tD$ intersecta un triángulo definido por los vértices V_0, V_1, V_2. Esto requiere verificar dos condiciones:

1) El rayo debe intersectar el plano que contiene al triángulo.
2) El punto de intersección debe encontrarse dentro de los límites del triángulo, lo cual se determina habitualmente mediante el cálculo de coordenadas baricéntricas (u, v, w) tal que $u \geq 0, v \geq 0, u + v \leq 1$.

Parte II

Práctica

Capítulo 1

Práctica 1. Escena básica y modos de visualización

1.1 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.

1.2 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.

1.3 Actividades

1.3.1 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.

1.3.2 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.

1.3.3 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.

1.3.4 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 1.1: *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)
```

1.3.5 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.

1.3.6 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.

1.3.7 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 1.2: *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
5       # Coordenadas de la base (cuadrado centrado en el origen,
        lado 1)
6       var p1 = Vector3(-0.5, 0, -0.5)
7       var p2 = Vector3( 0.5, 0, -0.5)
8       var p3 = Vector3( 0.5, 0,  0.5)
9       var p4 = Vector3(-0.5, 0,  0.5)
10      var apex = Vector3(0, h, 0)
11
12      # Caras laterales (triangulos)
13      _add_triangulo(st, p1, p2, apex)
14      _add_triangulo(st, p2, p3, apex)
15      _add_triangulo(st, p3, p4, apex)
16      _add_triangulo(st, p4, p1, apex)
17
18      # Base (dos triangulos)
19      _add_triangulo(st, p1, p3, p2, Vector3.DOWN)
20      _add_triangulo(st, p1, p4, p3, Vector3.DOWN)
21
22      return st.commit()
```

1.3.8 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.

1.3.9 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 2

Carga de modelos externos y normales

2.1 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.

2.2 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`.

2.3 Actividades

2.3.1 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`. De-

pendiendo 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).

2.3.2 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.

2.3.3 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`.

2.3.4 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.

2.3.5 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.

2.3.6 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.

Capítulo 3

Resolución práctica 2

En esta sección se abordará el código usado para cumplir con los requisitos de la práctica 2 del curso de Informática Gráfica. Dejando de lado las nociones básicas como importar figuras en diversos formatos vamos a ir tratando el código por bloques, explicando cada uno de ellos.

Vamos a ver el código del fichero que se proporciona de *utilidades.gd*.

Código 3.1: *Código de utilidades.gd*

```
1
2  extends Node  # El script extiende la clase Node de Godot
3
4  ## ------------------------------------------------------------
       ----------------
5  ## Función que calcula las normales promedio de los vértices de
       una malla,
6  ## a partir de la tabla de posiciones de vértices y la tabla de
       triángulos
7
8  @export var normal_length: float = 0.6  # longitud de las líneas
        de las normales
9  @export var normal_color: Color = Color(0.967, 0.83, 0.917, 1.0)
        # color de las normales
10
11 func calcNormales(verts: PackedVector3Array, tris:
       PackedInt32Array) -> PackedVector3Array:
12     # Paso 1: comprobar precondiciones
13     assert(verts.size() >= 3, "CalcNormales: la malla debe tener
        al menos 3 vértices")
14     assert(tris.size() % 3 == 0, "CalcNormales: el número de
       enteros en 'tris' debe ser múltiplo de 3")
15
16     var nv: int = verts.size()  # número de vértices
17     var nt: int = tris.size() / 3  # número de triángulos
18
19     # Paso 2: inicializa normales a cero
20     var normales := PackedVector3Array([])
21     for i in nv:
22         normales.append(Vector3.ZERO)
```

```
23
24      # Paso 3: sumar en cada vértice las normales de sus triá
    ngulos adyacentes
25      for it in nt:
26          var t := Vector3i(tris[3 * it + 0], tris[3 * it + 1],
    tris[3 * it + 2])
27          var a := verts[t[0]]
28          var b := verts[t[1]]
29          var c := verts[t[2]]
30          var normalv := (c - a).cross(b - a).normalized()  #
    calcula la normal del triángulo
31          for iv in 3:
32              normales[t[iv]] += normalv  # suma la normal al vé
    rtice correspondiente
33
34      # Paso 4: normalizar normales
35      for iv in nv:
36          normales[iv] = normales[iv].normalized()
37
38      # Hecho
39      return normales
40
41  ## ------------------------------------------------------------
    ----------------
42  ## Función de parametrización de un toroide (donut)
43  ## u, v: parámetros entre 0 y 1; R: radio mayor; r: radio menor
44
45  func ParamDonut(u, v, r, R: float) -> Vector3:
46      var cu := cos(2.0 * PI * u)
47      var su := sin(2.0 * PI * u)
48      var cv := cos(2.0 * PI * v)
49      var sv := sin(2.0 * PI * v)
50      return Vector3((R + r * cv) * cu, (R + r * cv) * su, r * sv)
51
52  ## ------------------------------------------------------------
    ----------------
53  ## Genera una tabla de triángulos (índices) con topología
    toroidal
54  ## nu: divisiones del primer parámetro; nv: divisiones del
    segundo parámetro
55
56  func GenTriToroidal(nu, nv: int, indices: PackedInt32Array):
57      for i in nu:
58          var isig = (i + 1) % nu
59          for j in nv:
60              var jsig = (j + 1) % nv
61              var i00 = i * nv + j
62              var i01 = i * nv + jsig
```

```
63              var i10 = isig * nv + j
64              var i11 = isig * nv + jsig
65
66              indices.append(i00)
67              indices.append(i11)
68              indices.append(i10)
69              indices.append(i00)
70              indices.append(i01)
71              indices.append(i11)
72
73  ## ----------------------------------------------------------------
    ----------------
74  ## Función que genera un toroide (donut) con 'nu x nv' vértices
75  ## vertices: tabla de vértices; indices: tabla de índices
76
77  func generarDonut(vertices: PackedVector3Array, indices:
    PackedInt32Array,
78                    nu: int = 128, nv: int = 32, R: float = 1.2, r
    : float = 0.4):
79      # Genera vértices con la geometría de un donut
80      for i in nu:
81          for j in nv:
82              vertices.append(ParamDonut(float(i) / nu, float(j) /
     nv, r, R))
83
84      # Genera los triángulos con topología toroidal
85      GenTriToroidal(nu, nv, indices)
86
87  ## ----------------------------------------------------------------
    ----------------
88  ## Función que crea y devuelve un nodo MeshInstance3D que dibuja
     las normales
89  ## de una malla ya existente
90
91  func crear_visualizador_de_normales(malla_instancia:
    MeshInstance3D) -> MeshInstance3D:
92      # Comprobar que el objeto y su malla son válidos
93      if not is_instance_valid(malla_instancia) or not
    is_instance_valid(malla_instancia.mesh):
94          return null
95
96      var malla_original: Mesh = malla_instancia.mesh
97      var transform_global: Transform3D = malla_instancia.
    global_transform
98
99      # Usamos MeshDataTool para leer los datos de la malla
100     var mdt = MeshDataTool.new()
101
```

```
102     # Creamos la malla para dibujar las líneas
103     var immediate_mesh = ImmediateMesh.new()
104     var material = StandardMaterial3D.new()
105     material.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADED
         # sin sombreado
106     material.albedo_color = normal_color  # color de las
        normales
107
108     immediate_mesh.surface_begin(Mesh.PRIMITIVE_LINES, material)
109
110     # Recorremos cada superficie de la malla
111     for i in range(malla_original.get_surface_count()):
112         mdt.clear()
113         # Extraemos los datos de la superficie
114         if mdt.create_from_surface(malla_original, i) == OK:
115             # Recorremos cada vértice para dibujar su normal
116             for v_idx in range(mdt.get_vertex_count()):
117                 var vertice = mdt.get_vertex(v_idx)
118                 var normal = mdt.get_vertex_normal(v_idx)
119                 immediate_mesh.surface_add_vertex(vertice)  #
        inicio de la línea
120                 immediate_mesh.surface_add_vertex(vertice +
        normal * normal_length)  # fin de la línea
121
122     immediate_mesh.surface_end()
123
124     # Creamos el nodo que contendrá las líneas
125     var visualizador = MeshInstance3D.new()
126     visualizador.mesh = immediate_mesh
127     visualizador.name = "VisualizadorNormales_" +
        malla_instancia.name
128
129     # Colocamos el visualizador en la misma posición y rotación
        que el objeto original
130     visualizador.global_transform = transform_global
131
132     return visualizador
```

Cabe destacar que el código que se va a ver en esta sección puede cambiar respecto del que se ofrece en prácticas, ya que se ha ido mejorando y optimizando.

3.1 Problema de los cuadrados

En esta sección se verán los códigos correspondientes al cuadrado de 8 vértices, el cual tiene un problema con las normales y luego se verá el cuadrado de 24 vértices, el cual soluciona el problema de las normales. Se debe mencionar que en el código de `utilidades.gd` se añade la función `crear_visualizador_de_normales` que permite visualizar las normales de cualquier malla. Además,

el script inicial de la raíz esta modificado para hacer que cuando se pulse la tecla `N` se muestren o se oculten las normales de todas las mallas que haya en la escena (al pulsar W se muestran las aristas, funcionalidad que se proporciona en clase). Veremos todos estos códigos.

Código 3.2: *Script de la raíz para alternar visualización de aristas y normales*

```
extends Node3D

# --- Variables de Estado ---
var dibujar_aristas: bool = false  # Indica si se debe mostrar el modo alambre (aristas)
var dibujar_normales_activado: bool = false  # Indica si se deben mostrar las normales
var nodos_visualizadores: Array = []  # Almacena los nodos que visualizan las normales

# --- Funciones de Godot ---

# Se ejecuta al crear el nodo. Activa la generación de wireframes en el motor.
func _init():
    RenderingServer.set_debug_generate_wireframes(true)

# Gestiona la entrada de teclado no procesada por la interfaz.
func _unhandled_key_input(event: InputEvent):
    # Solo actúa cuando la tecla se suelta para evitar repeticiones.
    if event is InputEventKey and not event.pressed:

        # --- Tecla 'W': Alterna el modo alambre ---
        if event.keycode == KEY_W:
            dibujar_aristas = not dibujar_aristas  # Cambia el estado del modo alambre
            if dibujar_aristas:
                dibujar_normales_activado = false  # Desactiva el modo normales si se activa el alambre
            _actualizar_modos_de_vista()

        # --- Tecla 'N': Alterna la visualización de normales --
        elif event.keycode == KEY_N:
            dibujar_normales_activado = not dibujar_normales_activado  # Cambia el estado del modo normales
            if dibujar_normales_activado:
                dibujar_aristas = false  # Desactiva el modo alambre si se activan las normales
            _actualizar_modos_de_vista()
```

```
32
33 # --- Funciones de Ayuda ---
34
35 # Actualiza la vista según el estado de las variables de modo.
36 func _actualizar_modos_de_vista():
37     var viewport = get_viewport()
38
39     # 1. Elimina los visualizadores de normales antiguos.
40     for nodo in nodos_visualizadores:
41         if is_instance_valid(nodo):
42             nodo.queue_free()
43     nodos_visualizadores.clear()
44
45     # 2. Activa el modo correspondiente.
46     if dibujar_aristas:
47         viewport.debug_draw = Viewport.DEBUG_DRAW_WIREFRAME  #
    Activa el modo alambre
48         print("Dibujar en modo aristas: activado")
49     elif dibujar_normales_activado:
50         viewport.debug_draw = Viewport.DEBUG_DRAW_DISABLED  #
    Desactiva el modo alambre
51         print("Mostrando normales...")
52         _buscar_y_crear_visualizadores(get_tree().root)  # Busca
     y crea visualizadores de normales
53     else:
54         viewport.debug_draw = Viewport.DEBUG_DRAW_DISABLED  #
    Desactiva todos los modos de depuración
55         print("Modos de depuración: desactivados")
56
57 # Busca recursivamente en la escena todos los nodos de malla
    visibles y les crea un visualizador de normales.
58 func _buscar_y_crear_visualizadores(nodo_actual: Node):
59     # Si el nodo es una malla visible, crea su visualizador de
    normales.
60     if nodo_actual is MeshInstance3D and nodo_actual.
    is_visible_in_tree():
61         var visualizador = Utilidades.
    crear_visualizador_de_normales(nodo_actual)
62         if is_instance_valid(visualizador):
63             get_tree().root.add_child(visualizador)  # Añade el
    visualizador a la escena
64             nodos_visualizadores.append(visualizador)  # Lo
    guarda para poder eliminarlo después
65
66     # Llama recursivamente a todos los hijos del nodo actual.
67     for hijo in nodo_actual.get_children():
68         _buscar_y_crear_visualizadores(hijo)
```

El script anterior permite alternar entre la visualización de aristas (modo alambre) y la visualización de normales en todas las mallas de la escena mediante las teclas `W` y `N`. Los comentarios explican cada bloque y línea relevante del código, facilitando su comprensión y mantenimiento. Cabe destacar que las partes de añadir iluminación, como añadir un nodo y demás se dan por hecho que el lector ya las sabe hacer, en caso contrario son muy triviales de encontrar en la documentación oficial de Godot, o bien mediante intuición al usar la plataforma.

Código 3.3: *Cubo de 8 vértices: definición, normales y material*

```
extends MeshInstance3D

func _ready() -> void:

    # === 1. Definición de Vértices y Triángulos (Cubo de 8 Vértices, solo Y positivas) ===

    var vertices := PackedVector3Array([
        # Esquina 0: (-X, +Y0, -Z)
        Vector3(-0.5, 0.0, -0.5), # 0
        # Esquina 1: (+X, +Y0, -Z)
        Vector3( 0.5, 0.0, -0.5), # 1
        # Esquina 2: (+X, +Y0, +Z)
        Vector3( 0.5, 0.0,  0.5), # 2
        # Esquina 3: (-X, +Y0, +Z)
        Vector3(-0.5, 0.0,  0.5), # 3
        # Esquina 4: (-X, +Y1, -Z)
        Vector3(-0.5, 1.0, -0.5), # 4
        # Esquina 5: (+X, +Y1, -Z)
        Vector3( 0.5, 1.0, -0.5), # 5
        # Esquina 6: (+X, +Y1, +Z)
        Vector3( 0.5, 1.0,  0.5), # 6
        # Esquina 7: (-X, +Y1, +Z)
        Vector3(-0.5, 1.0,  0.5)  # 7
    ])

    # Cada línea define los índices de los vértices que forman los triángulos de cada cara del cubo
    var triangulos := PackedInt32Array([
        # Cara inferior (Y baja)
        0, 3, 2,  0, 2, 1,
        # Cara superior (Y alta)
        4, 5, 6,  4, 6, 7,
        # Cara frontal (Z-)
        0, 1, 5,  0, 5, 4,
        # Cara trasera (Z+)
        3, 7, 6,  3, 6, 2,
        # Cara lateral derecha (X+)
        1, 2, 6,  1, 6, 5,
```

```
        # Cara lateral izquierda (X-)
        0, 4, 7,  0, 7, 3
    ])

    # 2. Cálculo de Normales Suaves
    # Se calculan las normales promedio para cada vértice usando la función de utilidades
    var normales := Utilidades.calcNormales(vertices, triangulos)

    # 3. Creación y asignación de la Malla
    # Se prepara el array de datos de la malla (vértices, índices y normales)
    var tablas : Array = []
    tablas.resize(Mesh.ARRAY_MAX)
    tablas[Mesh.ARRAY_VERTEX] = vertices
    tablas[Mesh.ARRAY_INDEX] = triangulos
    tablas[Mesh.ARRAY_NORMAL] = normales

    # Se crea la malla y se añade la superficie con los datos anteriores
    mesh = ArrayMesh.new()
    mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES, tablas)

    # 4. Material (Sombreado por píxel)
    # Se crea y configura el material para el cubo
    var mat := StandardMaterial3D.new()
    mat.albedo_color = Color(0.4, 0.4, 1.0)  # Color azul claro
    mat.metallic = 0.3                       # Un poco metálico
    mat.roughness = 0.2                      # Poco rugoso
    mat.shading_mode = BaseMaterial3D.SHADING_MODE_PER_VERTEX  # Sombreado por vértice

    # Se asigna el material a la malla
    material_override = mat
```

Código 3.4: *Cubo de 24 vértices: definición, normales y material*

```
extends MeshInstance3D

func _ready() -> void:
    # === 1. Definición de Vértices ===
    # Se definen 24 vértices, 4 para cada una de las 6 caras del cubo.
    # Esto permite que cada cara tenga su propia normal,
```

```
    logrando un sombreado plano y correcto.
    var vertices := PackedVector3Array([
        # Cara frontal (Z+)
        Vector3(-0.5,  0.5,  0.5), # 0 - Arriba-Izquierda
        Vector3( 0.5,  0.5,  0.5), # 1 - Arriba-Derecha
        Vector3( 0.5, -0.5,  0.5), # 2 - Abajo-Derecha
        Vector3(-0.5, -0.5,  0.5), # 3 - Abajo-Izquierda

        # Cara trasera (Z-)
        Vector3( 0.5,  0.5, -0.5), # 4 - Arriba-Derecha
        Vector3(-0.5,  0.5, -0.5), # 5 - Arriba-Izquierda
        Vector3(-0.5, -0.5, -0.5), # 6 - Abajo-Izquierda
        Vector3( 0.5, -0.5, -0.5), # 7 - Abajo-Derecha

        # Cara derecha (X+)
        Vector3( 0.5,  0.5,  0.5), # 8 - Arriba-Frontal
        Vector3( 0.5,  0.5, -0.5), # 9 - Arriba-Trasera
        Vector3( 0.5, -0.5, -0.5), # 10 - Abajo-Trasera
        Vector3( 0.5, -0.5,  0.5), # 11 - Abajo-Frontal

        # Cara izquierda (X-)
        Vector3(-0.5,  0.5, -0.5), # 12 - Arriba-Trasera
        Vector3(-0.5,  0.5,  0.5), # 13 - Arriba-Frontal
        Vector3(-0.5, -0.5,  0.5), # 14 - Abajo-Frontal
        Vector3(-0.5, -0.5, -0.5), # 15 - Abajo-Trasera

        # Cara superior (Y+)
        Vector3(-0.5,  0.5, -0.5), # 16 - Atrás-Izquierda
        Vector3( 0.5,  0.5, -0.5), # 17 - Atrás-Derecha
        Vector3( 0.5,  0.5,  0.5), # 18 - Adelante-Derecha
        Vector3(-0.5,  0.5,  0.5), # 19 - Adelante-Izquierda

        # Cara inferior (Y-)
        Vector3(-0.5, -0.5,  0.5), # 20 - Adelante-Izquierda
        Vector3( 0.5, -0.5,  0.5), # 21 - Adelante-Derecha
        Vector3( 0.5, -0.5, -0.5), # 22 - Atrás-Derecha
        Vector3(-0.5, -0.5, -0.5)  # 23 - Atrás-Izquierda
    ])

    # === 2. Definición de Triángulos ===
    # Se definen los triángulos para cada cara usando los vértices correspondientes.
    # El orden de los vértices es horario (Clockwise, CW) para que las normales se calculen correctamente.
    var triangulos := PackedInt32Array([
        # Cara frontal (Z+)
        0, 1, 2,  0, 2, 3,
        # Cara trasera (Z-)
```

```
        4, 5, 6,  4, 6, 7,
        # Cara derecha (X+)
        8, 9, 10,  8, 10, 11,
        # Cara izquierda (X-)
        12, 13, 14,  12, 14, 15,
        # Cara superior (Y+)
        16, 17, 18,  16, 18, 19,
        # Cara inferior (Y-)
        20, 21, 22,  20, 22, 23
    ])

    # === 3. Cálculo de Normales ===
    # Se calculan las normales para cada vértice usando la función de utilidades.
    # Al tener vértices duplicados por cara, cada normal será perpendicular a su cara.
    var normales := Utilidades.calcNormales(vertices, triangulos)

    # === 4. Creación de la Malla ===
    # Se prepara el array de datos de la malla (vértices, índices y normales).
    var arrays := []
    arrays.resize(Mesh.ARRAY_MAX)
    arrays[Mesh.ARRAY_VERTEX] = vertices
    arrays[Mesh.ARRAY_INDEX] = triangulos
    arrays[Mesh.ARRAY_NORMAL] = normales

    var new_mesh := ArrayMesh.new()
    new_mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES, arrays)
    mesh = new_mesh

    # === 5. Asignación de Material ===
    # Se crea y configura el material para el cubo.
    var mat := StandardMaterial3D.new()
    mat.albedo_color = Color(0.4, 0.4, 1.0) # Color azul claro
    mat.metallic = 0.3                      # Un poco metálico
    mat.roughness = 0.2                     # Poco rugoso
    mat.shading_mode = BaseMaterial3D.SHADING_MODE_PER_PIXEL # Sombreado por píxel

    material_override = mat
```

Se **debe de evitar** el típico error de definir los triángulos en sentido antihorario (CounterClockwise, CCW), ya que las normales se calcularán al revés y la iluminación no será correcta.

3.2 Creación de mallas por revolución de un perfil

Acorde al guión de prácticas, en esta parte se debe de definir d enuevo un fichero global conc ciertas funciones, como puede ser el cálculo de malla por revolución, entre otros.

Código 3.5: *Función para generar mallas por revolución de un perfil*

```
# Archivo: revolucion_utils.gd
extends Node

const PI = 3.14159265359 # Se usa la constante PI

## Genera una malla indexada por revolución alrededor del eje Y.
## El perfil se asume en el plano X-Y (Vector2(x, y) -> Vector3(x, y, 0)).
func generar_malla_revolucion(
    perfil: PackedVector2Array,
    num_copias: int,
    vertices: PackedVector3Array,
    triangulos: PackedInt32Array
) -> void:

    var num_puntos_perfil = perfil.size()
    if num_puntos_perfil < 2 or num_copias < 3:
        # Se requiere al menos 2 puntos en el perfil y 3 copias para la revolución
        return

    var angulo_paso = 2.0 * PI / float(num_copias)

    # 1. Generación de Vértices
    for i in range(num_copias):
        var angulo = float(i) * angulo_paso
        var cos_a = cos(angulo)
        var sin_a = sin(angulo)

        for j in range(num_puntos_perfil):
            var p2 = perfil[j]
            var x = p2.x
            var y = p2.y

            # Rotación del perfil (x, y, 0) sobre el eje Y
            var x_rot = x * cos_a
            var z_rot = x * sin_a # Rotación en el plano XZ

            # Coordenada Y (altura) permanece igual
            var nuevo_vertice = Vector3(x_rot, y, z_rot)
            # Se añaden vértices al array de salida
```

```
            vertices.append(nuevo_vertice)

    # 2. Generación de Triángulos (Índices)
    # Se crean cuadriláteros (quads) y cada uno se divide en dos triángulos.
    for i in range(num_copias):
        # El índice 'siguiente_i' conecta el último segmento con el primero (cierre completo)
        var siguiente_i = (i + 1) % num_copias

        for j in range(num_puntos_perfil - 1):
            # Índices en la capa actual (i) y la siguiente (i+1)
            var i0 = i * num_puntos_perfil + j           # Vértice A (i, j)
            var i1 = siguiente_i * num_puntos_perfil + j  # Vértice B (i+1, j)
            var i2 = siguiente_i * num_puntos_perfil + j + 1 # Vértice C (i+1, j+1)
            var i3 = i * num_puntos_perfil + j + 1           # Vértice D (i, j+1)

            # Triángulo 1: (A, B, D)
            triangulos.append(i0)
            triangulos.append(i1)
            triangulos.append(i3)

            # Triángulo 2: (B, C, D)
            triangulos.append(i1)
            triangulos.append(i2)
            triangulos.append(i3)
```

Como ejemplos vamos a ver la creación de una esfera y la de una pieza de peón del ajedrez.

Código 3.6: *Generación de una esfera por revolución de perfil*

```
# Archivo: malla_revolucion.gd
extends MeshInstance3D

# Parámetros exportados para configurar el modelo desde el editor
@export var num_copias: int = 64                # Número de copias del perfil (divisiones horizontales)
@export var radio: float = 1.0                  # Radio de la esfera
@export var sombreado_por_pixel: bool = true    # Permite alternar el modo de sombreado

```

```
## Función que genera el perfil 2D de media circunferencia para la esfera
func generar_perfil_esfera(segmentos_verticales: int) -> PackedVector2Array:
    var perfil = PackedVector2Array()
    var R = radio
    # Recorre los segmentos verticales para crear puntos desde abajo (PI) hasta arriba (0)
    for i in range(segmentos_verticales + 1):
        var angulo = PI * float(i) / float(segmentos_verticales)
        # Calcula la posición X e Y del punto en el perfil usando funciones trigonométricas
        var x_coord = R * sin(angulo)   # Componente X del perfil
        var y_coord = R * cos(angulo)   # Componente Y del perfil
        perfil.append(Vector2(x_coord, y_coord)) # Añade el punto al perfil
    return perfil

func _ready() -> void:
    # Define el número de segmentos verticales para la esfera
    var segmentos_verticales = 32
    # Genera el perfil de media circunferencia
    var perfil_actual = generar_perfil_esfera(segmentos_verticales)

    # Inicializa los arrays de salida para vértices y triángulos
    var vertices   := PackedVector3Array([])
    var triangulos := PackedInt32Array([])

    # Genera la malla por revolución usando el perfil y el nú mero de copias
    RevolucionUtils.generar_malla_revolucion(perfil_actual, num_copias, vertices, triangulos)

    # Calcula las normales promedio para cada vértice
    var normales := Utilidades.calcNormales(vertices, triangulos)

    # Prepara el array de datos de la malla (estructura SOA)
    var tablas : Array = []
    tablas.resize(Mesh.ARRAY_MAX)
    tablas[Mesh.ARRAY_VERTEX] = vertices      # Posiciones de los vértices
    tablas[Mesh.ARRAY_INDEX]  = triangulos    # Índices de los triángulos
    tablas[Mesh.ARRAY_NORMAL] = normales      # Normales de los
```

```
    vértices
44
45      # Crea la malla y añade la superficie con los datos
    anteriores
46      mesh = ArrayMesh.new()
47      mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
    tablas)
48
49      # Crea y configura el material para la esfera
50      var mat := StandardMaterial3D.new()
51      mat.albedo_color = Color(0.2, 0.5, 1.0)   # Color azul claro
52      mat.metallic = 0.3                        # Un poco metálico
53      mat.roughness = 0.2                       # Poco rugoso
54
55      # Configura el modo de sombreado según el parámetro
    exportado
56      if sombreado_por_pixel:
57          mat.shading_mode = BaseMaterial3D.SHADING_MODE_PER_PIXEL
        # Sombreado por píxel
58      else:
59          mat.shading_mode = BaseMaterial3D.
    SHADING_MODE_PER_VERTEX # Sombreado por vértice
60
61      material_override = mat
```

Código 3.7: *Generación de un peón de ajedrez por revolución de perfil*

```
1  # Archivo: peon_ajedrez_revolucion.gd
2  extends MeshInstance3D
3
4  @export var num_copias: int = 64                        # Número
       de copias del perfil (divisiones horizontales)
5  @export var segmentos_verticales_por_tramo: int = 16    #
       Segmentos por tramo entre puntos clave del perfil
6  @export var sombreado_por_pixel: bool = true            #
       Permite alternar el modo de sombreado
7
8  # Constante que define los puntos clave del perfil 2D del peón
9  const PUNTOS_PEON_CLAVE: PackedVector2Array = [
10     Vector2(0.0, -1.0), Vector2(0.5, -1.0), Vector2(0.55, -0.8),
11     Vector2(0.2, -0.4), Vector2(0.3, 0.1), Vector2(0.1, 0.5),
12     Vector2(0.35, 0.8), Vector2(0.2, 0.95), Vector2(0.0, 1.0)
13 ]
14
15 # Genera el perfil suavizado interpolando linealmente entre los
       puntos clave
```

```
16 func generar_perfil_peon(segmentos: int) -> PackedVector2Array:
17     var perfil = PackedVector2Array()
18     # Para cada tramo entre dos puntos clave, interpola '
   segmentos' puntos
19     for i in range(PUNTOS_PEON_CLAVE.size() - 1):
20         var p_inicio = PUNTOS_PEON_CLAVE[i]
21         var p_fin = PUNTOS_PEON_CLAVE[i + 1]
22         for j in range(segmentos):
23             perfil.append(p_inicio.lerp(p_fin, float(j) / float(
   segmentos)))
24     perfil.append(PUNTOS_PEON_CLAVE[-1]) # Añade el último punto
    clave
25     return perfil
26
27 func _ready() -> void:
28     # Genera el perfil 2D del peón usando la función de
   interpolación
29     var perfil_actual = generar_perfil_peon(
   segmentos_verticales_por_tramo)
30
31     # Inicializa los arrays de salida para vértices y triángulos
32     var vertices := PackedVector3Array([])
33     var triangulos := PackedInt32Array([])
34
35     # Genera la malla por revolución usando el perfil y el nú
   mero de copias
36     RevolucionUtils.generar_malla_revolucion(perfil_actual,
   num_copias, vertices, triangulos)
37
38     # Calcula las normales promedio para cada vértice
39     var normales := Utilidades.calcNormales(vertices, triangulos
   )
40
41     # Prepara el array de datos de la malla (estructura SOA)
42     var tablas: Array = []
43     tablas.resize(Mesh.ARRAY_MAX)
44     tablas[Mesh.ARRAY_VERTEX] = vertices      # Posiciones de
   los vértices
45     tablas[Mesh.ARRAY_INDEX] = triangulos     # Índices de los
   triángulos
46     tablas[Mesh.ARRAY_NORMAL] = normales      # Normales de los
   vértices
47
48     # Crea la malla y añade la superficie con los datos
   anteriores
49     mesh = ArrayMesh.new()
50     mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
   tablas)
```

```
51
52      # Crea y configura el material para el peón
53      var mat := StandardMaterial3D.new()
54      mat.albedo_color = Color(1.0, 1.0, 0.8)   # Color marfil
        claro
55      mat.metallic = 0.5                        # Más metálico
56      mat.roughness = 0.4                       # Más rugoso
57
58      # Configura el modo de sombreado según el parámetro
        exportado
59      if sombreado_por_pixel:
60          mat.shading_mode = BaseMaterial3D.SHADING_MODE_PER_PIXEL
          # Sombreado por píxel
61      else:
62          mat.shading_mode = BaseMaterial3D.
        SHADING_MODE_PER_VERTEX # Sombreado por vértice
63
64      material_override = mat
```

Capítulo 4

Resolución práctica 3

4.1 Introducción a la Práctica de Grafos de Escena

4.1.1 Objetivos de la Práctica 3

La Práctica 3, titulada "Grafos de escena", tiene como propósito fundamental la aplicación de los conceptos de modelado jerárquico en el entorno de Godot Engine.

Los objetivos específicos que deben ser cubiertos por los estudiantes son:

1) Aprender a diseñar e implementar **modelos jerárquicos de objetos articulados**.
2) Aprender a **crear el grafo de escena** que formaliza la jerarquía.
3) Comprender el funcionamiento de la **pila de transformaciones** (o composición de transformaciones).
4) Aprender a **modificar interactivamente parámetros** del modelo.
5) Aprender a **implementar animaciones sencillas**.

4.1.2 Marco Teórico: Modelos Jerárquicos y Transformaciones

Concepto de Grafo de Escena

En Godot y en los motores gráficos en general, el **grafo de escena** (o jerarquía) es una estructura de datos, generalmente un árbol o un grafo dirigido acíclico, que modela las **relaciones jerárquicas** entre los objetos que componen la aplicación, tales como cámaras, luces y objetos geométricos.

Definición 4.1 (Modelado Jerárquico). Un objeto articulado o compuesto se modela mediante un nodo no terminal en el grafo, que contiene instancias de otros nodos (hijos). Los objetos simples (mallas) actúan como nodos terminales.

El desarrollo de aplicaciones en Godot se organiza en torno a **proyectos, escenas y nodos**. Una escena es una estructura jerárquica de nodos (un árbol de escena) que representan los elementos de la aplicación.

Transformaciones y Composiciones

Cada nodo en Godot (`Node2D` o `Node3D`) tiene asociado un **marco afín** local, $\mathcal{N}$. Este marco $\mathcal{N}$ se define en relación con el marco de su nodo padre, $\mathcal{P}$, mediante una matriz de transformación M_N, llamada **transformación del nodo**. La relación se expresa como $\mathcal{P}M_N = \mathcal{N}$.

La Pila de Transformaciones

Las coordenadas de los vértices de una malla se consideran expresadas en el marco $\mathcal{N}$ del nodo (coordenadas locales). Al visualizar un objeto, la transformación final que se aplica es la composición de las transformaciones de todos los nodos en el camino desde la raíz hasta el nodo terminal.

Para un nodo N, la matriz de transformación M_N es la composición de varias transformaciones, y el orden es crucial. En el contexto de Godot 3D (`Node3D`), la transformación se compone generalmente (de derecha a izquierda) como una secuencia de **Escalado, Rotación y Traslación**.

Proposición 4.1 (Transformación de Nodo 3D). La transformación de un nodo N se construye mediante la composición de transformaciones geométricas afines:

$$M_N = T \circ R \circ S$$

Donde T es la Traslación (definida por `position`), R es la Rotación (definida por `rotation` o cuaternión), y S es el Escalado (definido por `scale`).

Es importante notar la distinción entre composición por la izquierda (actúa en el marco del padre, $M_{nuevo} = T_{padre} \cdot M_{viejo}$) y composición por la derecha (actúa en el marco local del objeto, $M_{nuevo} = M_{viejo} \cdot T_{local}$). Los métodos como `rotate_object_local` componen por la derecha, interpretando los vectores de entrada en el marco local del nodo.

4.1.3 Requisitos Previos y Configuración del Proyecto

Requisitos y Estructura Base

Para comenzar la Práctica 3, es indispensable haber **completado la Práctica 2**.

El proyecto debe mantener la estructura base establecida en las prácticas anteriores, incluyendo:

- Un nodo raíz de la escena principal.
- Un nodo para la **cámara orbital** (`Camara3DOrbital`, con el script `camara_3d_orbital_simple.gd`).
- Un nodo para una **fuente de luz** (como `DirectionalLight3D`).
- Un nodo para el objeto que contiene los **ejes visibles** (`Ejes3D`).
- Un nodo padre de todos los objetos articulados, sugerido como `ObjetosP3` o `GrafoP3`.

Organización del Proyecto

Se recomienda encarecidamente una organización de archivos clara para facilitar la gestión de recursos:

- Utilizar una carpeta `modelos_3D` para los modelos externos (`glb`, `obj`).
- Utilizar una carpeta `scripts`. Dentro de esta, separe los scripts:
  - `scripts/nodos`: para scripts asociados a nodos específicos del grafo.
  - `scripts/globales`: para scripts globales (autoloads), como `utilidades.gd`.

Debe recordarse que los nodos y recursos deben tener **nombres descriptivos** (ej., `PelotaTenis` en lugar de `MeshInstance5`).

4.2 Desarrollo Detallado de las Actividades

4.2.1 Actividad 1: Diseño del Grafo de Escena

El primer paso es el **diseño formal** del modelo jerárquico.

Definición del Modelo Articulado

El modelo debe ser un objeto articulado con **al menos tres articulaciones** (grados de libertad), que se controlarán mediante giros y desplazamientos. Al menos dos de estas articulaciones deben ser **dependientes** entre sí.

Ejercicio 4.2.1. Diseño del Grafo de Escena Articulado Diseñe un objeto jerárquico. Un ejemplo sugerido es una grúa que posea tres grados de libertad (DOF):

1) Ángulo de giro del brazo principal (rotación en Y).
2) Desplazamiento del gancho a lo largo del brazo.
3) Altura del gancho (desplazamiento vertical o extensión de un segmento).

Este diseño se debe plasmar en un documento PDF que detalle el grafo, las transformaciones involucradas, los parámetros (grados de libertad), y las referencias a los objetos terminales (mallas). Los nodos terminales pueden provenir de:

- Objetos predefinidos de Godot (`CubeMesh`).
- Objetos creados en Práctica 1 (ej., la pirámide).
- Modelos importados (`glb`, `obj`) de Práctica 2.
- Objetos de revolución generados proceduralmente (Práctica 2).

4.2.2 Actividad 2: Creación del Modelo en Godot

Una vez diseñado el grafo, se procede a su implementación en Godot. Esto implica la creación de nodos (`Node3D`) para la estructura jerárquica y nodos `MeshInstance3D` para los elementos geométricos.

Implementación de la Jerarquía

La estructura jerárquica debe reflejar la dependencia de las transformaciones. Por ejemplo, si el *Brazo* rota, todos sus hijos (como el *Gancho*) deben rotar con él.

Proposición 4.2 (Reutilización de Mallas). Para optimizar la memoria y el rendimiento, es crucial no duplicar mallas complejas. Los nodos `MeshInstance3D` deben contener únicamente una referencia a la malla (`mesh`), permitiendo que múltiples instancias compartan el mismo objeto `ArrayMesh` o `CubeMesh`.

La implementación se puede realizar en el editor o mediante scripts en la función `_ready()` del nodo raíz (`EscenaPrincipal` o `ObjetosP3`), creando los nodos descendientes programáticamente.

4.2.3 Actividad 3: Generación de una Animación del Modelo

La animación se logra **modificando los atributos de transformación** de los nodos que controlan los grados de libertad en cada *frame* de ejecución.

Implementación de la Animación en GDScript

La lógica de actualización continua debe residir en el método `_process(delta)` del script asociado al nodo que se desea animar. El parámetro `delta` representa el tiempo transcurrido (en segundos) desde el último frame.

Para una **rotación continua** (como la del brazo de la grúa alrededor del eje Y) se puede implementar de la siguiente forma:

Código 4.1: *Ejemplo de Animación de Rotación 3D (Brazo)*

```
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)
```

Para implementar animaciones sencillas, se pueden utilizar variaciones **lineales** o **oscilantes**. Un movimiento oscilante (útil para el desplazamiento del gancho o la altura) se puede lograr utilizando la función trigonométrica seno (sin):

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

Donde a y b son los valores mínimo y máximo, w es la frecuencia de oscilación, y t es el tiempo acumulado. La actualización del estado debe hacerse en `_process(delta)`.

4.2.4 Actividad 4: Activación y Desactivación de la Animación

Para permitir la interacción, se debe implementar la capacidad de activar o desactivar la animación de cada articulación mediante la entrada del usuario.

Gestión de Eventos y Acciones de Entrada

Se recomienda usar el sistema `Input Map` de Godot para definir acciones, como `activar_cabeza` o `activar_brazo`, y asociarlas a teclas, como las numéricas 1 a 9.

Dentro del método `_process(delta)`, se verifica si se ha pulsado la acción de entrada deseada mediante `Input.is_action_just_pressed(accion)`:

Código 4.2: *Ejemplo de Activación/Desactivación de Animación*

```
extends Node3D

@export var activar := "activar\_brazo" \# Acción definida en
    Input Map
@export var rotation\_speed\_deg := 10.0
var activa := true

func \_process(delta):
    \# 1. Manejo de la entrada para alternar el estado
    if Input.is\_action\_just\_pressed(activar):
            activa = !activa

    \# 2. Aplicar la transformación solo si está activa
    if activa:
            rotation.y += deg\_to\_rad(rotation\_speed\_deg *
    delta)
```

Este mecanismo de control basado en el estado (la variable `activa`) permite la **modificación interactiva de parámetros** del modelo, cumpliendo con uno de los objetivos clave de la práctica.

4.3 Entrega de la Práctica

La entrega final debe consistir en dos elementos:

1) La **carpeta del proyecto Godot** completa, comprimida en formato ZIP. El proyecto debe estar organizado según las recomendaciones de la Práctica 2 (incluyendo subcarpetas para scripts y modelos).
2) Un **documento breve** (adicional a esta guía) que contenga:
   - El **grafo de escena** diseñado en la Actividad 1.
   - Una explicación de los **scripts de interacción** implementados para controlar los grados de libertad y la animación.

El diseño del grafo de escena debe ser claro, conciso y profesional, explicando cómo se han implementado las dependencias jerárquicas y cómo las transformaciones (traslación, rotación, escalado) se componen para lograr el movimiento articulado deseado.

Capítulo 5

Ejercicios adicionales

5.1 Práctica 1

Ejercicio 5.1.1. Pirámide con base en ”L”.

Solución 5.1.1. Para resolver este ejercicio, se debe construir una malla 3D indexada que combine una base en forma de ”L” con una pirámide de 6 caras que converge en un único vértice (ápice).

La malla debe estar compuesta por un total de 7 vértices: 6 para la base en ”L” y 1 para el ápice. Los vértices de la base se sitúan en el plano $Y = 0$, mientras que el ápice se coloca sobre la esquina interior de la ”L”, a una altura determinada.

La base se triangula utilizando el número mínimo de triángulos, en este caso 4, dividiendo la ”L” en dos rectángulos y cada uno en dos triángulos. Los triángulos de la base se definen mediante los siguientes índices de vértices:

- $0, 1, 3$
- $1, 2, 3$
- $0, 3, 5$
- $3, 4, 5$

Las caras laterales de la pirámide se forman conectando el ápice con cada uno de los lados de la base, resultando en 6 triángulos adicionales:

- $6, 0, 1$
- $6, 1, 2$
- $6, 2, 3$
- $6, 3, 4$
- $6, 4, 5$
- $6, 5, 0$

En Godot, se recomienda crear un nodo `MeshInstance3D` y asignarle un script donde se definan los vértices y los índices de los triángulos. El siguiente ejemplo ilustra cómo implementar la malla en GDScript:

```
extends MeshInstance3D

func _ready():
    var vertices = PoolVector3Array([
        Vector3(0, 0, 0), # v0
        Vector3(2, 0, 0), # v1
        Vector3(2, 0, 1), # v2
        Vector3(1, 0, 1), # v3
```

```
        Vector3(1, 0, 2), # v4
        Vector3(0, 0, 2), # v5
        Vector3(1, 2, 1)  # v6 (ápice)
    ])

    var indices = PoolIntArray([
        0, 1, 3,
        1, 2, 3,
        0, 3, 5,
        3, 4, 5,
        6, 0, 1,
        6, 1, 2,
        6, 2, 3,
        6, 3, 4,
        6, 4, 5,
        6, 5, 0
    ])

    var arrays = []
    arrays.resize(ArrayMesh.ARRAY_MAX)
    arrays[ArrayMesh.ARRAY_VERTEX] = vertices
    arrays[ArrayMesh.ARRAY_INDEX] = indices

    var mesh = ArrayMesh.new()
    mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
    arrays)
    self.mesh = mesh
```

Al ejecutar la escena, se visualizará la pirámide con base en "L" y 6 caras laterales, cumpliendo con los requisitos del ejercicio.

Ejercicio 5.1.2. Polígono regular en el plano XY.

Solución 5.1.2. Este ejercicio consiste en construir una malla 2D en el plano XY con forma de polígono regular de n lados. El procedimiento es:

1) **Configuración del nodo:** Crear un nodo `MeshInstance3D` y asignarle un script, por ejemplo `PoligonoRegular.gd`.
2) **Definición de parámetros:** El número de lados n es editable desde el inspector.
3) **Generación de vértices:** Se define un vértice central y n vértices exteriores, distribuidos uniformemente en círculo alrededor del centro.
4) **Triangulación:** Se generan n triángulos, cada uno formado por el centro y dos vértices exteriores consecutivos.
5) **Material:** Se asigna un material sin sombreado, usando los colores de los vértices.

El siguiente código en GDScript implementa la malla:

```
extends MeshInstance3D
```

```
2
3 @export var n: int = 8
4
5 func _ready():
6     if n < 3:
7         n = 3
8         print("El polígono debe tener al menos 3 lados. Usando n
    =3.")
9     var new_mesh = poligono_regular(n)
10    self.mesh = new_mesh
11    var material = StandardMaterial3D.new()
12    material.vertex_color_use_as_albedo = true
13    material.shading_mode = StandardMaterial3D.
    SHADING_MODE_UNSHADED
14    self.material_override = material
15
16 func poligono_regular(p_n: int) -> ArrayMesh:
17    var vertices = PackedVector3Array()
18    var colors = PackedColorArray()
19    var indices = PackedInt32Array()
20    var centro = Vector3(0.5, 0.5, 0)
21    vertices.push_back(centro)
22    colors.push_back(Color.WHITE)
23    var radio = 0.5
24    for i in range(p_n):
25        var angulo = (float(i) / float(p_n)) * TAU
26        var x = centro.x + radio * cos(angulo)
27        var y = centro.y + radio * sin(angulo)
28        var z = 0.0
29        vertices.push_back(Vector3(x, y, z))
30        colors.push_back(Color(x, y, z))
31    for i in range(p_n):
32        var idx_centro = 0
33        var idx_v1 = i + 1
34        var idx_v2 = (i + 1) % p_n + 1
35        indices.push_back(idx_centro)
36        indices.push_back(idx_v1)
37        indices.push_back(idx_v2)
38    var arrays = []
39    arrays.resize(ArrayMesh.ARRAY_MAX)
40    arrays[ArrayMesh.ARRAY_VERTEX] = vertices
41    arrays[ArrayMesh.ARRAY_COLOR] = colors
42    arrays[ArrayMesh.ARRAY_INDEX] = indices
43    var mesh = ArrayMesh.new()
44    mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
    arrays)
45    return mesh
```

Explicación: El vértice central se sitúa en $(0{,}5, 0{,}5, 0)$. Los vértices exteriores se distribuyen uniformemente en círculo. Cada triángulo conecta el centro con dos vértices consecutivos, formando así el polígono regular. El material se configura para mostrar los colores de los vértices y sin sombreado.

Ejercicio 5.1.3. Estrella plana en el plano $Z = 0$ alternando radios.

Solución 5.1.3. Este ejercicio consiste en construir una malla 2D en el plano $Z = 0$ con forma de estrella de n puntas, alternando dos radios distintos para generar las puntas y los valles de la estrella.

El procedimiento es el siguiente:

1) **Configuración del nodo:** Se crea un nodo `MeshInstance3D` y se le asigna un script, por ejemplo `EstrellaZ.gd`.
2) **Definición de parámetros:** El número de puntas n es editable. Para una estrella clásica, $n = 5$.
3) **Generación de vértices:** Se define un vértice central y $2n$ vértices exteriores, alternando entre el radio de las puntas y el de los valles. El vértice central se sitúa en $(0{,}5, 0{,}5, 0)$ y los exteriores se calculan usando ángulos equiespaciados en la circunferencia.
4) **Triangulación:** Se generan $2n$ triángulos, cada uno formado por el centro y dos vértices exteriores consecutivos.
5) **Material:** Se asigna un material sin sombreado y con doble cara, usando los colores de los vértices.

A continuación se muestra el código en GDScript que implementa la malla:

```
extends MeshInstance3D

@export var n: int = 5

func _ready():
    if n < 2:
        n = 2
        print("La estrella debe tener al menos 2 puntas. Usando n=2.")
    var new_mesh = ArrayMeshEstrellaZ(n)
    self.mesh = new_mesh
    var material = StandardMaterial3D.new()
    material.vertex_color_use_as_albedo = true
    material.shading_mode = StandardMaterial3D.SHADING_MODE_UNSHADED
    material.cull_mode = StandardMaterial3D.CULL_DISABLED
    self.material_override = material

func ArrayMeshEstrellaZ(p_n: int) -> ArrayMesh:
    var vertices = PackedVector3Array()
    var colors = PackedColorArray()
    var indices = PackedInt32Array()
    var centro = Vector3(0.5, 0.5, 0)
```

```
22      vertices.push_back(centro)
23      colors.push_back(Color.WHITE)
24      var radio_punta = 0.5
25      var radio_valle = 0.25
26      var num_vertices_externos = 2 * p_n
27      for i in range(num_vertices_externos):
28          var radio_actual = radio_punta if i % 2 == 0 else
    radio_valle
29          var angulo = (float(i) / float(num_vertices_externos)) *
     TAU
30          var x = centro.x + radio_actual * cos(angulo)
31          var y = centro.y + radio_actual * sin(angulo)
32          var z = 0.0
33          vertices.push_back(Vector3(x, y, z))
34          colors.push_back(Color(x, y, z))
35      for i in range(num_vertices_externos):
36          var idx_centro = 0
37          var idx_v1 = i + 1
38          var idx_v2 = (i + 1) % num_vertices_externos + 1
39          indices.push_back(idx_centro)
40          indices.push_back(idx_v1)
41          indices.push_back(idx_v2)
42      var arrays = []
43      arrays.resize(ArrayMesh.ARRAY_MAX)
44      arrays[ArrayMesh.ARRAY_VERTEX] = vertices
45      arrays[ArrayMesh.ARRAY_COLOR] = colors
46      arrays[ArrayMesh.ARRAY_INDEX] = indices
47      var mesh = ArrayMesh.new()
48      mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
    arrays)
49      return mesh
```

Explicación: El vértice central se sitúa en el centro de la figura. Los vértices exteriores alternan entre dos radios para formar las puntas y valles de la estrella. Cada triángulo conecta el centro con dos vértices exteriores consecutivos, formando así la estrella completa. El material se configura para que la malla sea visible por ambas caras y utilice los colores de los vértices.

5.2 Práctica 2

Ejercicio 5.2.1. Casa alargada en el eje X con tejado a dos aguas.

Solución 5.2.1. Se trata de construir una casa alargada, formada por un prisma rectangular sin base ni tapa, y un tejado a dos aguas. Se definen 10 vértices: 8 para las esquinas de las paredes y 2 para la arista superior del tejado. Las paredes laterales se forman con 8 triángulos (2 por cada cara), y el tejado con 6 triángulos (2 gabletes y 4 faldones). Los colores de los vértices se asignan según su posición, y se utiliza un material sin sombreado. El siguiente código en GDScript implementa la malla:

```
extends MeshInstance3D

func _ready():
    var vertices = PackedVector3Array([
        Vector3(0.0, 0.0, 0.0), # v0
        Vector3(1.0, 0.0, 0.0), # v1
        Vector3(1.0, 0.0, 0.5), # v2
        Vector3(0.0, 0.0, 0.5), # v3
        Vector3(0.0, 0.5, 0.0), # v4
        Vector3(1.0, 0.5, 0.0), # v5
        Vector3(1.0, 0.5, 0.5), # v6
        Vector3(0.0, 0.5, 0.5), # v7
        Vector3(0.0, 1.0, 0.25), # v8
        Vector3(1.0, 1.0, 0.25)  # v9
    ])

    var colors = PackedColorArray()
    for v in vertices:
        colors.push_back(Color(v.x, v.y, v.z))

    var indices = PackedInt32Array([
        0, 1, 5,   0, 5, 4,
        2, 3, 7,   2, 7, 6,
        1, 2, 6,   1, 6, 5,
        3, 0, 4,   3, 4, 7,
        4, 7, 8,
        5, 9, 6,
        4, 5, 9,   4, 9, 8,
        7, 6, 9,   7, 9, 8
    ])

    var arrays = []
    arrays.resize(ArrayMesh.ARRAY_MAX)
    arrays[ArrayMesh.ARRAY_VERTEX] = vertices
    arrays[ArrayMesh.ARRAY_INDEX] = indices
    arrays[ArrayMesh.ARRAY_COLOR] = colors

    var new_mesh = ArrayMesh.new()
    new_mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
    arrays)
    self.mesh = new_mesh

    var material = StandardMaterial3D.new()
    material.vertex_color_use_as_albedo = true
    material.shading_mode = StandardMaterial3D.
    SHADING_MODE_UNSHADED
    material.cull_mode = StandardMaterial3D.CULL_DISABLED
    self.material_override = material
```

La malla resultante representa una casa alargada con tejado a dos aguas, sin base ni tapa, y con colores interpolados según la posición de los vértices.

Ejercicio 5.2.2. Pirámide con base de estrella.

Solución 5.2.2. El objetivo es construir una pirámide cuya base es una estrella plana de n puntas (como en el ejercicio anterior), y cuyas caras laterales convergen en un ápice centrado sobre la base. Se definen $2n + 2$ vértices: uno central, $2n$ en el borde de la estrella (alternando radios), y uno para el ápice. Se generan $2n$ triángulos para la base y $2n$ triángulos laterales para las caras de la pirámide. El siguiente código en GDScript implementa la malla para $n = 5$:

```
extends MeshInstance3D

func _ready():
    const n: int = 5
    if n <= 1:
        push_error("n debe ser mayor que 1")
        return

    var vertices = PackedVector3Array()
    var colors = PackedColorArray()
    var indices = PackedInt32Array()

    var centro = Vector3(0.5, 0.5, 0)
    vertices.push_back(centro)
    colors.push_back(Color.WHITE)

    var radio_punta = 0.5
    var radio_valle = 0.25
    var num_vertices_externos = 2 * n

    for i in range(num_vertices_externos):
        var radio_actual = radio_punta if i % 2 == 0 else
   radio_valle
        var angulo = (float(i) / float(num_vertices_externos)) *
    TAU
        var x = centro.x + radio_actual * cos(angulo)
        var y = centro.y + radio_actual * sin(angulo)
        var z = 0.0
        vertices.push_back(Vector3(x, y, z))
        colors.push_back(Color(x, y, z))

    vertices.push_back(Vector3(0.5, 0.5, 0.5))
    colors.push_back(Color.WHITE)
    var idx_apex = vertices.size() - 1
```

```
33
34      for i in range(num_vertices_externos):
35          var idx_centro = 0
36          var idx_v1 = i + 1
37          var idx_v2 = (i + 1) % num_vertices_externos + 1
38          indices.push_back(idx_centro)
39          indices.push_back(idx_v1)
40          indices.push_back(idx_v2)
41
42      for i in range(num_vertices_externos):
43          var idx_v1 = i + 1
44          var idx_v2 = (i + 1) % num_vertices_externos + 1
45          indices.push_back(idx_apex)
46          indices.push_back(idx_v1)
47          indices.push_back(idx_v2)
48
49      var arrays = []
50      arrays.resize(ArrayMesh.ARRAY_MAX)
51      arrays[ArrayMesh.ARRAY_VERTEX] = vertices
52      arrays[ArrayMesh.ARRAY_COLOR] = colors
53      arrays[ArrayMesh.ARRAY_INDEX] = indices
54
55      var new_mesh = ArrayMesh.new()
56      new_mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
    arrays)
57      self.mesh = new_mesh
58
59      var material = StandardMaterial3D.new()
60      material.vertex_color_use_as_albedo = true
61      material.shading_mode = StandardMaterial3D.
    SHADING_MODE_UNSHADED
62      material.cull_mode = StandardMaterial3D.CULL_DISABLED
63      self.material_override = material
```

La malla resultante es una pirámide con base de estrella y colores interpolados, cumpliendo con los requisitos de vértices y triángulos.

Ejercicio 5.2.3. Rejilla perpendicular al eje Y.

Solución 5.2.3. El objetivo es construir una malla indexada que represente una rejilla plana perpendicular al eje Y, es decir, situada en el plano $Y = 0$ y ocupando el cuadrado $[0, 1] \times [0, 1]$ en los ejes X y Z. La rejilla está formada por $m \times n$ vértices y $(m - 1) \times (n - 1)$ celdas, cada una compuesta por dos triángulos.

Cada vértice tiene color RGB igual a sus coordenadas X, Y, Z. La malla resultante tiene $m \times n$ vértices y $2(m - 1)(n - 1)$ triángulos.

A continuación se muestra el código en GDScript para el nodo `RejillaY`:

```
extends MeshInstance3D

func _ready():
    var new_mesh = ArrayMeshRejilla(10, 10)
    self.mesh = new_mesh
    var material = StandardMaterial3D.new()
    material.vertex_color_use_as_albedo = true
    material.shading_mode = StandardMaterial3D.
    SHADING_MODE_UNSHADED
    material.cull_mode = StandardMaterial3D.CULL_DISABLED
    self.material_override = material

func ArrayMeshRejilla(m: int, n: int) -> ArrayMesh:
    var vertices = PackedVector3Array()
    var colors = PackedColorArray()
    var indices = PackedInt32Array()
    # Generar vértices y colores
    for i in range(m):
        for j in range(n):
            var x = float(i) / float(m - 1)
            var y = 0.0
            var z = float(j) / float(n - 1)
            vertices.push_back(Vector3(x, y, z))
            colors.push_back(Color(x, y, z))
    # Generar índices de triángulos
    for i in range(m - 1):
        for j in range(n - 1):
            var v0 = i * n + j
            var v1 = (i + 1) * n + j
            var v2 = i * n + (j + 1)
            var v3 = (i + 1) * n + (j + 1)
            # Triángulo 1
            indices.push_back(v0)
            indices.push_back(v2)
            indices.push_back(v3)
            # Triángulo 2
            indices.push_back(v0)
            indices.push_back(v3)
            indices.push_back(v1)
    var arrays = []
    arrays.resize(ArrayMesh.ARRAY_MAX)
    arrays[ArrayMesh.ARRAY_VERTEX] = vertices
    arrays[ArrayMesh.ARRAY_COLOR] = colors
    arrays[ArrayMesh.ARRAY_INDEX] = indices
    var mesh = ArrayMesh.new()
    mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
    arrays)
    return mesh
```

Explicación: Se generan $m \times n$ vértices en el plano $Y = 0$, con coordenadas X y Z equiespaciadas entre 0 y 1. Cada celda de la rejilla se triangula en dos triángulos usando los índices de los vértices. El color de cada vértice es igual a sus coordenadas. El material se configura para mostrar los colores de los vértices y sin sombreado.

Ejercicio 5.2.4. Torre de planta cuadrada.

Solución 5.2.4. La tarea es construir una torre hueca de planta cuadrada y altura n, formada por n secciones apiladas. Cada sección tiene 4 vértices en la base y 4 en la parte superior, resultando en $4(n+1)$ vértices en total. Las caras laterales se forman con triángulos que conectan los vértices de las secciones adyacentes.

Explicación Detallada

El objetivo es crear una torre hueca apilando n secciones de paredes cuadradas, una encima de la otra. Se generan $n+1$.ªnillos"de vértices, cada uno con 4 vértices formando un cuadrado de lado 1, situados en alturas $Y = 0, 1, \ldots, n$. Así, hay $4(n+1)$ vértices en total.

Para las caras, cada sección conecta dos anillos consecutivos. Cada cara lateral se forma con 2 triángulos, y hay 4 caras por sección, resultando en $8n$ triángulos en total. Los índices se calculan usando aritmética modular para cerrar la figura correctamente.

Código en GDScript (Torre.gd)

```
extends MeshInstance3D

func _ready():
    const n: int = 5
    if n < 1:
        push_error("n debe ser 1 o mayor")
        return

    var vertices = PackedVector3Array()
    var indices = PackedInt32Array()

    # Generar vértices
    for i in range(n + 1):
        var y = float(i)
        vertices.push_back(Vector3(0.0, y, 0.0))
        vertices.push_back(Vector3(1.0, y, 0.0))
        vertices.push_back(Vector3(1.0, y, 1.0))
        vertices.push_back(Vector3(0.0, y, 1.0))

    # Generar triángulos
    for i in range(n):
        for j in range(4):
```

```
            var idx0 = i * 4 + j
            var idx1 = i * 4 + (j + 1) % 4
            var idx2 = (i + 1) * 4 + j
            var idx3 = (i + 1) * 4 + (j + 1) % 4

            indices.push_back(idx0)
            indices.push_back(idx1)
            indices.push_back(idx3)

            indices.push_back(idx0)
            indices.push_back(idx3)
            indices.push_back(idx2)

    var arrays = []
    arrays.resize(ArrayMesh.ARRAY_MAX)
    arrays[ArrayMesh.ARRAY_VERTEX] = vertices
    arrays[ArrayMesh.ARRAY_INDEX] = indices

    var new_mesh = ArrayMesh.new()
    new_mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
    arrays)
    self.mesh = new_mesh

    var material = StandardMaterial3D.new()
    material.albedo = Color.WHITE
    material.shading_mode = StandardMaterial3D.
    SHADING_MODE_UNSHADED
    self.material_override = material
```

Resumen: Se construye una torre hueca de planta cuadrada y altura n, formada por $4(n+1)$ vértices y $8n$ triángulos, con material sin sombreado.

5.3 Práctica 3

Ejercicio 5.3.1. Grafo de escena: estrella y conos (GrafoEstrellaX).

Solución 5.3.1. Este ejercicio consiste en construir un grafo de escena jerárquico en Godot, donde el nodo raíz (`GrafoEstrellaX`) contiene una estrella plana en el plano XZ y n conos, uno en cada punta de la estrella. Los conos comparten malla y material, y se orientan hacia fuera desde el centro. Además, todo el grafo puede rotar sobre el eje X mediante animación.

Puntos clave:

- La estrella se genera con una función que crea la malla en el plano XZ.
- Los conos se generan con revolución de un perfil y se colocan/orientan en las puntas.
- Se usa un único material y malla para todos los conos.
- La animación rota el nodo raíz, haciendo girar todo el conjunto.

Código GDScript:

```
extends Node3D

const VELOCIDAD_GIRO = 2.5 * TAU
var activar := "activar_giro_grafoEstrellaX"
var activa := true

func _ready():
    const n: int = 5
    if n <= 1:
        push_error("n debe ser > 1")
        return

    var star_mesh = ArrayMeshEstrellaZ(n)
    var star_node = MeshInstance3D.new()
    star_node.mesh = star_mesh
    var star_material = StandardMaterial3D.new()
    star_material.vertex_color_use_as_albedo = true
    star_material.shading_mode = StandardMaterial3D.
    SHADING_MODE_UNSHADED
    star_material.cull_mode = StandardMaterial3D.CULL_DISABLED
    star_node.material_override = star_material
    add_child(star_node)

    var cone_mesh = crear_mesh_cono()
    var cone_material = StandardMaterial3D.new()
    cone_material.albedo = Color.WHITE
    cone_material.shading_mode = StandardMaterial3D.
    SHADING_MODE_UNSHADED

    var centro = Vector3.ZERO
    var radio_punta = 0.5
    var num_vertices_totales_estrella = 2 * n

    for i in range(n):
        var angulo_punta = (float(i * 2) / float(
    num_vertices_totales_estrella)) * TAU
        var x = centro.x + radio_punta * cos(angulo_punta)
        var y = 0.0
        var z = centro.z + radio_punta * sin(angulo_punta)
        var pos_punta = Vector3(x, y, z)
        var cone_node = MeshInstance3D.new()
        cone_node.mesh = cone_mesh
        cone_node.material_override = cone_material
        cone_node.position = pos_punta
        var dir_original = Vector3.UP
        var dir_deseada = (pos_punta - centro).normalized()
        var rotation_axis = dir_original.cross(dir_deseada).
    normalized()
```

```
45          var rotation_angle = dir_original.angle_to(dir_deseada)
46          cone_node.rotate(rotation_axis, rotation_angle)
47          add_child(cone_node)
48
49  func _process(delta):
50      if Input.is_action_just_pressed(activar):
51          activa = !activa
52      if activa:
53          rotate_x(VELOCIDAD_GIRO * delta)
54
55  func crear_mesh_cono() -> ArrayMesh:
56      var perfil_cono = PackedVector2Array([
57          Vector2(0.0, 0.15),
58          Vector2(0.14, 0.0),
59          Vector2(0.0, 0.0)
60      ])
61      var vertices = PackedVector3Array()
62      var triangulos = PackedInt32Array()
63      RevolucionUtils.generar_malla_revolucion(perfil_cono, 16,
    vertices, triangulos)
64      var normales = Utilidades.calcNormales(vertices, triangulos)
65      var arrays = []
66      arrays.resize(ArrayMesh.ARRAY_MAX)
67      arrays[Mesh.ARRAY_VERTEX] = vertices
68      arrays[Mesh.ARRAY_INDEX] = triangulos
69      arrays[Mesh.ARRAY_NORMAL] = normales
70      var new_mesh = ArrayMesh.new()
71      new_mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
    arrays)
72      return new_mesh
73
74  func ArrayMeshEstrellaZ(p_n: int) -> ArrayMesh:
75      var vertices = PackedVector3Array()
76      var colors = PackedColorArray()
77      var indices = PackedInt32Array()
78      var centro = Vector3.ZERO
79      vertices.push_back(centro)
80      colors.push_back(Color.WHITE)
81      var radio_punta = 0.5
82      var radio_valle = 0.25
83      var num_vertices_externos = 2 * p_n
84      for i in range(num_vertices_externos):
85          var radio_actual = radio_punta if i % 2 == 0 else
    radio_valle
86          var angulo = (float(i) / float(num_vertices_externos)) *
     TAU
87          var x = centro.x + radio_actual * cos(angulo)
88          var y = 0.0
```

```
89          var z = centro.z + radio_actual * sin(angulo)
90          vertices.push_back(Vector3(x, y, z))
91          colors.push_back(Color(x, y, z))
92      for i in range(num_vertices_externos):
93          var idx_centro = 0
94          var idx_v1 = i + 1
95          var idx_v2 = (i + 1) % num_vertices_externos + 1
96          indices.push_back(idx_centro)
97          indices.push_back(idx_v1)
98          indices.push_back(idx_v2)
99      var arrays = []
100     arrays.resize(ArrayMesh.ARRAY_MAX)
101     arrays[Mesh.ARRAY_VERTEX] = vertices
102     arrays[Mesh.ARRAY_COLOR] = colors
103     arrays[Mesh.ARRAY_INDEX] = indices
104     var new_mesh = ArrayMesh.new()
105     new_mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
        arrays)
106     return new_mesh
```

Ejercicio 5.3.2. Grafo de escena: cubo central y cubos pequeños (GrafoCubos).

Solución 5.3.2. Se pide construir un grafo de escena donde un cubo central grande se forma con 6 rejillas y, en el centro de cada cara, hay un cubo pequeño alargado. Los cubos pequeños rotan alrededor del eje que pasa por su centro y el origen, usando pivotes. Todo el sistema tiene un único grado de libertad de animación.

Puntos clave:

- El cubo central se construye con 6 mallas de rejilla, cada una orientada y posicionada en una cara.
- Los cubos pequeños se crean una vez y se colocan como hijos de nodos pivote, que se rotan para animar.
- Se usan materiales y mallas compartidos.
- La animación rota los pivotes de los cubos pequeños.

Código GDScript:

```
1  extends Node3D
2
3  var angulo_giro := 0.0
4  const VELOCIDAD_GIRO = TAU
5  var activar_accion := "activar_giro_cubos"
6  var activa := true
7
8  var pivot_x_pos: Node3D
9  var pivot_x_neg: Node3D
10 var pivot_y_pos: Node3D
```

```
var pivot_y_neg: Node3D
var pivot_z_pos: Node3D
var pivot_z_neg: Node3D

func _ready():
    var rejilla_mesh = ArrayMeshRejilla(11, 11)
    var cubo_mesh = ArrayMeshCubo24()
    var rejilla_material = StandardMaterial3D.new()
    rejilla_material.vertex_color_use_as_albedo = true
    rejilla_material.shading_mode = StandardMaterial3D.SHADING_MODE_UNSHADED
    rejilla_material.cull_mode = StandardMaterial3D.CULL_DISABLED
    var cubo_material = StandardMaterial3D.new()
    cubo_material.albedo = Color.WHITE
    cubo_material.shading_mode = StandardMaterial3D.SHADING_MODE_UNSHADED

    var cubo_central = Node3D.new()
    add_child(cubo_central)
    var centro_rejilla = Vector3(-0.5, 0, -0.5)
    var transforms = [
        Transform3D(Basis(), Vector3(0, 0.5, 0)) * Transform3D(Basis(), centro_rejilla),
        Transform3D(Basis.from_euler(Vector3(PI, 0, 0)), Vector3(0, -0.5, 0)) * Transform3D(Basis(), centro_rejilla),
        Transform3D(Basis.from_euler(Vector3(0, 0, PI/2.0)), Vector3(0.5, 0, 0)) * Transform3D(Basis(), centro_rejilla),
        Transform3D(Basis.from_euler(Vector3(0, 0, -PI/2.0)), Vector3(-0.5, 0, 0)) * Transform3D(Basis(), centro_rejilla),
        Transform3D(Basis.from_euler(Vector3(-PI/2.0, 0, 0)), Vector3(0, 0, 0.5)) * Transform3D(Basis(), centro_rejilla),
        Transform3D(Basis.from_euler(Vector3(PI/2.0, 0, 0)), Vector3(0, 0, -0.5)) * Transform3D(Basis(), centro_rejilla)
    ]
    for i in range(6):
        var cara = MeshInstance3D.new()
        cara.mesh = rejilla_mesh
        cara.material_override = rejilla_material
        cara.transform = transforms[i]
        cubo_central.add_child(cara)

    var dist_cubo_peq = 0.7
    var escala_alargada = Vector3(0.2, 0.4, 0.2)

    pivot_y_pos = Node3D.new()
    add_child(pivot_y_pos)
    crear_cubo_pequeno(pivot_y_pos, cubo_mesh, cubo_material,
```

```
    Vector3(0, dist_cubo_peq, 0), escala_alargada)
50
51      pivot_y_neg = Node3D.new()
52      add_child(pivot_y_neg)
53      crear_cubo_pequeno(pivot_y_neg, cubo_mesh, cubo_material,
    Vector3(0, -dist_cubo_peq, 0), escala_alargada)
54
55      pivot_x_pos = Node3D.new()
56      add_child(pivot_x_pos)
57      crear_cubo_pequeno(pivot_x_pos, cubo_mesh, cubo_material,
    Vector3(dist_cubo_peq, 0, 0), escala_alargada.rotated(Vector3
    .FORWARD, PI/2.0))
58
59      pivot_x_neg = Node3D.new()
60      add_child(pivot_x_neg)
61      crear_cubo_pequeno(pivot_x_neg, cubo_mesh, cubo_material,
    Vector3(-dist_cubo_peq, 0, 0), escala_alargada.rotated(
    Vector3.FORWARD, -PI/2.0))
62
63      pivot_z_pos = Node3D.new()
64      add_child(pivot_z_pos)
65      crear_cubo_pequeno(pivot_z_pos, cubo_mesh, cubo_material,
    Vector3(0, 0, dist_cubo_peq), escala_alargada.rotated(Vector3
    .RIGHT, PI/2.0))
66
67      pivot_z_neg = Node3D.new()
68      add_child(pivot_z_neg)
69      crear_cubo_pequeno(pivot_z_neg, cubo_mesh, cubo_material,
    Vector3(0, 0, -dist_cubo_peq), escala_alargada.rotated(
    Vector3.RIGHT, -PI/2.0))
70
71  func _process(delta):
72      if Input.is_action_just_pressed(activar_accion):
73          activa = !activa
74      if not activa:
75          return
76      angulo_giro += VELOCIDAD_GIRO * delta
77      if pivot_y_pos:
78          pivot_y_pos.rotation.y = angulo_giro
79          pivot_y_neg.rotation.y = angulo_giro
80          pivot_x_pos.rotation.x = angulo_giro
81          pivot_x_neg.rotation.x = angulo_giro
82          pivot_z_pos.rotation.z = angulo_giro
83          pivot_z_neg.rotation.z = angulo_giro
84
85  func crear_cubo_pequeno(pivote: Node3D, mesh: ArrayMesh, mat:
    StandardMaterial3D, pos: Vector3, escala: Vector3):
86      var cubo = MeshInstance3D.new()
```

```
87      cubo.mesh = mesh
88      cubo.material_override = mat
89      cubo.position = pos
90      cubo.scale = escala
91      pivote.add_child(cubo)
92
93  func ArrayMeshRejilla(m: int, n: int) -> ArrayMesh:
94      var vertices = PackedVector3Array()
95      var colors = PackedColorArray()
96      var indices = PackedInt32Array()
97      var normales = PackedVector3Array()
98      for i in range(m):
99          for j in range(n):
100             var x = float(i) / float(m - 1)
101             var y = 0.0
102             var z = float(j) / float(n - 1)
103             vertices.push_back(Vector3(x, y, z))
104             colors.push_back(Color(x, y, z))
105             normales.push_back(Vector3.UP)
106     for i in range(m - 1):
107         for j in range(n - 1):
108             var v0 = i * n + j
109             var v1 = (i + 1) * n + j
110             var v2 = i * n + (j + 1)
111             var v3 = (i + 1) * n + (j + 1)
112             indices.push_back(v0)
113             indices.push_back(v2)
114             indices.push_back(v3)
115             indices.push_back(v0)
116             indices.push_back(v3)
117             indices.push_back(v1)
118     var arrays = []
119     arrays.resize(ArrayMesh.ARRAY_MAX)
120     arrays[Mesh.ARRAY_VERTEX] = vertices
121     arrays[Mesh.ARRAY_COLOR] = colors
122     arrays[Mesh.ARRAY_INDEX] = indices
123     arrays[Mesh.ARRAY_NORMAL] = normales
124     var mesh = ArrayMesh.new()
125     mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
        arrays)
126     return mesh
127
128 func ArrayMeshCubo24() -> ArrayMesh:
129     var vertices := PackedVector3Array([
130         Vector3(-0.5,  0.5,  0.5), Vector3( 0.5,  0.5,  0.5),
        Vector3( 0.5, -0.5,  0.5), Vector3(-0.5, -0.5,  0.5),
131         Vector3( 0.5,  0.5, -0.5), Vector3(-0.5,  0.5, -0.5),
        Vector3(-0.5, -0.5, -0.5), Vector3( 0.5, -0.5, -0.5),
```

```
132         Vector3( 0.5,  0.5,  0.5), Vector3( 0.5,  0.5, -0.5),
    Vector3( 0.5, -0.5, -0.5), Vector3( 0.5, -0.5,  0.5),
133         Vector3(-0.5,  0.5, -0.5), Vector3(-0.5,  0.5,  0.5),
    Vector3(-0.5, -0.5,  0.5), Vector3(-0.5, -0.5, -0.5),
134         Vector3(-0.5,  0.5, -0.5), Vector3( 0.5,  0.5, -0.5),
    Vector3( 0.5,  0.5,  0.5), Vector3(-0.5,  0.5,  0.5),
135         Vector3(-0.5, -0.5,  0.5), Vector3( 0.5, -0.5,  0.5),
    Vector3( 0.5, -0.5, -0.5), Vector3(-0.5, -0.5, -0.5)
136     ])
137     var triangulos := PackedInt32Array([
138         0, 1, 2,  0, 2, 3,    4, 5, 6,  4, 6, 7,
139         8, 9, 10, 8, 10, 11,   12, 13, 14, 12, 14, 15,
140         16, 17, 18, 16, 18, 19, 20, 21, 22, 20, 22, 23
141     ])
142     var normales := Utilidades.calcNormales(vertices, triangulos
    )
143     var arrays := []
144     arrays.resize(ArrayMesh.ARRAY_MAX)
145     arrays[Mesh.ARRAY_VERTEX] = vertices
146     arrays[Mesh.ARRAY_INDEX] = triangulos
147     arrays[Mesh.ARRAY_NORMAL] = normales
148     var new_mesh := ArrayMesh.new()
149     new_mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
    arrays)
150     return new_mesh
```

Parte III

Ejercicios

Nota: Se adjunta un índice para buscar más fácil el contenido.

I. Matemáticas y Demostraciones Vectoriales

Fundamentos teóricos sobre operaciones con vectores, matrices y transformaciones.

II. Implementación en GDScript (Godot)

Scripts para generación de geometría, jerarquías de escena y lógica de control.

A. Geometría Procedural y Mallas

- **1.1.1 Polígono Regular Relleno**
  Creación de una malla de N lados mediante `MeshInstance2D`.

- **1.1.2 Gradientes de Color**
  Uso de *Vertex Colors* para interpolación de colores en la malla.

- **1.1.4 Visualización de Aristas (Wireframe)**
  Diferencias de implementación entre mallas indexadas y no indexadas.

- **1.1.5 Debug de Normales**
  Script global (autoload) para generar líneas que visualicen las normales de una malla.

- **1.2.12 Función Gancho**
  Generación de una polilínea simple mediante código.

- **1.4.1 Figuras Compuestas**
  Script para generar un cuadrado azul con un triángulo inscrito y bordes diferenciados.

- **1.4.3 Triangulación Manual (Tronco)**
  Generación de un polígono cóncavo mediante descomposición en triángulos.

- **1.4.5 Modelado por Código (Logo Android)**
  Construcción 3D usando primitivas cilíndricas y semiesféricas.

B. Escena, Jerarquías y Animación

- **1.2.13 Instanciación y Pivotes**
  Rotaciones complejas alrededor de un punto de pivote desplazado.

- **1.4.2 Transformaciones Jerárquicas**
  Uso de nodos padre/hijo con escalado negativo (efecto espejo).

- **1.4.4 Árbol Fractal Recursivo**
  Script recursivo para generar ramas transformadas geométricamente.

- **1.8.1 Gestión de Input**
  Lógica para detectar la duración exacta de la pulsación de una tecla.

- **1.10.1 Curvas de Hermite**
  Interpolación suave de movimiento pasando por puntos de control con tangentes.

- **1.10.2 Oscilación Controlada**
  Movimiento periódico con velocidad constante y rebote exacto en extremos.

- **1.10.3 Reloj Analógico**
  Rotación de agujas sincronizada con el tiempo del sistema (`Time`).

- **1.10.4 Simulación de Péndulo**
  Animación basada en funciones armónicas (sin/cos) para oscilación física.

- **1.10.5 Tiro Parabólico**
  Animación física basada en la ecuación $p = p_0 + v_0 t + 0{,}5at^2$.

III. Algoritmos y Pseudocódigo (Ray Tracing)

Diseño lógico para cálculo de intersecciones y selección (Picking).

- **1.8.2 Intersección Rayo-Triángulo**
  Algoritmo completo: Intersección con plano + Coordenadas Baricéntricas.

- **1.8.3 Picking (Unproject)**
  Cálculo del rayo 3D en coordenadas de mundo a partir de un click en pantalla 2D.

- **1.9.1 Intersección Rayo-Disco**
  Lógica de intersección plano-rayo y verificación de distancia al centro (radio).

- **1.9.2 Intersección Rayo-Esfera**
  Resolución mediante ecuación cuadrática para esferas unitarias y genéricas.

- **1.9.3 Intersección Rayo-Cilindro/Cono**
  Algoritmos para cuádricas infinitas con *clipping* por altura finita.

- **1.3.5 Extracción de Aristas**
  Algoritmo para generar una tabla de aristas únicas desde una lista de triángulos.

- **1.3.6 Cálculo de Área**
  Algoritmo para sumar las áreas de los triángulos de una malla (producto cruz).

IV. Teoría de Mallas y Texturas

Eficiencia espacial, topología y mapeo de coordenadas UV.

- **1.3.1 Eficiencia de Memoria**
  Comparativa: Enumeración Espacial (Vóxeles $O(k^3)$) vs Malla Indexada ($O(k^2)$).

- **1.3.2 Rejilla Rectangular**
  Cálculo de memoria requerida para una topología de rejilla $M \times N$.

- **1.3.3 Triangle Strips**
  Análisis coste-beneficio: Ahorro de memoria vs coste de Vertex Shader.

- **1.3.4 Topología (Euler-Poincaré)**
  Demostración de relaciones en mallas cerradas: $N_A = 3(N_V - 2)$.

- **1.7.1 Mapeo UV (Dado)**
  Diseño de tabla de vértices mínima (14 vértices) para textura continua.

- **1.7.2 Normales y Costuras (Hard Edges)**
  Justificación de duplicado de vértices (24) para iluminación en cubo.

- **1.7.3 Textura Repetida (Tiling)**
  Tabla de coordenadas UV para repetir una imagen en todas las caras.

V. Cámara, Proyección e Iluminación

Matemáticas de la cámara virtual y modelos de reflexión de luz.

A. Configuración de Cámara

- **1.5.1 Cámara de Seguimiento**
  Script para posicionar la cámara detrás y arriba de un objetivo móvil.
- **1.5.2 LookAt (Ejes Alineados)**
  Cálculo de vectores a, u, n para una configuración ortogonal específica.
- **1.5.3 LookAt (Con Rotación)**
  Cálculo de vectores de cámara incluyendo rotación sobre el eje de vista (*Roll*).
- **1.5.4 Base de la Cámara**
  Código para derivar la base ortonormal (u, v, n) desde parámetros de vista.
- **1.5.5 Matriz de Vista**
  Construcción manual de la `Transform3D` (inversa de la cámara).
- **1.5.6 Control de Aspect Ratio**
  Script para mantener el FOV fijo (75º) independientemente del tamaño de ventana.

B. Proyección y Frustum

- **1.5.7 Frustum Ajustado (Cubo)**
  Cálculo de planos (n, f, l, r, t, b) para encuadrar perfectamente un cubo.
- **1.5.8 Frustum Ajustado (Esfera)**
  Ajuste de planos de proyección para encuadrar una esfera tangente.
- **1.5.9 Frustum No Cuadrado**
  Adaptación de la proyección para relaciones de aspecto *Landscape* y *Portrait*.
- **1.5.10 Posicionamiento por FOV**
  Cálculo de la distancia de la cámara dado un ángulo de apertura β.

C. Modelos de Iluminación

- **1.6.1 Especularidad**
  Implementación de las fórmulas de Phong y Blinn-Phong en GDScript.
- **1.6.2 Puntos de Brillo Máximo**
  Cálculo teórico de la posición del brillo en una esfera (Lambert/Phong).

– **1.6.3 BRDF GGX (Microfacetas)**
Implementación completa: Fresnel Schlick + Geometría + Distribución Normal.

Capítulo 1

Ejercicios Teóricos

Observación. Estos ejercicios usan una numeración distinta a la de las diapositivas, aunque están en el mismo orden. Hay veces que cuandos se hace referencia a un ejercicio se usa la enumeración de las diapositivas para encontrarlo más fácilmente. De la misma manera se recomienda revisar el orden de defición de vértices y demás para triángulos por si es el correcto.

1.1 Sesión 2

Para la resolución de los siguientes ejercicios se ha usado varios scripts como autoload:

```
# Script de Godot para asignar a un nodo de tipo 'Node2D'
# de forma que se visualizan los ejes, con fondo blanco, la
# vista 2D es controlable con el ratón.
# Fija la vista para que inicialmente la región visible incluya a
# un cuadrado de lado 2 y centro en el origen ( [-1,-1] .. [+1,+1] )

extends Node2D

# -------------------------------
# tamaños del viewport guardados

var viewport_tamano_dcc_int  : Vector2i = Vector2i( 0, 0 ) # tamaño actual del area de dibujo en pixels (dcc)
var viewport_tamano_dcc      : Vector2 = Vector2( 0, 0 )

var vp : Viewport = null

# -------------------------------
# Definición de la vista (transf. de vista, desde WCC a DC)

const ejeX := Vector2( 1.0, 0.0 )
const ejeY := Vector2( 0.0, 1.0 )

var tp  : float = 1.0 # tamaño (== alto, ancho) de un pixel en coords de mundo (WCC)
var cvp : Vector2 = Vector2( 0, 0 ) # centro del viewport en coordenadas de mundo (WCC)
```

```
25 var fev : float = 1.0 # factor de escalado de la vista, se
       controla con la rueda del ratón
26
27 # --------------------------
28 # Estado de arrastre del ratón
29
30 var raton_izq_pulsado : bool = false
31
32 # -------------------------------
33 # Actualiza la transformación de vista en función del tamaño del
       área de dibujo
34
35 var c : int = 0
36
37 func _actualiza_transf_vista(  ) -> void :
38
39   tp = 2.0/(fev*min( vp.size.x, vp.size.y ))
40   var t1 := Transform2D( ejeX, ejeY, -cvp )
41   var t2 := Transform2D( ejeX/tp, -ejeY/tp, cvp+0.5*vp.size )
42   transform = t2*t1
43
44 # --------------------------
45 # Procesar evento de entrada.
46 # Se usa por ahora únicamente para controlar la vista 2d con el
       ratón
47
48 func _unhandled_input( event : InputEvent ):
49
50   var actualizada : bool = false
51
52   if event is InputEventMouseButton:
53     if event.button_index == MOUSE_BUTTON_LEFT:
54       raton_izq_pulsado = event.pressed
55
56     if event.button_index == MOUSE_BUTTON_WHEEL_UP:
57       fev *= 1.05
58       actualizada = true
59
60     if event.button_index == MOUSE_BUTTON_WHEEL_DOWN:
61       fev /= 1.05
62       actualizada = true
63
64   elif event is InputEventMouseMotion and raton_izq_pulsado:
65     cvp += tp * Vector2( -event.relative.x, +event.relative.y )
66     actualizada = true
67
68   elif event is InputEventKey:
69     if event.keycode == KEY_ESCAPE and event.is_released():
```

```
70        get_tree().quit()
71
72    if actualizada:
73      _actualiza_transf_vista(   )
74
75  # -----------------------------------------------
76  # crear objetos para los ejes y añadirlos como hijos
77
78  func _crear_ejes_2d() :
79
80    const w2 : float = 0.01  # mitad del ancho de la barra (en X)
81    const f  : float = 2.5  # ancho de la flecha en X relativo al
       ancho de la barra
82    const l : float = 0.9   # longitud de la flecha en Y (entre 0
       y 1), resto hasta 1 es el triangulo.
83
84    # crear tablas para una barra (indexado) y un triángulo
85    var t_barra : Array = [] ; t_barra.resize( Mesh.ARRAY_MAX )
86    t_barra[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([
87      Vector2(-w2,-w2/2.0), Vector2(w2,-w2/2.0), Vector2(-w2,l),
       Vector2(w2,l)
88    ])
89    t_barra[ Mesh.ARRAY_INDEX ] = PackedInt32Array([ 0,1,2, 2,1,3
       ])
90
91    var t_tri : Array = [] ; t_tri.resize( Mesh.ARRAY_MAX )
92    t_tri[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([
93      Vector2(-w2*f,l), Vector2(w2*f,l), Vector2(0,1)
94    ])
95
96    # crear un arrayMesh y añadir las dos surfaces
97    var am := ArrayMesh.new()
98    am.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, t_barra
       )
99    am.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES, t_tri )
100
101   # crear mesh instances, para ej X (rojo) y para eje Y (verde)
102   var ex := MeshInstance2D.new(); ex.mesh = am; ex.rotate( -PI
       /2.0 )
103   var ey := MeshInstance2D.new(); ey.mesh = am
104   ex.modulate = Color(1,0,0)
105   ey.modulate = Color(0,1,0)
106
107   # crear lineas en el eje X y en el Y
108   var lex := Line2D.new()
109   lex.points = PackedVector2Array([ Vector2(-1000,0), Vector2
       (1000,0) ])
110   lex.width = w2*0.75
```

```
111   lex.default_color = Color(1,0,0)
112   lex.antialiased = true
113
114   var ley := Line2D.new()
115   ley.points = PackedVector2Array([ Vector2(0,-1000), Vector2
        (0,1000) ])
116   ley.width = w2*0.75
117   ley.default_color = Color(0,1,0)
118   ley.antialiased = true
119
120   # añadir eje X e Y a un nuevo nodo 2D
121   var ejes := Node2D.new()
122   ejes.add_child( ex ) ; ejes.add_child( lex )
123   ejes.add_child( ey ) ; ejes.add_child( ley )
124   ejes.z_index = RenderingServer.CANVAS_ITEM_Z_MIN  # ponerlo
        detrás de todo
125
126   # poner ejes como hijo de este:
127   add_child( ejes )
128
129
130 # inicialización
131
132 func _ready() -> void:
133
134   _crear_ejes_2d()
135   RenderingServer.set_default_clear_color( Color( 1.0, 1.0, 1.0
        ))
136
137   # actualizar el viewport ('vp') y la vista.
138   vp = get_viewport() ; assert( vp is Viewport )
139   _actualiza_transf_vista()
140   vp.connect( "size_changed", _actualiza_transf_vista )
```

```
1 extends Node
2
3 func genSegNormales( verts, norms : PackedVector3Array, lon :
      float, color : Color ) -> MeshInstance3D:
4   var line_verts = PackedVector3Array()
5   var line_colors = PackedColorArray()
6
7   for i in range(verts.size()):
8     var origen = verts[i]
9     var destino = origen + norms[i] * lon
10
11    line_verts.push_back(origen)
```

```
12      line_verts.push_back(destino)
13
14      line_colors.push_back(color)
15      line_colors.push_back(color)
16
17    var arrays = []
18    arrays.resize(Mesh.ARRAY_MAX)
19    arrays[Mesh.ARRAY_VERTEX] = line_verts
20    arrays[Mesh.ARRAY_COLOR] = line_colors
21
22    var arr_mesh = ArrayMesh.new()
23    arr_mesh.add_surface_from_arrays(Mesh.PRIMITIVE_LINES, arrays)
24
25    var material = StandardMaterial3D.new()
26    material.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADED
27    material.vertex_color_use_as_albedo = true
28
29    var mi = MeshInstance3D.new()
30    mi.mesh = arr_mesh
31    mi.material_override = material
32
33    return mi
34
35  func gancho() -> ArrayMesh:
36    var vertices = PackedVector2Array([
37      Vector2(0,0),
38      Vector2(1,0),
39      Vector2(1,1),
40      Vector2(0,1),
41      Vector2(0,2)
42    ])
43
44    var arrays = []
45    arrays.resize(Mesh.ARRAY_MAX)
46    arrays[Mesh.ARRAY_VERTEX] = vertices
47
48    var mesh = ArrayMesh.new()
49    mesh.add_surface_from_arrays(Mesh.PRIMITIVE_LINE_STRIP, arrays
       )
50    return mesh
```

```
1  extends Node
2  y
3  # Transformación identidad auxiliar para facilitar la lectura
       del código
4  var tr_identidad := Transform2D()
```

```
5
6  # Variables para almacenar las mallas generadas
7  var cuadrado: ArrayMesh
8  var triangulo: ArrayMesh
9  var casa: ArrayMesh
10 var circunferencia: ArrayMesh
11
12
13 # ============================
14 # FUNCIÓN: _inicializar_meshes
15 # Propósito: Construye las geometrías básicas (primitivas) que
       se reutilizarán.
16 # Se ejecuta una sola vez al inicio para optimizar memoria.
17 # ============================
18 func _inicializar_meshes():
19   # Definición del Cuadrado (Unitario)
20   cuadrado = CrearArrayMesh(PackedVector2Array([
21     Vector2(0.0, 0.0), Vector2(1.0, 0.0),
22     Vector2(1.0, 1.0), Vector2(0.0, 1.0),
23     Vector2(0.0, 0.0)
24   ]))
25
26   # Definición del Triángulo
27   triangulo = CrearArrayMesh(PackedVector2Array([
28     Vector2(0.0, 0.0), Vector2(1.0, 0.0), Vector2(0.0, 1.0),
29     Vector2(0.0, 0.0)
30   ]))
31
32   # Definición de la Casa (Polígono irregular)
33   casa = CrearArrayMesh(PackedVector2Array([
34     Vector2(0.0, 0.0), Vector2(1.0, 0.0),
35     Vector2(1.0, 1.0), Vector2(0.5, 1.4), Vector2(0.0, 1.0),
36     Vector2(0.0, 0.0)
37   ]))
38
39   # Definición de la Circunferencia (64 segmentos)
40   circunferencia = CrearCircunferencia(64)
41
42
43 # ============================
44 # FUNCIONES AUXILIARES: GENERACIÓN DE GEOMETRÍA
45 # ============================
46
47 # ============================
48 # Función: CrearArrayMesh
49 # Descripción: Crea un objeto ArrayMesh a partir de un array de
       vectores.
50 # Usa la primitiva PRIMITIVE_LINE_STRIP (polilínea abierta).
```

```
# Parámetros: v (PackedVector2Array) - Lista de vértices.
# Retorno: ArrayMesh configurado.
# ============================
func CrearArrayMesh(v: PackedVector2Array) -> ArrayMesh:
  var tablas: Array = []
  tablas.resize(Mesh.ARRAY_MAX)
  tablas[Mesh.ARRAY_VERTEX] = v
  var am: ArrayMesh = ArrayMesh.new()
  am.add_surface_from_arrays(Mesh.PRIMITIVE_LINE_STRIP, tablas)
  return am

# ============================
# Función: CrearCircunferencia
# Descripción: Genera una malla circular aproximada por segmentos de línea.
# Parámetros: n (int) - Número de segmentos (resolución).
# Retorno: ArrayMesh con la forma de la circunferencia.
# ============================
func CrearCircunferencia(n: int) -> ArrayMesh:
  var v := PackedVector2Array()
  for i in range(n + 1):
    var a: float = (float(i) * 2.0 * PI) / float(n)
    v.append(Vector2(cos(a), sin(a)))
  return CrearArrayMesh(v)


# ============================
# FUNCIONES AUXILIARES: INSTANCIACIÓN Y TRANSFORMACIÓN DE NODOS
# ============================

# ============================
# Función: CrearMeshInstance2D
# Descripción: Crea un nodo visual (MeshInstance2D) asignándole una malla y una
# transformación inicial. Asigna un color azul por defecto (modulate).
# Parámetros:
#   - am: La malla (ArrayMesh) a visualizar.
#   - tr: La transformación (Transform2D) a aplicar.
# Retorno: MeshInstance2D listo para añadir al árbol.
# ============================
func CrearMeshInstance2D(am: ArrayMesh, tr: Transform2D) -> MeshInstance2D:
  var mi := MeshInstance2D.new()
  mi.transform = tr
  mi.modulate = Color(0.0, 0.0, 0.7) ## Azul estándar
  mi.mesh = am
  return mi
```

```
95
96  # ============================
97  # Función: TransformaNode2D
98  # Descripción: Aplica una transformación adicional a un nodo
        existente.
99  # Realiza una composición por la izquierda (tr * n.transform).
100 # Parámetros:
101 #   - n: El nodo a transformar.
102 #   - tr: La matriz de transformación a aplicar.
103 # Retorno: El mismo nodo 'n' modificado.
104 # ============================
105 func TransformaNode2D(n: Node2D, tr: Transform2D) -> Node2D:
106   n.transform = tr * n.transform
107   return n
108
109
110 # ============================
111 # CONSTRUCCIÓN DEL MODELO JERÁRQUICO (ÁRBOL 2D)
112 # Funciones que construyen las partes compuestas del objeto "
        Casa".
113 # ============================
114
115 # ============================
116 # Componente: Pomo
117 # Descripción: Crea el pomo de la puerta usando la
        circunferencia escalada y trasladada.
118 # ============================
119 func Pomo() -> Node2D:
120   var tra = Transform2D().translated(Vector2(0.1, 0.5))
121   var esc = Transform2D().scaled(Vector2(0.06, 0.06))
122   return CrearMeshInstance2D(circunferencia, tra * esc)
123
124 # ============================
125 # Componente: HojaDer (Hoja Derecha)
126 # Descripción: Crea una hoja de puerta compuesta por un cuadrado
        y un pomo.
127 # ============================
128 func HojaDer() -> Node2D:
129   var tra = Transform2D().translated(Vector2(0.5, 0.0))
130   var esc = Transform2D().scaled(Vector2(0.5, 1.0))
131   var n = Node2D.new()
132   # Añade el pomo transformado
133   n.add_child(TransformaNode2D(Pomo(), tra))
134   # Añade la base de la hoja (cuadrado transformado)
135   n.add_child(CrearMeshInstance2D(cuadrado, tra * esc))
136   return n
137
138 # ============================
```

```
# Componente: Puerta
# Descripción: Crea una puerta doble compuesta por una hoja normal y otra reflejada.
# ============================
func Puerta() -> Node2D:
  var tra = Transform2D().translated(Vector2(1.0, 0.0))
  var esc = Transform2D().scaled(Vector2(-1.0, 1.0)) # Escalado negativo para reflejar
  var n = Node2D.new()
  n.add_child(HojaDer()) # Hoja derecha original
  n.add_child(TransformaNode2D(HojaDer(), tra * esc)) # Hoja izquierda (reflejada)
  return n

# ============================
# Componente: MarcoIzq (Marco Izquierdo)
# Descripción: Parte decorativa de la ventana, compuesta por un cuadrado y un triángulo.
# ============================
func MarcoIzq() -> Node2D:
  var tra = Transform2D().translated(Vector2(0.0, 1.0))
  var esc = Transform2D().scaled(Vector2(0.9, -0.8))
  var n = Node2D.new()
  n.add_child(CrearMeshInstance2D(cuadrado, tr_identidad))
  n.add_child(CrearMeshInstance2D(triangulo, tra * esc))
  return n

# ============================
# Componente: Marcos
# Descripción: Conjunto de marcos para la ventana (izquierdo y derecho reflejado).
# ============================
func Marcos() -> Node2D:
  var esc1 = Transform2D().scaled(Vector2(0.8, 1.0))
  var tra = Transform2D().translated(Vector2(2.4, 0.0))
  var esc2 = Transform2D().scaled(Vector2(-1.0, 1.0))
  var n = Node2D.new()
  n.add_child(TransformaNode2D(MarcoIzq(), esc1))
  n.add_child(TransformaNode2D(MarcoIzq(), esc1 * tra * esc2))
  return n

# ============================
# Componente: Ventana
# Descripción: Crea una ventana completa con fondo (cuadrado) y marcos interiores.
# ============================
func Ventana() -> Node2D:
  var tra = Transform2D().translated(Vector2(0.1, 0.1))
```

```
181   var esc = Transform2D().scaled(Vector2(0.4, 0.8))
182   var n = Node2D.new()
183   n.add_child(CrearMeshInstance2D(cuadrado, tr_identidad))
184   n.add_child(TransformaNode2D(Marcos(), tra * esc))
185   return n
186 
187 # ============================
188 # Componente: InstPuerta (Instancia de Puerta)
189 # Descripción: Instancia la puerta completa en su posición final
      relativa a la fachada.
190 # ============================
191 func InstPuerta() -> Node2D:
192   var tra = Transform2D().translated(Vector2(0.56, 0.0))
193   var esc = Transform2D().scaled(Vector2(0.3, 0.43))
194   var n = Node2D.new()
195   n.add_child(TransformaNode2D(Puerta(), tra * esc))
196   return n
197 
198 # ============================
199 # Componente: InstVentana (Instancia de Ventana)
200 # Descripción: Prepara una instancia de ventana con una escala
    base.
201 # ============================
202 func InstVentana() -> Node2D:
203   var esc = Transform2D().scaled(Vector2(0.3, 0.3))
204   var n = Node2D.new()
205   n.add_child(TransformaNode2D(Ventana(), esc))
206   return n
207 
208 # ============================
209 # Componente Principal: Fachada
210 # Descripción: Raíz del modelo jerárquico. Ensambla la
    estructura de la casa,
211 # la puerta y múltiples instancias de ventanas en posiciones
    específicas.
212 # ============================
213 func Fachada() -> Node2D:
214   # Definición de transformaciones para posicionar elementos
215   var tra1 = Transform2D().translated(Vector2(0.13, 0.13))
216   var tra2 = Transform2D().translated(Vector2(0.00, 0.43))
217   var tra3 = Transform2D().translated(Vector2(0.43, 0.00))
218 
219   var n = Node2D.new()
220 
221   # 1. Estructura base de la casa
222   n.add_child(CrearMeshInstance2D(casa, tr_identidad))
223 
224   # 2. Puerta principal
```

```
225   n.add_child(TransformaNode2D(InstPuerta(), tr_identidad))
226
227   # 3. Ventana inferior izquierda
228   n.add_child(TransformaNode2D(InstVentana(), tra1))
229
230   # 4. Ventana superior izquierda (tra1 + tra2)
231   n.add_child(TransformaNode2D(InstVentana(), tra1 * tra2))
232
233   # 5. Ventana superior derecha (tra1 + tra2 + tra3)
234   n.add_child(TransformaNode2D(InstVentana(), tra1 * tra2 * tra3
      ))
235
236   return n
237
238
239 # ============================
240 # FUNCIONES EXTRA PARA EJERCICIOS ESPECÍFICOS
241 # ============================
242
243 # ============================
244 # Helper Local para Rellenos (PRIMITIVE_TRIANGLES)
245 # Necesario porque 'funcionesauxiliarest5.CrearArrayMesh' usa
      LINE_STRIP.
246 # ============================
247
248 func _crear_malla_rellena(v: PackedVector2Array) -> ArrayMesh:
249   var tablas: Array = []
250   tablas.resize(Mesh.ARRAY_MAX)
251   tablas[Mesh.ARRAY_VERTEX] = v
252   var am: ArrayMesh = ArrayMesh.new()
253   # Aquí usamos TRIANGLES en lugar de LINE_STRIP
254   am.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES, tablas)
255   return am
256
257 # ============================
258 # Helper Local para Líneas por pares (PRIMITIVE_LINES)
259 # Necesario para dibujar líneas discontinuas o saltando vértices
      .
260 # ============================
261 func _crear_malla_lineas_pares(v: PackedVector2Array) ->
      ArrayMesh:
262   var tablas: Array = []
263   tablas.resize(Mesh.ARRAY_MAX)
264   tablas[Mesh.ARRAY_VERTEX] = v
265   var am: ArrayMesh = ArrayMesh.new()
266   # Usamos LINES en lugar de LINE_STRIP
267   am.add_surface_from_arrays(Mesh.PRIMITIVE_LINES, tablas)
268   return am
```

Puede ser que se usasen de otros ficheros, pero estos son los principales. De todas formas, si no se incluye la implementación de algún método se sobreentiende.

Ejercicio 1.1.1. Polígono regular relleno de color plano

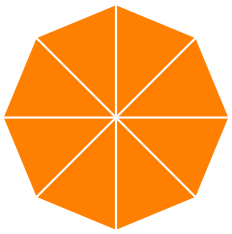
Implementa un nodo de tipo `MeshInstance2D` con una malla (no indexada) para un polígono regular de n lados relleno de color naranja plano (RGB(1.0, 0.7, 0.0)), con radio r y centro en el origen.

El polígono estará formado por n triángulos, cada uno con un vértice en el centro y los otros dos en el contorno.

Los valores de n y r se declaran como dos constantes de GDScript (`const`), como se indica a continuación:

```
const n: int = 8
const r: float = 0.8
```

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



Solución 1.1.1. Solución al problema 2.1:

```
# Problema 2.1:
# Implementa un nodo de tipo MeshInstance con
# una malla (no indexada) para un polígono regular
# de n lados relleno de color naranja plano (RGB
# (1.0, 0.7, 0.0)), con radio r y centro en el origen (ver
# figura).
# El polígono estará formado por n triángulos, cada uno
# con un vértice en el centro y los otros dos en el
# contorno. Los valores de n y r de declaran como dos
# constantes de GDScript (const), como se indica aquí:
# const n : int = 8
# const r : float = 0.8
# Los valores de estas constantes se podrán cambiar sin
# tocar nada del resto del script.

extends MeshInstance2D # <-- IMPORTANTE

const n: int = 8
const r: float = 0.8

func _ready():
  var vertices = PackedVector2Array()

```

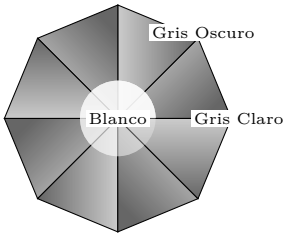
```
24  var center = Vector2(0, 0)
25
26  var angle_step = TAU / n
27
28  for i in range(n):
29    var angle1 = i * angle_step
30    var angle2 = (i + 1) * angle_step
31
32    var p1 = Vector2(r * cos(angle1), r * sin(angle1))
33    var p2 = Vector2(r * cos(angle2), r * sin(angle2))
34
35    vertices.push_back(center)
36    vertices.push_back(p1)
37    vertices.push_back(p2)
38
39  var tablas = []
40
41  tablas.resize(Mesh.ARRAY_MAX)
42  tablas[Mesh.ARRAY_VERTEX] = vertices
43
44  mesh = ArrayMesh.new()
45  mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES, tablas)
46
47  modulate = Color(1.0, 0.7, 0.0)
```

Ejercicio 1.1.2. Polígono regular relleno con gradaciones

Crea otro `Node2D`, y asígnale un script para visualizar el mismo polígono regular que antes (también con una malla no indexada), solo que ahora debes asignar colores a los vértices para que los triángulos aparezcan con una graduación en tonos de gris como en la figura.

Cada triángulo que forma el polígono regular será blanco en el vértice del centro, gris claro en otro vértice del borde y gris oscuro en el tercero.

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



Solución 1.1.2. Solución al problema 2.2:

```
# Problema 2.2:
# Crea otro Node2D, y asígnale un script para
# visualizar el mismo polígono regular que antes
```

```
# (también con una malla no indexada), solo que
# ahora debes asignar colores a los vértices para
# que los triángulos aparezcan con una gradua7
# ción en tonos de gris como en la figura. Cada
# triángulo que forma el polígono regular será
# blanco en el vértice del centro, gris claro en
# otro y gris oscuro en el tercero.
# Responde razonadamente a esta cuestión:
# ¿ cuantos vértices debe tener la tabla de vér7
# tices ?

extends MeshInstance2D

const n: int = 8
const r: float = 0.8

func _ready():
    # 1. Creamos arrays para vértices Y para colores
    var vertices = PackedVector2Array()
    var colors = PackedColorArray()

    var center_pos = Vector2(0, 0)
    var angle_step = TAU / n

    # Definimos los colores según pide el enunciado [cite: 2422]
    var c_center = Color(1.0, 1.0, 1.0) # Blanco (Centro)
    var c_light = Color(0.8, 0.8, 0.8)  # Gris Claro (Vértice 1)
    var c_dark = Color(0.2, 0.2, 0.2)   # Gris Oscuro (Vértice
    2)

    for i in range(n):
        var angle1 = i * angle_step
        var angle2 = (i + 1) * angle_step

        # Calculamos posiciones del borde
        var p1 = Vector2(r * cos(angle1), r * sin(angle1))
        var p2 = Vector2(r * cos(angle2), r * sin(angle2))

        # --- AÑADIMOS VÉRTICES (Topología Triángulos) ---
        vertices.push_back(center_pos)
        vertices.push_back(p1)
        vertices.push_back(p2)

        # --- AÑADIMOS COLORES (En el mismo orden estricto) ---
        colors.push_back(c_center)
        colors.push_back(c_light)
        colors.push_back(c_dark)

```

```
    # 2. Preparamos las tablas (SOA - Structure of Arrays)
    var tablas = []
    tablas.resize(Mesh.ARRAY_MAX)
    tablas[Mesh.ARRAY_VERTEX] = ver

    # 3. Generamos la malla
    mesh = ArrayMesh.new()
    mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
    tablas)

    # NOTA: Ya no usamos 'modulate' porque los colores vienen
    dentro de la malla.
```

Para ver cuantos vértices tiene la tabla de vértices, hay que tener en cuenta que cada triángulo tiene 3 vértices, y como hay n triángulos, la tabla de vértices debe tener $3n$ vértices. Por lo tanto, la respuesta es $3n$. Siendon $n = 8$, la tabla de vértices tiene 24 vértices.

Ejercicio 1.1.3. Repite los dos problemas anteriores (2.1 y 2.2), con los mismos requerimientos, pero ahora usando **mallas indexadas**, de forma que el número de vértices e índices sea mínimo.

Responde razonadamente a estas cuestiones:

- ¿Cuántos vértices debe tener ahora la tabla de vértices en cada caso?
- ¿Y cuántos índices debe haber?

Solución 1.1.3. Solución al problema 2.3:

```
extends MeshInstance2D
const n: int = 8
const r: float = 0.8
const activate_2_1: bool = true

func _ready():
    if activate_2_1:
        var vertices = PackedVector2Array()
        var indices = PackedInt32Array()

        var center = Vector2(0, 0)
        var angle_step = TAU / n

        # --- 1. GENERACIÓN DE VÉRTICES (TABLA DE VÉRTICES) ---
        # Añadimos el centro (Índice 0)
        vertices.push_back(center)

        # Añadimos los puntos del perímetro (Índices 1 a n)
        for i in range(n):
            var angle = i * angle_step
            var p = Vector2(cos(angle), sin(angle)) * r
```

```
            vertices.push_back(p)

        # --- 2. GENERACIÓN DE ÍNDICES (TABLA DE TRIÁNGULOS) ---
        # Conectamos los vértices ya existentes
        for i in range(n):
            # El centro siempre es el índice 0
            var idx_center = 0

            # Vértice actual del perímetro (empiezan en el índice 1)
            var idx_current = i + 1

            # Siguiente vértice. Usamos módulo (%) para cerrar el círculo
            # (Si estamos en el último, el siguiente debe ser el 1)
            var idx_next = (i + 1) % n + 1

            # Definimos el triángulo
            indices.push_back(idx_center)
            indices.push_back(idx_current)
            indices.push_back(idx_next)

        # --- 3. CREACIÓN DE LA MALLA ---
        var tablas = []
        tablas.resize(Mesh.ARRAY_MAX)
        tablas[Mesh.ARRAY_VERTEX] = vertices
        tablas[Mesh.ARRAY_INDEX] = indices # ¡Ahora usamos índices!

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

        # Color plano para toda la malla
        modulate = Color(1.0, 0.7, 0.0)
    else:
        var vertices = PackedVector2Array()
        var colors = PackedColorArray()
        var indices = PackedInt32Array()

        var angle_step = TAU / n

        # Colores definidos
        var c_center = Color(1.0, 1.0, 1.0) # Blanco
        var c_light = Color(0.8, 0.8, 0.8)  # Gris Claro (Inicio arco)
        var c_dark = Color(0.2, 0.2, 0.2)   # Gris Oscuro (Fin
```

```
    arco)
64
65          # --- 1. VÉRTICES Y COLORES ---
66
67          # Vértice 0: El Centro (Compartido por todos)
68          vertices.push_back(Vector2(0,0))
69          colors.push_back(c_center)
70
71          # Vértices del perímetro (No se pueden compartir entre
    triángulos vecinos)
72          for i in range(n):
73              var angle1 = i * angle_step
74              var angle2 = (i + 1) * angle_step
75
76              var p1 = Vector2(cos(angle1), sin(angle1)) * r
77              var p2 = Vector2(cos(angle2), sin(angle2)) * r
78
79              # Para cada triángulo, añadimos sus dos vértices del
     borde específicos
80              vertices.push_back(p1) # Vértice 'Light' de este tri
    ángulo
81              colors.push_back(c_light)
82
83              vertices.push_back(p2) # Vértice 'Dark' de este trié
    ngulo
84              colors.push_back(c_dark)
85
86          # --- 2. ÍNDICES ---
87          for i in range(n):
88              # El centro siempre es 0
89              # Los vértices del perímetro están agrupados de 2 en
     2 a partir del índice 1
90              # Triángulo 0 usa índices: 1 y 2
91              # Triángulo 1 usa índices: 3 y 4...
92              var base_idx = 1 + (i * 2)
93
94              indices.push_back(0)        # Centro
95              indices.push_back(base_idx) # Light
96              indices.push_back(base_idx + 1) # Dark
97
98          # --- 3. MALLA ---
99          var tablas = []
100         tablas.resize(Mesh.ARRAY_MAX)
101         tablas[Mesh.ARRAY_VERTEX] = vertices
102         tablas[Mesh.ARRAY_COLOR] = colors
103         tablas[Mesh.ARRAY_INDEX] = indices
104
105         mesh = ArrayMesh.new()
```

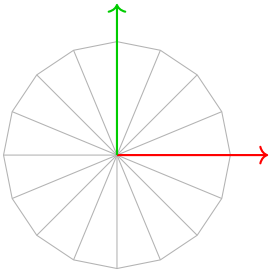
```
106         mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
    tablas)
```

Ejercicio 1.1.4. Aristas del polígono regular

Crea un nuevo nodo `MeshInstance2D` de forma que ahora veamos simplemente las aristas del contorno del polígono regular descrito en los anteriores problemas. En la figura se ve el resultado para $n = 16$ y el mismo radio.

Considera dos casos:

1) Usando una malla **no indexada**.
2) Usando una malla **indexada**.



Solución 1.1.4. Solución al problema 2.4:

```
1  extends MeshInstance2D
2
3  # Problema 2.4:
4  # Crea un nuevo nodo MeshInstance2D de for7
5  # ma que ahora veamos simplemente las aristas
6  # del polígono regular descrito en los anteriores
7  # problemas. En la figura se ve para n a 16 y el
8  # mismo radio.
9  # Considera dos casos:
10 # - Usando una malla no indexada.
11 # - Usando una malla indexadas.
12
13 var no_indexado: bool = false
14 const n: int = 16 # Actualizado a 16 como pide el enunciado
15 const r: float = 0.8
16 var centre: Vector2 = Vector2(0, 0)
17
18 func _ready():
19   if no_indexado:
20     var vertices = PackedVector2Array()
21     var angle_step = TAU / n # TAU es 2*PI
22
23     # Generamos pares de vértices para cada línea
24     for i in range(n):
```

```
25        var angle_current = i * angle_step
26        var angle_next = (i + 1) * angle_step
27
28        # Calculamos los dos extremos del segmento actual
29        var p1 = Vector2(cos(angle_current), sin(angle_current)) *
      r
30        var p2 = Vector2(cos(angle_next), sin(angle_next)) * r
31
32        # Añadimos ambos al array.
33        # Como NO es indexada, repetimos vértices geométricos.
34        vertices.push_back(centre)
35        vertices.push_back(p1)
36
37        vertices.push_back(p1)
38        vertices.push_back(p2)
39
40
41      # Preparamos la estructura SOA
42      var tablas = []
43      tablas.resize(Mesh.ARRAY_MAX)
44      tablas[Mesh.ARRAY_VERTEX] = vertices
45
46      mesh = ArrayMesh.new()
47      # IMPORTANTE: Cambiamos el tipo de primitiva a LÍNEAS
48      mesh.add_surface_from_arrays(Mesh.PRIMITIVE_LINES, tablas)
49
50      # Color para las líneas (ej. Verde o Rojo)
51      modulate = Color(0.0, 1.0, 0.0)
52    else:
53      var vertices = PackedVector2Array()
54      var indices = PackedInt32Array()
55      var angle_step = TAU / n
56
57      # 1. TABLA DE VÉRTICES
58      # Primero añadimos el CENTRO (Índice 0)
59      vertices.push_back(centre)
60
61      # Luego los puntos del perímetro (Índices 1 a n)
62      for i in range(n):
63        var angle = i * angle_step
64        var p = Vector2(cos(angle), sin(angle)) * r
65        vertices.push_back(p)
66
67      # 2. TABLA DE ÍNDICES
68      for i in range(n):
69        # El centro es el índice 0
70        var idx_center = 0
71        # Vértice actual del borde (offset +1 porque el 0 es el
```

```
    centro)
72      var idx_current = i + 1
73      # Siguiente vértice (con módulo para cerrar el círculo)
74      var idx_next = (i + 1) % n + 1
75
76      # --- DEFINIMOS LAS LÍNEAS ---
77
78      # Línea 1: Radio (Conecta Centro con Actual)
79      indices.push_back(idx_center)
80      indices.push_back(idx_current)
81
82      # Línea 2: Borde (Conecta Actual con Siguiente)
83      indices.push_back(idx_current)
84      indices.push_back(idx_next)
85
86    # 3. CREACIÓN DE LA MALLA
87    var tablas = []
88    tablas.resize(Mesh.ARRAY_MAX)
89    tablas[Mesh.ARRAY_VERTEX] = vertices
90    tablas[Mesh.ARRAY_INDEX] = indices # ¡Asignamos los índices!
91
92    mesh = ArrayMesh.new()
93    mesh.add_surface_from_arrays(Mesh.PRIMITIVE_LINES, tablas)
94
95    modulate = Color(1.0, 0.0, 0.0) # Rojo
```

Ejercicio 1.1.5. Generación de malla con segmentos de normales

Crea un script global (autoload) con una función que genere un objeto de tipo `MeshInstance3D` con una malla no indexada que contenga los segmentos representando las normales de una malla dada. La función tendrá la siguiente declaración:

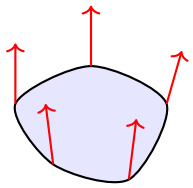
```
func genSegNormales(
    verts: PackedVector3Array,
    norms: PackedVector3Array,
    lon: float,
    color: Color
) -> MeshInstance3D:
```

Donde `verts` es la tabla de vértices de la malla original, `norms` la tabla de normales, `lon` la longitud de los segmentos y `color` el color de los segmentos. Usa el tipo de primitiva líneas (`PRIMITIVE_LINES`), y asegúrate de que a los segmentos no les afecta la iluminación.

Continuación (Uso): Una vez tengas la función disponible, úsala en la función `_ready` de alguna malla (por ejemplo, el Donut o los cubos de la práctica), para añadir al objeto un nodo hijo con la malla de segmentos creada por la función.

Puedes capturar el evento de pulsación de la tecla **N** del objeto para activar y desactivar la visualización de las normales en ese objeto. Para ello, usa un valor lógico y el atributo de visibilidad

de la malla de segmentos.



Esquema conceptual: Superficie y Normales

Solución 1.1.5. Solución al problema 2.5: Se encuentra en el fichero `Global.gd` del autoload, cargando anteriormente. De todas formas, se añade aquí la función para que se vea más fácilmente:

```
func genSegNormales( verts, norms : PackedVector3Array, lon :
    float, color : Color ) -> MeshInstance3D:
    var line_verts = PackedVector3Array()
    var line_colors = PackedColorArray()

    for i in range(verts.size()):
        var origen = verts[i]
        var destino = origen + norms[i] * lon

        line_verts.push_back(origen)
        line_verts.push_back(destino)

        line_colors.push_back(color)
        line_colors.push_back(color)

    var arrays = []
    arrays.resize(Mesh.ARRAY_MAX)
    arrays[Mesh.ARRAY_VERTEX] = line_verts
    arrays[Mesh.ARRAY_COLOR] = line_colors

    var arr_mesh = ArrayMesh.new()
    arr_mesh.add_surface_from_arrays(Mesh.PRIMITIVE_LINES,
    arrays)

    var material = StandardMaterial3D.new()
    material.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADED
    material.vertex_color_use_as_albedo = true

    var mi = MeshInstance3D.new()
    mi.mesh = arr_mesh
    mi.material_override = material

  return mi

```

1.2 Sesión 3

Ejercicio 1.2.1. Demuestra que efectivamente el producto escalar de dos vectores se puede calcular (usando sus coordenadas en cualquier marco cartesiano) como la suma del producto componente a componente. Usa las propiedades que definen dicho producto escalar.

Solución 1.2.1. **Hipótesis y Datos de Partida:**

1) Definimos dos vectores $\vec{u}$ y $\vec{v}$ en un espacio vectorial V.
2) Trabajamos en un **marco cartesiano**, lo que implica una base ortonormal $\{\hat{e}_i\}$.
3) Las propiedades de esta base especial son:
   - $\hat{e}_i \cdot \hat{e}_i = 1$ (son unitarios).
   - $\hat{e}_i \cdot \hat{e}_j = 0$ si $i \neq j$ (son perpendiculares).
4) Expresamos los vectores mediante sus coordenadas en esta base:

$$\vec{u} = \sum_{i=1}^{n} a_i \hat{e}_i$$
$$\vec{v} = \sum_{j=1}^{n} b_j \hat{e}_j$$

Demostración Paso a Paso:

1) **Planteamiento del producto:**

$$\vec{u} \cdot \vec{v} = \left(\sum_{i=1}^{n} a_i \hat{e}_i \right) \cdot \left(\sum_{j=1}^{n} b_j \hat{e}_j \right)$$

2) **Aplicación de la Propiedad Distributiva:**

$$= \sum_{i=1}^{n} \sum_{j=1}^{n} a_i b_j (\hat{e}_i \cdot \hat{e}_j)$$

3) **Aplicación de la Propiedad Asociativa (Escalares):** Los coeficientes a_i y b_j son reales, así que pueden factorizarse fuera del producto escalar.
4) **Aplicación de las Propiedades de la Base Ortonormal:**
   - Cuando $i \neq j$, el término es 0.
   - Cuando $i = j$, el término es 1.

   Así, sólo sobreviven los términos con $i = j$:

$$\vec{u} \cdot \vec{v} = \sum_{i=1}^{n} a_i b_i$$

Conclusión: Queda demostrado que, en un marco cartesiano, el producto escalar es la suma de los productos de las componentes homólogas.

Ejercicio 1.2.2. Demuestra que el producto vectorial de dos vectores se puede calcular usando sus coordenadas en cualquier marco cartesiano según se ha indicado.

Solución 1.2.2. **Hipótesis y Datos de Partida:**

1) Trabajamos en 3D con la base especial $\{\hat{x}, \hat{y}, \hat{z}\}$.
2) Propiedades definitorias del producto vectorial en esta base:
   - $\hat{x} \times \hat{y} = \hat{z}$, $\hat{y} \times \hat{z} = \hat{x}$, $\hat{z} \times \hat{x} = \hat{y}$ (ciclo dextrógiro).
   - Por la propiedad anticonmutativa, el orden inverso invierte el signo: $\hat{y} \times \hat{x} = -\hat{z}$, etc.
   - El producto de un vector por sí mismo es nulo: $\hat{x} \times \hat{x} = \vec{0}$, etc.
3) Vectores definidos por coordenadas:

$$\vec{u} = x_0\hat{x} + y_0\hat{y} + z_0\hat{z}$$
$$\vec{v} = x_1\hat{x} + y_1\hat{y} + z_1\hat{z}$$

Demostración Paso a Paso:

1) **Planteamiento:**
$$\vec{u} \times \vec{v} = (x_0\hat{x} + y_0\hat{y} + z_0\hat{z}) \times (x_1\hat{x} + y_1\hat{y} + z_1\hat{z})$$

2) **Expansión (Distributiva):**

$$\begin{aligned} &= x_0x_1(\hat{x} \times \hat{x}) + x_0y_1(\hat{x} \times \hat{y}) + x_0z_1(\hat{x} \times \hat{z}) \\ &\quad + y_0x_1(\hat{y} \times \hat{x}) + y_0y_1(\hat{y} \times \hat{y}) + y_0z_1(\hat{y} \times \hat{z}) \\ &\quad + z_0x_1(\hat{z} \times \hat{x}) + z_0y_1(\hat{z} \times \hat{y}) + z_0z_1(\hat{z} \times \hat{z}) \end{aligned}$$

3) **Simplificación con Propiedades de la Base:**

$$\begin{aligned} &= 0 + x_0y_1\hat{z} + x_0z_1(-\hat{y}) \\ &\quad + y_0x_1(-\hat{z}) + 0 + y_0z_1\hat{x} \\ &\quad + z_0x_1\hat{y} + z_0y_1(-\hat{x}) + 0 \end{aligned}$$

4) **Agrupación por componentes:**

$$\begin{aligned} \text{Componente } \hat{x}: &\quad y_0z_1 - z_0y_1 \\ \text{Componente } \hat{y}: &\quad z_0x_1 - x_0z_1 \\ \text{Componente } \hat{z}: &\quad x_0y_1 - y_0x_1 \end{aligned}$$

$$\vec{u} \times \vec{v} = (y_0z_1 - z_0y_1)\hat{x} + (z_0x_1 - x_0z_1)\hat{y} + (x_0y_1 - y_0x_1)\hat{z}$$

Conclusión: El vector resultante en coordenadas es:

$$\vec{u} \times \vec{v} = \begin{pmatrix} y_0z_1 - z_0y_1 \\ z_0x_1 - x_0z_1 \\ x_0y_1 - y_0x_1 \end{pmatrix}$$

Esto coincide exactamente con la definición matricial dada en el documento.

Ejercicio 1.2.3. Demuestra que el producto vectorial de dos vectores es perpendicular a cada uno de esos dos vectores.

Solución 1.2.3. **Datos de Partida:**

1) Condición de perpendicularidad: Dos vectores son perpendiculares si su producto escalar es 0.

2) Usaremos los resultados demostrados en los Problemas 3.1 y 3.2.

Demostración (para $\vec{u}$):

Queremos probar que $\vec{u} \cdot \vec{w} = 0$.

Sea $\vec{w} = \vec{u} \times \vec{v}$. Sus componentes son (del Prob 3.2):

$$w_x = y_0 z_1 - z_0 y_1$$
$$w_y = z_0 x_1 - x_0 z_1$$
$$w_z = x_0 y_1 - y_0 x_1$$

Calculamos el producto escalar $\vec{u} \cdot \vec{w}$ usando la fórmula de componentes (del Prob 3.1):

$$\vec{u} \cdot \vec{w} = x_0 w_x + y_0 w_y + z_0 w_z$$

Sustituyendo los componentes de $\vec{w}$:

$$= x_0(y_0 z_1 - z_0 y_1) + y_0(z_0 x_1 - x_0 z_1) + z_0(x_0 y_1 - y_0 x_1)$$
$$= x_0 y_0 z_1 - x_0 z_0 y_1 + y_0 z_0 x_1 - y_0 x_0 z_1 + z_0 x_0 y_1 - z_0 y_0 x_1$$

Reordenamos los términos para ver las cancelaciones:

- $x_0 y_0 z_1$ se cancela con $-y_0 x_0 z_1$ (son idénticos, el orden de factores reales no altera el producto).
- $-x_0 z_0 y_1$ se cancela con $z_0 x_0 y_1$.
- $y_0 z_0 x_1$ se cancela con $-z_0 y_0 x_1$.

Resultado:

$$\vec{u} \cdot \vec{w} = 0$$

(Nota: La demostración para $\vec{v}$ es análoga, sustituyendo las coordenadas de $\vec{v}$ en el producto escalar, y también resultará en 0).

Conclusión: Hemos demostrado algebraicamente la propiedad geométrica fundamental mencionada en la página 30: el producto vectorial genera una dirección perpendicular al plano formado por los dos vectores originales.

Ejercicio 1.2.4. Demuestra que el producto escalar de vectores en 2D es invariante por rotación. Es decir, que para cualquier ángulo θ y vectores $\vec{u}$ y $\vec{v}$ se cumple:

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

Se requiere realizar la demostración utilizando las coordenadas de los vectores en un marco cartesiano arbitrario.

Solución 1.2.4. Para demostrar la invariancia del producto escalar bajo una transformación de rotación en el espacio euclídeo bidimensional ($\mathbb{R}^2$), procederemos algebraicamente definiendo los componentes de los vectores y la matriz de transformación correspondiente.

Sean $\vec{u}$ y $\vec{v}$ dos vectores libres en $\mathbb{R}^2$ definidos por sus componentes en un marco cartesiano:

$$\vec{u} = \begin{pmatrix} u_x \\ u_y \end{pmatrix}, \quad \vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

La definición estándar del producto escalar (producto punto) en coordenadas cartesianas viene dada por:

$$\vec{u} \cdot \vec{v} = u_x v_x + u_y v_y \tag{1.1}$$

Sea R_θ la transformación de rotación por un ángulo θ alrededor del origen. La matriz asociada a esta transformación en 2D, M_R, se define como:

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

Aplicamos la transformación lineal a los vectores $\vec{u}$ y $\vec{v}$ mediante la multiplicación matricial:

1. Para el vector $\vec{u}' = R_\theta(\vec{u})$:

$$\vec{u}' = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} u_x \\ u_y \end{pmatrix} = \begin{pmatrix} u_x\cos\theta - u_y\sin\theta \\ u_x\sin\theta + u_y\cos\theta \end{pmatrix}$$

Denotamos las componentes transformadas como $u'_x = u_x\cos\theta - u_y\sin\theta$ y $u'_y = u_x\sin\theta + u_y\cos\theta$.

2. Para el vector $\vec{v}' = R_\theta(\vec{v})$:

$$\vec{v}' = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} v_x\cos\theta - v_y\sin\theta \\ v_x\sin\theta + v_y\cos\theta \end{pmatrix}$$

Denotamos las componentes transformadas como $v'_x = v_x\cos\theta - v_y\sin\theta$ y $v'_y = v_x\sin\theta + v_y\cos\theta$.

Procedemos ahora a calcular el producto escalar de los vectores transformados, $R_\theta(\vec{u}) \cdot R_\theta(\vec{v}) = u'_x v'_x + u'_y v'_y$:

$$\begin{aligned} R_\theta(\vec{u}) \cdot R_\theta(\vec{v}) = {} & (u_x\cos\theta - u_y\sin\theta)(v_x\cos\theta - v_y\sin\theta) \\ & + (u_x\sin\theta + u_y\cos\theta)(v_x\sin\theta + v_y\cos\theta) \end{aligned}$$

Expandimos los términos algebraicos:

$$\begin{aligned} R_\theta(\vec{u}) \cdot R_\theta(\vec{v}) = {} & (u_x v_x\cos^2\theta - u_x v_y\cos\theta\sin\theta - u_y v_x\sin\theta\cos\theta + u_y v_y\sin^2\theta) \\ & + (u_x v_x\sin^2\theta + u_x v_y\sin\theta\cos\theta + u_y v_x\cos\theta\sin\theta + u_y v_y\cos^2\theta) \end{aligned}$$

Agrupamos los términos comunes en función de los coeficientes de los vectores originales:

$$\begin{aligned} R_\theta(\vec{u}) \cdot R_\theta(\vec{v}) = u_x v_x(\cos^2\theta + \sin^2\theta) \\ + u_y v_y(\sin^2\theta + \cos^2\theta) \\ + u_x v_y(-\cos\theta\sin\theta + \sin\theta\cos\theta) \\ + u_y v_x(-\sin\theta\cos\theta + \cos\theta\sin\theta) \end{aligned}$$

Aplicamos la identidad trigonométrica fundamental $\cos^2\theta + \sin^2\theta = 1$ y observamos que los términos cruzados se cancelan:

$$\begin{aligned} R_\theta(\vec{u}) \cdot R_\theta(\vec{v}) &= u_x v_x(1) + u_y v_y(1) + u_x v_y(0) + u_y v_x(0) \\ &= u_x v_x + u_y v_y \end{aligned}$$

Comparando este resultado con la definición inicial en la Ecuación (1), concluimos que:

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

Q.E.D.

Ejercicio 1.2.5. Demuestra que en 2D las rotaciones no modifican la longitud de un vector (isometría). Es decir, que para cualquier ángulo θ y vector $\vec{v}$, se cumple:

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

Solución 1.2.5. Para demostrar que la rotación es una transformación isométrica que preserva la norma (longitud) de los vectores, utilizaremos la relación fundamental entre la norma euclídea y el producto escalar.

La definición de la norma de un vector $\vec{v}$ en función del producto escalar es:

$$\|\vec{v}\| = \sqrt{\vec{v} \cdot \vec{v}}$$

Elevando al cuadrado ambos lados, tenemos:

$$\|\vec{v}\|^2 = \vec{v} \cdot \vec{v} \tag{1.2}$$

Consideremos ahora la norma al cuadrado del vector transformado $R_\theta(\vec{v})$:

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

Basándonos en la propiedad demostrada en el Ejercicio 3.4 (invariancia del producto escalar bajo rotación), sabemos que para cualesquiera vectores $\vec{a}$ y $\vec{b}$, se cumple $\vec{a} \cdot \vec{b} = R_\theta(\vec{a}) \cdot R_\theta(\vec{b})$.

En este caso particular, hacemos $\vec{a} = \vec{v}$ y $\vec{b} = \vec{v}$. Aplicando la propiedad de invariancia:

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

Sustituyendo esto en la expresión de la norma transformada:

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

Dado que $\vec{v} \cdot \vec{v} = \|\vec{v}\|^2$ según la Ecuación (2), obtenemos:

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

Tomando la raíz cuadrada positiva en ambos lados (dado que la norma es una magnitud no negativa):

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

Por lo tanto, queda demostrado que la aplicación de una matriz de rotación R_θ no altera la longitud del vector, confirmando que las rotaciones son isometrías.

Ejercicio 1.2.6. Demuestra que si rotamos en 2D un vector +90 grados ($\pi/2$) o -90 grados ($-\pi/2$), obtenemos otro vector perpendicular al original. Es decir, si $\|\theta\| = \pi/2$, entonces:

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

Solución 1.2.6. Para demostrar la perpendicularidad entre un vector original $\vec{v}$ y su versión rotada $\pm 90°$, utilizaremos la definición algebraica del producto escalar y la matriz de rotación específica para estos ángulos.

Sea $\vec{v}$ un vector arbitrario en $\mathbb{R}^2$:

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

La matriz de rotación general R_θ es:

$$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

Analizaremos los dos casos solicitados: $\theta = \pi/2$ y $\theta = -\pi/2$.

Caso 1: Rotación de $+\pi/2$ $(90°)$ Sustituimos $\theta = \pi/2$ en la matriz de rotación, sabiendo que $\cos(\pi/2) = 0$ y $\sin(\pi/2) = 1$:

$$R_{\pi/2} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

Calculamos el vector transformado $\vec{v}' = R_{\pi/2}(\vec{v})$:

$$\vec{v}' = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} -v_y \\ v_x \end{pmatrix}$$

Ahora calculamos el producto escalar entre el vector original y el transformado:

$$\vec{v} \cdot \vec{v}' = v_x(-v_y) + v_y(v_x) = -v_x v_y + v_x v_y = 0$$

Caso 2: Rotación de $-\pi/2$ ($-90°$) Sustituimos $\theta = -\pi/2$ en la matriz, sabiendo que $\cos(-\pi/2) = 0$ y $\sin(-\pi/2) = -1$:

$$R_{-\pi/2} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

Calculamos el vector transformado $\vec{v}'' = R_{-\pi/2}(\vec{v})$:

$$\vec{v}'' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} v_y \\ -v_x \end{pmatrix}$$

Calculamos el producto escalar:

$$\vec{v} \cdot \vec{v}'' = v_x(v_y) + v_y(-v_x) = v_x v_y - v_x v_y = 0$$

Conclusión: En ambos casos, el producto escalar es nulo. Dado que $\vec{a} \cdot \vec{b} = 0 \iff \vec{a} \perp \vec{b}$ (para vectores no nulos), queda demostrado que el vector rotado $\pm 90°$ es perpendicular al original.

Ejercicio 1.2.7. Demuestra que una matriz de rotación en 2D es siempre ortonormal, independientemente del ángulo. Esto implica demostrar que: 1. Sus filas son ortogonales entre sí (perpendiculares). 2. Sus columnas son ortogonales entre sí. 3. Cada fila y cada columna tiene norma (longitud) igual a 1.

Solución 1.2.7. Una matriz M es ortonormal (u ortogonal) si cumple que $M^T M = I$, lo cual equivale a que sus filas y columnas formen una base ortonormal. Analizaremos las propiedades de filas y columnas de la matriz de rotación general.

Sea R_θ la matriz de rotación:

$$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

Denotamos las filas como vectores $\vec{r}_1, \vec{r}_2$ y las columnas como $\vec{c}_1, \vec{c}_2$:

$$\vec{r}_1 = (\cos\theta, -\sin\theta), \quad \vec{r}_2 = (\sin\theta, \cos\theta)$$

$$\vec{c}_1 = \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}, \quad \vec{c}_2 = \begin{pmatrix} -\sin\theta \\ \cos\theta \end{pmatrix}$$

1. Ortogonalidad de las filas: Calculamos el producto escalar $\vec{r}_1 \cdot \vec{r}_2$:

$$\vec{r}_1 \cdot \vec{r}_2 = (\cos\theta)(\sin\theta) + (-\sin\theta)(\cos\theta) = \sin\theta\cos\theta - \sin\theta\cos\theta = 0$$

Las filas son perpendiculares.

2. Normalidad de las filas (Longitud unitaria): Calculamos la norma al cuadrado de cada fila usando la identidad $\cos^2\theta + \sin^2\theta = 1$:

$$\|\vec{r}_1\|^2 = (\cos\theta)^2 + (-\sin\theta)^2 = \cos^2\theta + \sin^2\theta = 1 \implies \|\vec{r}_1\| = 1$$

$$\|\vec{r}_2\|^2 = (\sin\theta)^2 + (\cos\theta)^2 = \sin^2\theta + \cos^2\theta = 1 \implies \|\vec{r}_2\| = 1$$

3. Ortogonalidad de las columnas: Calculamos el producto escalar $\vec{c}_1 \cdot \vec{c}_2$:

$$\vec{c}_1 \cdot \vec{c}_2 = (\cos\theta)(-\sin\theta) + (\sin\theta)(\cos\theta) = -\sin\theta\cos\theta + \sin\theta\cos\theta = 0$$

Las columnas son perpendiculares.

4. Normalidad de las columnas:

$$\|\vec{c}_1\|^2 = \cos^2\theta + \sin^2\theta = 1 \implies \|\vec{c}_1\| = 1$$

$$\|\vec{c}_2\|^2 = (-\sin\theta)^2 + \cos^2\theta = \sin^2\theta + \cos^2\theta = 1 \implies \|\vec{c}_2\| = 1$$

Conclusión: Dado que tanto las filas como las columnas son vectores unitarios y ortogonales entre sí para cualquier valor de θ, la matriz de rotación R_θ es siempre una matriz ortonormal.

Ejercicio 1.2.8. Demuestra que, en 2D, el producto de una matriz de rotación y una de escalado no es conmutativo en general, excepto si el escalado es uniforme.

Solución 1.2.8. Para analizar la conmutatividad entre la rotación y el escalado, definiremos las matrices correspondientes en el espacio bidimensional. Consideraremos las matrices de 2×2, dado que ambas son transformaciones lineales y no requieren necesariamente coordenadas homogéneas para demostrar esta propiedad (aunque el resultado es idéntico en 3×3 con la última fila/columna canónica).

Sean las matrices de rotación R_θ y de escalado $S(s_x, s_y)$:

$$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}, \quad S = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$$

Calculamos el producto $R_\theta \cdot S$ (aplicar escalado y luego rotación):

$$R_\theta \cdot S = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} = \begin{pmatrix} s_x\cos\theta & -s_y\sin\theta \\ s_x\sin\theta & s_y\cos\theta \end{pmatrix}$$

Calculamos el producto $S \cdot R_\theta$ (aplicar rotación y luego escalado):

$$S \cdot R_\theta = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} = \begin{pmatrix} s_x\cos\theta & -s_x\sin\theta \\ s_y\sin\theta & s_y\cos\theta \end{pmatrix}$$

Para que las matrices conmuten, es decir, $R_\theta \cdot S = S \cdot R_\theta$, sus componentes deben ser idénticos término a término. Comparamos los términos fuera de la diagonal principal:

1) Elemento $(1,2)$: $-s_y\sin\theta = -s_x\sin\theta \implies (s_x - s_y)\sin\theta = 0$
2) Elemento $(2,1)$: $s_x\sin\theta = s_y\sin\theta \implies (s_x - s_y)\sin\theta = 0$

Para que la igualdad se cumpla para un ángulo de rotación general ($\sin\theta \neq 0$), es condición necesaria y suficiente que:

$$s_x = s_y$$

Conclusión: Si $s_x \neq s_y$ (escalado no uniforme), los productos matriciales son distintos, demostrando que la operación no es conmutativa en general. Si $s_x = s_y = s$ (escalado uniforme), la matriz

de escalado se convierte en sI (donde I es la identidad), la cual conmuta con cualquier matriz cuadrada.

Ejercicio 1.2.9. Demuestra que en 2D, el producto de una matriz de rotación y otra de traslación (por un vector no nulo) no es conmutativo.

Solución 1.2.9. Dado que la traslación es una transformación afín y no lineal, es imprescindible utilizar **coordenadas homogéneas** para representarla como una multiplicación matricial. Trabajaremos con matrices de 3×3.

Sea R_θ la matriz de rotación y $T_{\vec{t}}$ la matriz de traslación por un vector $\vec{t} = (t_x, t_y)$:

$$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad T_{\vec{t}} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Calculamos el producto $R_\theta \cdot T_{\vec{t}}$ (primero se traslada, luego se rota):

$$R_\theta \cdot T_{\vec{t}} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & t_x\cos\theta - t_y\sin\theta \\ \sin\theta & \cos\theta & t_x\sin\theta + t_y\cos\theta \\ 0 & 0 & 1 \end{pmatrix}$$

Geométricamente, esto rota el punto y también rota el vector de traslación aplicado.

Calculamos el producto $T_{\vec{t}} \cdot R_\theta$ (primero se rota, luego se traslada):

$$T_{\vec{t}} \cdot R_\theta = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Geométricamente, esto rota el punto alrededor del origen y luego aplica la traslación original sin modificar.

Comparación: Observamos la tercera columna (la componente de traslación resultante) de ambas matrices resultantes:

$$\begin{pmatrix} t_x\cos\theta - t_y\sin\theta \\ t_x\sin\theta + t_y\cos\theta \\ 1 \end{pmatrix} \neq \begin{pmatrix} t_x \\ t_y \\ 1 \end{pmatrix}$$

Para que fuesen iguales en un caso general ($\theta \neq 0$), se requeriría que $t_x = 0$ y $t_y = 0$. Dado que el enunciado especifica un vector de traslación no nulo, concluimos que las matrices son distintas.

Conclusión: El orden de las operaciones altera el resultado final: rotar y luego trasladar lleva a una posición diferente que trasladar y luego rotar (donde el desplazamiento también sufre la rotación). Por tanto, no son conmutativas.

Ejercicio 1.2.10. Demuestra que el producto escalar de vectores en 3D es invariante por rotaciones entorno a los ejes cartesianos, y que estas tampoco modifican la longitud de un vector.

Solución 1.2.10. Para demostrar la invariancia del producto escalar en $\mathbb{R}^3$ bajo rotaciones cartesianas, tomaremos sin pérdida de generalidad el caso de una rotación alrededor del eje Z por un ángulo θ. El procedimiento es análogo para los ejes X e Y debido a la simetría cíclica de las coordenadas.

La matriz de rotación $R_{z,\theta}$ se define como:

$$R_{z,\theta} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Sean $\vec{u} = (u_x, u_y, u_z)$ y $\vec{v} = (v_x, v_y, v_z)$ dos vectores arbitrarios. El producto escalar original es:

$$\vec{u} \cdot \vec{v} = u_x v_x + u_y v_y + u_z v_z \tag{1.3}$$

Calculamos los vectores transformados $\vec{u}' = R_{z,\theta}(\vec{u})$ y $\vec{v}' = R_{z,\theta}(\vec{v})$:

$$\vec{u}' = \begin{pmatrix} u_x \cos\theta - u_y \sin\theta \\ u_x \sin\theta + u_y \cos\theta \\ u_z \end{pmatrix}, \quad \vec{v}' = \begin{pmatrix} v_x \cos\theta - v_y \sin\theta \\ v_x \sin\theta + v_y \cos\theta \\ v_z \end{pmatrix}$$

Ahora calculamos el producto escalar de los vectores transformados:

$$\begin{aligned} \vec{u}' \cdot \vec{v}' = &(u_x \cos\theta - u_y \sin\theta)(v_x \cos\theta - v_y \sin\theta) \\ &+ (u_x \sin\theta + u_y \cos\theta)(v_x \sin\theta + v_y \cos\theta) \\ &+ u_z v_z \end{aligned}$$

Expandiendo los términos correspondientes a las componentes x e y (idéntico al caso 2D):

$$\begin{aligned} = &(u_x v_x \cos^2\theta - u_x v_y \cos\theta \sin\theta - u_y v_x \sin\theta \cos\theta + u_y v_y \sin^2\theta) \\ &+ (u_x v_x \sin^2\theta + u_x v_y \sin\theta \cos\theta + u_y v_x \cos\theta \sin\theta + u_y v_y \cos^2\theta) \\ &+ u_z v_z \end{aligned}$$

Agrupando y simplificando usando $\sin^2\theta + \cos^2\theta = 1$:

$$\begin{aligned} \vec{u}' \cdot \vec{v}' &= u_x v_x (\cos^2\theta + \sin^2\theta) + u_y v_y (\sin^2\theta + \cos^2\theta) + u_z v_z \\ &= u_x v_x + u_y v_y + u_z v_z \\ &= \vec{u} \cdot \vec{v} \end{aligned}$$

Invariancia de la longitud: Utilizando la relación $\|\vec{v}\|^2 = \vec{v} \cdot \vec{v}$ y la propiedad recién demostrada:

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

Tomando la raíz cuadrada, concluimos que $\|R_{z,\theta}(\vec{v})\| = \|\vec{v}\|$.

Ejercicio 1.2.11. Demuestra que el producto vectorial de dos vectores rota igual que lo hacen esos dos vectores. Es decir, para cualesquiera vectores $\vec{u}, \vec{v}$ y un ángulo θ con eje $\hat{e}$, se cumple:

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

Solución 1.2.11. Para esta demostración, consideraremos la rotación alrededor del eje Z ($R_{z,\theta}$), ya que la lógica es extensible a cualquier eje cartesiano por permutación de índices.

Definimos $\vec{w} = \vec{u} \times \vec{v}$. Sus componentes son:

$$\vec{w} = \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} = \begin{pmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{pmatrix}$$

Parte 1: Rotación del producto vectorial original ($R_{z,\theta}(\vec{w})$) Aplicamos la matriz de rotación al vector $\vec{w}$:

$$R_{z,\theta}(\vec{w}) = \begin{pmatrix} w_x \cos\theta - w_y \sin\theta \\ w_x \sin\theta + w_y \cos\theta \\ w_z \end{pmatrix}$$

Sustituyendo los componentes de $\vec{w}$:

$$R_{z,\theta}(\vec{w})_x = (u_y v_z - u_z v_y)\cos\theta - (u_z v_x - u_x v_z)\sin\theta \tag{1.4}$$

$$R_{z,\theta}(\vec{w})_y = (u_y v_z - u_z v_y)\sin\theta + (u_z v_x - u_x v_z)\cos\theta \tag{1.5}$$

$$R_{z,\theta}(\vec{w})_z = u_x v_y - u_y v_x \tag{1.6}$$

Parte 2: Producto vectorial de los vectores rotados ($\vec{u}' \times \vec{v}'$) Sean $\vec{u}' = R_{z,\theta}(\vec{u})$ y $\vec{v}' = R_{z,\theta}(\vec{v})$. Sus componentes son:

$$\vec{u}' = (u_x c - u_y s,\ u_x s + u_y c,\ u_z)$$

$$\vec{v}' = (v_x c - v_y s,\ v_x s + v_y c,\ v_z)$$

(donde $c = \cos\theta, s = \sin\theta$).

Calculamos la componente X de $\vec{u}' \times \vec{v}'$:

$$\begin{aligned} (\vec{u}' \times \vec{v}')_x &= u'_y v'_z - u'_z v'_y \\ &= (u_x s + u_y c)v_z - u_z(v_x s + v_y c) \\ &= u_x v_z s + u_y v_z c - u_z v_x s - u_z v_y c \\ &= c(u_y v_z - u_z v_y) - s(u_z v_x - u_x v_z) \end{aligned}$$

Este resultado coincide exactamente con la Ecuación (1).

Calculamos la componente Y de $\vec{u}' \times \vec{v}'$:

$$\begin{aligned} (\vec{u}' \times \vec{v}')_y &= u'_z v'_x - u'_x v'_z \\ &= u_z(v_x c - v_y s) - (u_x c - u_y s)v_z \\ &= u_z v_x c - u_z v_y s - u_x v_z c + u_y v_z s \\ &= s(u_y v_z - u_z v_y) + c(u_z v_x - u_x v_z) \end{aligned}$$

Este resultado coincide exactamente con la Ecuación (2).

Calculamos la componente Z de $\vec{u}' \times \vec{v}'$:

$$\begin{aligned}(\vec{u}' \times \vec{v}')_z &= u'_x v'_y - u'_y v'_x \\ &= (u_x c - u_y s)(v_x s + v_y c) - (u_x s + u_y c)(v_x c - v_y s)\end{aligned}$$

Desarrollando y simplificando:

$$\begin{aligned}&= (u_x v_x cs + u_x v_y c^2 - u_y v_x s^2 - u_y v_y sc) - (u_x v_x sc - u_x v_y s^2 + u_y v_x c^2 - u_y v_y cs) \\ &= u_x v_y (c^2 + s^2) - u_y v_x (s^2 + c^2) \\ &= u_x v_y - u_y v_x\end{aligned}$$

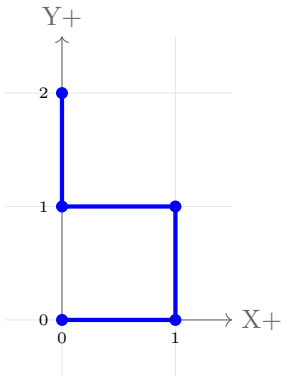
Este resultado coincide con la Ecuación (3).

Conclusión: Dado que todas las componentes coinciden, queda demostrado que:

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

Ejercicio 1.2.12. Crea un script global (autoload) con una función llamada `gancho` (sin parámetros) que crea y devuelve un objeto de la clase `Mesh` con una polilínea azul como la de la figura (los ejes se han dibujado por claridad).

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



Solución 1.2.12. Solución al problema 3.12:

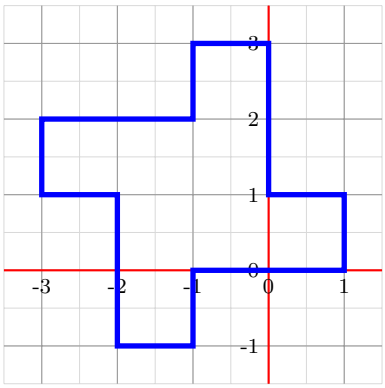
```
# Problema 3.12:
# Crea un script global (autoload) con una función
# llamada gancho (sin parámetros) que crea y
# devuelve un objeto de la clase Mesh con una
# polilínea azul como la de la figura (los ejes se han
# dibujado por claridad).
# Crea en tu proyecto un nodo 2D de tipo
# MeshInstance2D y en _ready asígnale como
# malla (propiedad mesh) el objeto resultado de
# llamar a gancho(), ponle un color azul (propie7
# dad modulate) y verifica que el gancho aparece
```

```
# en pantalla al ejecutar el proyecto.
extends MeshInstance2D

func _ready():
    self.mesh = Global.gancho()
    self.modulate = Color(0,0,1)
```

Además, añadimos el codigo de gancho que se encuentra en el script global:

```
    func gancho() -> ArrayMesh:
        var vertices = PackedVector2Array([
            Vector2(0,0),
            Vector2(1,0),
            Vector2(1,1),
            Vector2(0,1),
            Vector2(0,2)
        ])

        var arrays = []
        arrays.resize(Mesh.ARRAY_MAX)
        arrays[Mesh.ARRAY_VERTEX] = vertices

        var mesh = ArrayMesh.new()
        mesh.add_surface_from_arrays(Mesh.PRIMITIVE_LINE_STRIP,
    arrays)
        return mesh

```

Ejercicio 1.2.13. Crea un nodo 2D de tipo `Node2D` y llámalo `Gancho_x4`. En `_ready`, añádele cuatro nodos hijos de tipo `MeshInstance2D`, cada uno de ellos con un malla creada con la función `gancho` del problema anterior, pero con su `transform` modificada para que el objeto `Gancho_x4` se vea como en la figura (la rejilla y los ejes en rojo se han dibujado por claridad).



Solución 1.2.13. Solución al problema 3.13:

```gdscript
extends Node2D

func _ready():
  var malla_gancho = Global.gancho()

  # Definimos el centro de rotación observado en la imagen
  var centro_rotacion = Vector2(-1, 1)

  # Creamos las 4 instancias
  for i in range(4):
    var instancia = MeshInstance2D.new()
    instancia.mesh = malla_gancho
    instancia.name = "Gancho_" + str(i)

    # Color azul para las aristas (modulate afecta a todo el mesh)
    instancia.modulate = Color(0, 0, 1)

    # Calculamos el ángulo: 0, 90, 180, 270 grados
    var angulo = i * (PI / 2.0) # usamos pi/2 ya que tenemos que el círculo completo es 2pi y como tenemos 4 instancias, 2pi/4 = pi/2

    # --- CÁLCULO DE LA MATRIZ DE TRANSFORMACIÓN ---
    # Aplicamos la fórmula: M = T(C) * R(theta) * T(-C)

    # 1. Matriz para mover el pivote al origen
    var t_al_origen = Transform2D().translated(-centro_rotacion)

    # 2. Matriz de rotación
    var rotacion = Transform2D().rotated(angulo)

    # 3. Matriz para devolver el pivote a su sitio
    var t_de_vuelta = Transform2D().translated(centro_rotacion)

    # En Godot, las matrices se multiplican en orden de aplicación (Padre * Hijo),
    # pero aquí estamos componiendo una sola transformación compleja.
    # El orden lógico es: primero T_al_origen, luego Rotacion, luego T_de_vuelta.
    # M * v = (T_vuelta * (Rot * (T_origen * v)))
    instancia.transform = t_de_vuelta * rotacion * t_al_origen

    add_child(instancia)

# Nota: la función gancho() localmente:
# func gancho() -> ArrayMesh:
```

```
#     var vertices = PackedVector2Array([
#         Vector2(0, 0), Vector2(1, 0), Vector2(1, 1), Vector2
    (0, 1), Vector2(0, 2)
#     ])
#     var am = ArrayMesh.new()
#     var arr = []
#     arr.resize(Mesh.ARRAY_MAX)
#     arr[Mesh.ARRAY_VERTEX] = vertices
#     am.add_surface_from_arrays(Mesh.PRIMITIVE_LINE_STRIP, arr)
#     return am
```

1.3 Sesión 4

Ejercicio 1.3.1. Supongamos que queremos codificar una esfera de radio 1/2 y centro en el origen de dos formas:

1) Por enumeración espacial, dividiendo el cubo que engloba a la esfera en celdas, de forma que haya k celdas por lado del cubo, todas ellas son cubos de $1/k$ de ancho. Cada celda ocupa un bit de memoria (si su centro está en la esfera, se guarda un 1, en otro caso un 0).
2) Usando un modelo de fronteras (una malla indexada de triángulos), en el cual se usa una rejilla de triángulos y aristas que siguen los meridianos y paralelos, habiendo en cada meridiano y en cada paralelo un total de k vértices (se guarda únicamente la tabla de vértices y la de triángulos).

Asumiendo que un `float` y un `int` ocupan 4 bytes cada uno, contesta a estas cuestiones:

1) Expresa el tamaño de ambas representaciones en bytes como una función de k.
2) Suponiendo que $k = 16$ calcula cuántos KB de memoria ocupa cada estructura.
3) Haz lo mismo asumiendo ahora que $k = 1024$ (expresa los resultados en MB).
4) Compara los tamaños de ambas representaciones en ambos casos ($k = 16$ y $k = 1024$).

Solución 1.3.1. Para resolver este ejercicio, analizaremos detalladamente los requisitos de memoria de cada uno de los modelos propuestos, basándonos en la teoría de representación de modelos geométricos, específicamente la diferencia entre modelos de volúmenes (enumeración espacial) y modelos de fronteras (mallas de polígonos).

1) *Expresión del tamaño en memoria como función de k.*
   Analicemos primero el modelo por *enumeración espacial.*
   El espacio que engloba a la esfera de radio $r = 1/2$ es un cubo de lado $L = 2r = 1$. Este cubo se discretiza en una rejilla tridimensional de k celdas por lado. Por lo tanto, el número total de celdas (vóxeles) en el volumen es:

$$N_{celdas} = k \times k \times k = k^3$$

   El enunciado especifica que cada celda ocupa exactamente 1 bit. Para obtener el tamaño en bytes, debemos dividir el número total de bits por 8 (dado que 1 byte = 8 bits).

$$Mem_{enum}(k) = \frac{k^3}{8} \text{ bytes}$$

   Analicemos ahora el modelo de fronteras mediante *malla indexada de triángulos.*

Una malla indexada consta de dos estructuras de datos principales: la tabla de vértices y la tabla de triángulos (índices).
El enunciado indica que la malla se forma siguiendo meridianos y paralelos con k vértices en cada uno. Esto sugiere una topología de rejilla rectangular de dimensiones $k \times k$ mapeada sobre la esfera. En consecuencia, el número de vértices n_V es:

$$n_V = k^2$$

Para una malla cerrada y conexa que representa una esfera, topológicamente equivalente a una rejilla envolvente, el número de caras (triángulos) n_T se aproxima al doble del número de vértices (según la característica de Euler para mallas triangulares cerradas donde $n_T \approx 2n_V$). Si consideramos una rejilla de $(k-1) \times (k-1)$ cuadriláteros, y cada cuadrilátero se divide en 2 triángulos, tendríamos $2(k-1)^2$ triángulos. Para valores grandes de k, podemos aproximar el número de triángulos como:

$$n_T \approx 2k^2$$

Calculamos ahora el uso de memoria para cada tabla:

1) *Tabla de vértices:* Cada vértice almacena 3 coordenadas (x, y, z) de tipo `float`. Si cada `float` ocupa 4 bytes, el tamaño de un vértice es $3 \times 4 = 12$ bytes.

$$Mem_{vert} = 12 \times n_V = 12k^2 \text{ bytes}$$

2) *Tabla de triángulos:* Cada triángulo almacena 3 índices de tipo `int`. Si cada `int` ocupa 4 bytes, el tamaño de un triángulo es $3 \times 4 = 12$ bytes.

$$Mem_{tri} = 12 \times n_T \approx 12 \times (2k^2) = 24k^2 \text{ bytes}$$

El tamaño total de la malla indexada es la suma de ambas tablas:

$$Mem_{malla}(k) = 12k^2 + 24k^2 = 36k^2 \text{ bytes}$$

2) *Cálculo de memoria para $k = 16$ (en KB).*
Sustituimos $k = 16$ en las funciones obtenidas:
Para la *enumeración espacial*:

$$Mem_{enum}(16) = \frac{16^3}{8} = \frac{4096}{8} = 512 \text{ bytes}$$

Convirtiendo a Kilobytes (1 KB = 1024 bytes):

$$Mem_{enum}(16) = \frac{512}{1024} = 0{,}5 \text{ KB}$$

Para la *malla indexada*:

$$Mem_{malla}(16) = 36 \times 16^2 = 36 \times 256 = 9216 \text{ bytes}$$

Convirtiendo a Kilobytes:

$$Mem_{malla}(16) = \frac{9216}{1024} = 9 \text{ KB}$$

3) *Cálculo de memoria para $k = 1024$ (en MB).*
Sustituimos $k = 1024$ en las funciones. Nótese que $1024 = 2^{10}$.
Para la *enumeración espacial*:

$$Mem_{enum}(1024) = \frac{(2^{10})^3}{2^3} = \frac{2^{30}}{2^3} = 2^{27} \text{ bytes}$$

Sabemos que 1 MB $= 1024^2$ bytes $= 2^{20}$ bytes.

$$Mem_{enum}(1024) = \frac{2^{27}}{2^{20}} = 2^7 = 128 \text{ MB}$$

Para la *malla indexada*:

$$Mem_{malla}(1024) = 36 \times (1024)^2 = 36 \times 2^{20} \text{ bytes}$$

Convirtiendo a Megabytes:

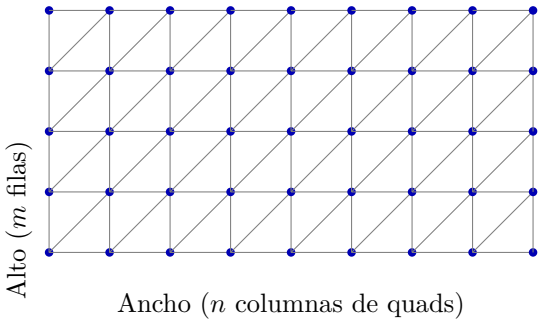
$$Mem_{malla}(1024) = 36 \text{ MB}$$

4) *Comparación de tamaños.*
Los resultados obtenidos ilustran la diferencia fundamental en la complejidad espacial entre los modelos volumétricos y los de frontera.
   1) *Caso $k = 16$ (Baja resolución):* La enumeración espacial ocupa *menos* memoria (0,5 KB) que la malla indexada (9 KB). Esto se debe a que, para resoluciones muy bajas, el coste de almacenar coordenadas e índices explícitos (36 bytes por elemento efectivo) supera el coste de almacenar simplemente 1 bit por celda, dado que el volumen total (k^3) aún no ha crecido lo suficiente para dominar la expresión.
   2) *Caso $k = 1024$ (Alta resolución):* La enumeración espacial ocupa significativamente *más* memoria (128 MB) que la malla indexada (36 MB). Aquí se observa la naturaleza cúbica $O(k^3)$ de la enumeración espacial frente a la naturaleza cuadrática $O(k^2)$ del modelo de fronteras. Al aumentar la resolución, el número de celdas interiores (volumen) crece mucho más rápido que el número de vértices necesarios para representar la superficie (área).

   Conclusión: La enumeración espacial es extremadamente ineficiente en memoria para altas resoluciones, mientras que los modelos de frontera (mallas) son mucho más eficientes para representar objetos sólidos mediante su superficie, especialmente a medida que aumenta la precisión requerida (k).

Ejercicio 1.3.2. Considera una malla indexada (tabla de vértices y tabla de caras, esta última con índices de vértices) con una topología de rejilla rectangular. La rejilla está compuesta por n columnas de pares de triángulos y m filas. Esto implica que la estructura tiene $n + 1$ columnas de vértices y $m + 1$ filas de vértices, con $n, m > 0$.

La figura siguiente ilustra un esquema simplificado de dicha topología (donde los puntos azules representan los vértices y las líneas las aristas que forman los triángulos):



En relación a este tipo de mallas, responde a las siguientes cuestiones:

(a) Supongamos que un `float` ocupa 4 bytes y un `int` ocupa también 4 bytes. ¿Qué tamaño en memoria ocupa la malla completa en bytes? Ten en cuenta únicamente el tamaño de la tabla de vértices y la tabla de triángulos. Expresa el tamaño como una función de m y n.

(b) Calcula el tamaño exacto en KiloBytes (KB) suponiendo que $m = n = 128$.

(c) Supongamos que m y n son ambos grandes (es decir, asumimos que términos como $1/n$ y $1/m$ son despreciables frente a 1). Deduce qué relación aproximada existe entre el número de caras (n_C) y el número de vértices (n_V) en este tipo de mallas.

Solución 1.3.2. Para resolver este problema, analizaremos por separado el consumo de memoria de la geometría (tabla de vértices) y de la topología (tabla de triángulos).

(a) **Cálculo de la función de memoria $Mem(m, n)$ en bytes**
Primero determinamos la cantidad de elementos:

- **Número de vértices (n_V):** La rejilla tiene m filas de celdas y n columnas de celdas. Los vértices se sitúan en las intersecciones.

$$\text{Filas de vértices} = m + 1$$

$$\text{Columnas de vértices} = n + 1$$

$$n_V = (m + 1)(n + 1)$$

- **Número de caras/triángulos (n_C):** Cada celda de la rejilla (formada por la intersección de una fila y una columna) es un cuadrilátero dividido en 2 triángulos.

$$\text{Número de celdas} = m \times n$$

$$n_C = 2 \times (m \times n) = 2mn$$

Ahora calculamos el uso de memoria sabiendo que 1 `float` = 4 bytes y 1 `int` = 4 bytes:

- **Memoria de la Tabla de Vértices (M_V):** Cada vértice almacena 3 coordenadas (x, y, z) de tipo `float`.

$$M_V = n_V \times 3 \times 4 \text{ bytes} = 12(m + 1)(n + 1) \text{ bytes}$$

- **Memoria de la Tabla de Triángulos (M_T):** Cada triángulo almacena 3 índices de vértices (i, j, k) de tipo `int`.

$$M_T = n_C \times 3 \times 4 \text{ bytes} = 12 \times (2mn) \text{ bytes} = 24mn \text{ bytes}$$

Memoria Total (Mem):

$$Mem(m,n) = M_V + M_T$$

$$Mem(m,n) = 12(mn + m + n + 1) + 24mn$$

Agrupando términos semejantes:

$$Mem(m,n) = 12mn + 12m + 12n + 12 + 24mn$$

$$\boxed{Mem(m,n) = 36mn + 12m + 12n + 12 \text{ bytes}}$$

(b) **Cálculo para $m = n = 128$**
Sustituimos m y n por 128 en la fórmula obtenida:

$$Mem(128,128) = 36(128 \times 128) + 12(128) + 12(128) + 12$$

$$Mem(128,128) = 36(16384) + 1536 + 1536 + 12$$

$$Mem(128,128) = 589824 + 3084$$

$$Mem(128,128) = 592908 \text{ bytes}$$

Para convertir a KiloBytes (KB), dividimos por 1024:

$$\text{Memoria en KB} = \frac{592908}{1024} \approx 579{,}01 \text{ KB}$$

Resultado: Aproximadamente **579 KB**.

(c) **Relación asintótica entre n_C y n_V**
Partimos de las expresiones deducidas en el apartado (a):

$$n_V = (m+1)(n+1) = mn + m + n + 1$$

$$n_C = 2mn$$

Si asumimos que m y n son grandes, los términos lineales (m, n) y el término constante (1) son despreciables frente al término cuadrático (mn). Matemáticamente:

$$\lim_{m,n\to\infty} \frac{n_V}{mn} = \lim_{m,n\to\infty} \frac{mn + m + n + 1}{mn} = 1$$

Por tanto, para valores grandes, podemos aproximar:

$$n_V \approx mn$$

Nota: se divide por el término de mayor grado porque de esta manera, en matemáticas, vemos como se comporta en el infinito, otra opción es el mismo límite de nc/nv.
Calculamos la relación (ratio) entre el número de caras y el número de vértices:

$$\frac{n_C}{n_V} \approx \frac{2mn}{mn} = 2$$

Conclusión: En mallas cerradas o mallas de rejilla densas (donde los efectos de borde son insignificantes), **el número de caras (triángulos) es aproximadamente el doble**

que el número de vértices:

$$n_C \approx 2n_V$$

Ejercicio 1.3.3. Imagina de nuevo una malla con topología de rejilla, en la cual hay n columnas de pares de triángulos y m filas. Supongamos que usamos una representación como **tiras de triángulos** (Triangle Strips), de forma que cada fila de triángulos (con $2n$ triángulos) se almacena en una tira independiente, habiendo un total de m tiras.

La estructura de datos consta de una tabla de punteros a tiras (que tiene un entero para el número de tiras y m punteros, donde cada puntero ocupa 8 bytes) y los arrays de coordenadas de las tiras. Asume que las coordenadas son de tipo `float` (4 bytes) y que no se usan índices (las coordenadas se almacenan explícitamente en el orden de la tira).

Responde a las siguientes cuestiones:

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

(b) Para m y n grandes (asumiendo que los términos lineales son despreciables frente a los cuadráticos), describe qué relación hay entre el tamaño en memoria de la malla indexada (Problema 4.2) y el tamaño de la malla almacenada como tiras de triángulos.

(c) Si suponemos que la transformación de cada vértice se hace en un tiempo constante igual a la unidad, describe qué relación hay entre los tiempos de procesamiento de vértices para esta malla cuando se representa como una malla indexada y como tiras de triángulos.

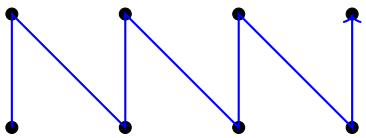
Solución 1.3.3. Para resolver este ejercicio, primero debemos determinar cuántos vértices se almacenan explícitamente en una tira de triángulos que representa una fila de la rejilla.

- Una tira de triángulos que contiene k triángulos requiere $k+2$ vértices. Básicamente, sabemos que cada nuevo triángulo en la tira comparte dos vértices con el triángulo anterior, si para 2 triangulos necesitamos 4 vértices, por inducción (3 vértices × (k-1) restantes × 1 vértice que añadimos) se llega a la fórmula $k + 2$.
- En la rejilla descrita, cada fila contiene n celdas cuadradas (pares de triángulos). Por lo tanto, el número de triángulos por fila (por tira) es $k = 2n$.
- El número de vértices almacenados por cada tira es:

$$V_{tira} = (2n) + 2 = 2n + 2$$

- Cada vértice consta de 3 coordenadas (x, y, z) de tipo `float` (4 bytes cada uno). El tamaño de un vértice es:

$$B_{vertice} = 3 \times 4 \text{ bytes} = 12 \text{ bytes}$$



Ejemplo de una tira ($n = 3$ quads, $2n = 6$ triángulos, $2n + 2 = 8$ vértices)

(a) Cálculo de Memoria

(1) Función de n y m en bytes

El tamaño total M_{total} se compone del tamaño de los datos de las tiras y la sobrecarga de la estructura de punteros.

1) **Memoria de los vértices:** Hay m tiras.

$$M_{geom} = m \times (2n+2) \text{ vértices} \times 12 \text{ bytes/vértice}$$

$$M_{geom} = 12m(2n+2) = 24nm + 24m \text{ bytes}$$

2) **Memoria de la tabla de punteros:** Contiene 1 entero (4 bytes) y m punteros (8 bytes c/u[1]).

$$M_{estructura} = 4 + 8m \text{ bytes}$$

3) **Memoria Total:**

$$M_{total}(n,m) = (24nm + 24m) + (8m + 4)$$

$$\boxed{M_{total}(n,m) = 24nm + 32m + 4 \text{ bytes}}$$

(2) Cálculo para $m = n = 128$

Sustituimos los valores en la fórmula obtenida:

$$M_{total}(128,128) = 24(128 \times 128) + 32(128) + 4$$

$$M_{total} = 24(16384) + 4096 + 4$$

$$M_{total} = 393216 + 4096 + 4 = 397316 \text{ bytes}$$

Para convertir a Kilobytes (asumiendo 1 KB = 1024 bytes):

$$M_{KB} = \frac{397316}{1024} \approx \boxed{388{,}00 \text{ KB}}$$

(b) Relación de tamaño con Malla Indexada

Para n, m grandes, solo consideramos los términos de mayor orden (nm).

1. Tamaño Malla Indexada (del Problema 4.2):

- Vértices únicos: $\approx nm$. Tamaño: $nm \times 12$ bytes.
- Triángulos: $\approx 2nm$. Índices: $2nm \times 3$ índices $\times 4$ bytes $= 24nm$ bytes. El cálculo de los índices ha sido número de triángulos por 3 índices por triángulo por 4 bytes por índice.
- Total Indexada: $12nm + 24nm =$ **36nm** bytes.

2. Tamaño Tiras de Triángulos (obtenido en a):

- Total Tiras: **24nm** bytes.

Comparación:

Calculamos la relación (ratio) entre ambas representaciones:

$$\frac{\text{Memoria Tiras}}{\text{Memoria Indexada}} \approx \frac{24nm}{36nm} = \frac{2}{3}$$

Conclusión:

La representación mediante tiras de triángulos ocupa aproximadamente **el 66.6 % (dos tercios)** de la memoria que ocupa la malla indexada para esta topología de rejilla. Esto se debe a que, aunque las tiras duplican los vértices compartidos entre filas adyacentes, evitan el coste de almacenar

[1] cada uno

3 enteros por cada triángulo, que es más costoso que almacenar coordenadas repetidas en este escenario específico.

(c) Comparación de tiempos de procesamiento

El tiempo de procesamiento de vértices (T_{proc}) en la GPU depende del número de veces que se debe ejecutar el *Vertex Shader*.

1. Malla Indexada:

Gracias al *Post-Transform Cache* de la GPU, los vértices indexados suelen procesarse una sola vez por cada vértice único (idealmente).

$$V_{unicos} \approx nm \implies T_{index} \propto nm$$

2. Tiras de Triángulos (No Indexadas):

En la implementación descrita (arrays de arrays), los vértices se envían explícitamente por cada tira. Los vértices que se encuentran en la frontera entre la fila i y la fila $i+1$ están duplicados en memoria (aparecen en la tira i y en la tira $i+1$). La GPU no sabe que son el mismo vértice geométrico y debe procesarlos dos veces.

$$V_{tiras} = m(2n+2) \approx 2nm \implies T_{tiras} \propto 2nm$$

Conclusión:

$$\frac{T_{tiras}}{T_{index}} \approx \frac{2nm}{nm} = 2$$

El tiempo de procesamiento usando tiras de triángulos independientes (no indexadas) es aproximadamente **el doble** que usando una malla indexada. Aunque las tiras ahorran memoria de almacenamiento en disco/RAM en este caso, son menos eficientes computacionalmente porque obligan a la GPU a transformar los mismos vértices frontera múltiples veces.

Ejercicio 1.3.4. Supongamos una malla cerrada, simplemente conexa (topológicamente equivalente a una esfera), cuyas caras son triángulos y cuyas aristas son todas adyacentes a exactamente dos caras (la malla es un poliedro simplemente conexo de caras triangulares).

Considera el número de vértices n_V, el número de aristas n_A y el número de caras n_C en este tipo de mallas.

Demuestra que cualquiera de esos números determina a los otros dos, en concreto, demuestra que se cumplen estas dos igualdades:

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

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

Solución 1.3.4. Para demostrar las igualdades propuestas, utilizaremos dos propiedades fundamentales de la topología de superficies cerradas y de las mallas triangulares. Procederemos paso a paso estableciendo un sistema de ecuaciones.

1) **Aplicación de la Fórmula de Euler-Poincaré:**
   Dado que el enunciado especifica que la malla es cerrada y topológicamente equivalente a una esfera (género $g = 0$), se cumple la característica de Euler para poliedros convexos:

$$n_V - n_A + n_C = 2 \tag{1.7}$$

Donde:
- n_V: Número de vértices.
- n_A: Número de aristas.
- n_C: Número de caras.

2) **Relación de adyacencia Caras-Aristas:**
En una malla compuesta exclusivamente por triángulos, cada cara tiene exactamente 3 aristas. Además, al ser una variedad cerrada (manifold), cada arista es compartida exactamente por 2 caras.
Podemos contar el número total de "lados" de los triángulos de dos formas:
- Multiplicando el número de caras por 3: $3 \cdot n_C$.
- Multiplicando el número de aristas por 2 (ya que cada arista cuenta para dos caras): $2 \cdot n_A$.

Igualando ambas cantidades obtenemos la segunda ecuación fundamental:

$$3n_C = 2n_A \implies n_C = \frac{2}{3}n_A \quad \text{o bien} \quad n_A = \frac{3}{2}n_C \tag{1.8}$$

3) **Demostración de $n_C = 2(n_V - 2)$:**
Sustituimos n_A en la Ecuación (1.7) utilizando la relación obtenida en (1.8) ($n_A = \frac{3}{2}n_C$):

$$n_V - \left(\frac{3}{2}n_C\right) + n_C = 2$$

Multiplicamos toda la ecuación por 2 para eliminar la fracción:

$$2n_V - 3n_C + 2n_C = 4$$

Simplificamos los términos de n_C:

$$2n_V - n_C = 4$$

Despejamos n_C:

$$n_C = 2n_V - 4$$

Factorizamos el 2:

$$\boxed{n_C = 2(n_V - 2)}$$

Q.E.D. (Queda demostrado que el número de caras es aproximadamente el doble que el de vértices).

4) **Demostración de $n_A = 3(n_V - 2)$:**
Partimos de nuevo de la Ecuación (1.7), pero esta vez sustituimos n_C despejándolo de (1.8) como $n_C = \frac{2}{3}n_A$:

$$n_V - n_A + \left(\frac{2}{3}n_A\right) = 2$$

Multiplicamos toda la ecuación por 3 para eliminar la fracción:

$$3n_V - 3n_A + 2n_A = 6$$

Simplificamos los términos de n_A:

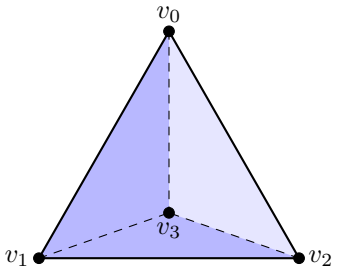
$$3n_V - n_A = 6$$

Despejamos n_A:

$$n_A = 3n_V - 6$$

Factorizamos el 3:

$$\boxed{n_A = 3(n_V - 2)}$$



Ejemplo: Tetraedro ($n_V = 4, n_A = 6, n_C = 4$)

Ejercicio 1.3.5. En una malla indexada, queremos añadir a la estructura de datos una tabla de aristas. Será un vector `ari`, que en cada entrada tendrá una tupla de tipo `Vector2i` (contiene dos `int`) con los índices en la tabla de vértices de los dos vértices en los extremos de la arista. El orden en el que aparecen los vértices en una arista es indiferente, pero cada arista debe aparecer una sola vez.

Escribe el código de una función GDScript para crear y calcular la tabla de aristas a partir de la tabla de triángulos. Intenta encontrar una solución con la mínima complejidad en tiempo y memoria posible. Suponer que el número de vértices adyacentes a uno cualquiera de ellos es como mucho un valor constante $k > 0$, valor que no depende del número total de vértices, que llamamos n.

Considerar dos casos:

(a) Los triángulos se dan con orientación no coherente: esto quiere decir que si un triángulo está formado por los vértices i, j, k, estos tres índices pueden aparecer en cualquier orden en la correspondiente entrada de la tabla de triángulos. Además, no sabemos si la malla es cerrada o no.

(b) Los triángulos se dan con orientación coherente: esto quiere decir que si dos triángulos comparten una arista entre los vértices i y j, entonces en uno de los triángulos la arista aparece como (i, j) y en el otro aparece como (j, i). Además, asumimos que la malla es cerrada, es decir, que cada arista es compartida por exactamente dos triángulos.

Solución 1.3.5. Para resolver este problema, debemos iterar sobre la tabla de triángulos y extraer las aristas potenciales. La diferencia fundamental entre los dos casos radica en cómo garantizamos la unicidad de las aristas (evitar duplicados) de manera eficiente.

Caso (a): Orientación no coherente y malla general

En este escenario, no podemos predecir el orden de los índices ni cuántas veces aparece una arista (podría ser 1 si es frontera, o 2 si es interna, o más si la malla no es "manifold").

Estrategia:

1) Recorremos cada triángulo y extraemos sus 3 aristas: (v_0, v_1), (v_1, v_2) y (v_2, v_0).
2) Para identificar una arista de forma única sin importar el orden (es decir, que la arista $i-j$ sea igual a $j-i$), ordenamos los índices de cada par: guardamos siempre $(\text{mín}(i,j), \text{máx}(i,j))$.
3) Usamos una estructura de datos tipo *Set* (Conjunto) o un Diccionario para almacenar las aristas encontradas. Esto elimina duplicados automáticamente con una complejidad promedio de $O(1)$ por inserción.

Código GDScript:

```
func calcular_aristas_caso_a(triangulos: Array[Vector3i]) ->
    Array[Vector2i]:
    var aristas_unicas = {} # Usamos un diccionario como Set
    for t in triangulos:
        # Extraemos los 3 pares de vértices
        var pares = [
            Vector2i(t[0], t[1]),
            Vector2i(t[1], t[2]),
            Vector2i(t[2], t[0])
        ]
        for par in pares:
            # Normalizamos la arista: (menor, mayor)
            var a = par.x
            var b = par.y
            var key: Vector2i
            if a < b:
                key = Vector2i(a, b)
            else:
                key = Vector2i(b, a)
            # Insertamos en el diccionario (la clave evita
    duplicados)
            aristas_unicas[key] = true
    # Convertimos las claves del diccionario a un Array
    var ari: Array[Vector2i] = []
    for key in aristas_unicas.keys():
        ari.append(key)
    return ari
```

Complejidad:

- Tiempo: $O(N_t)$, donde N_t es el número de triángulos (asumiendo inserción en hash map constante).
- Memoria: $O(N_a)$, donde N_a es el número de aristas únicas, necesario para el diccionario auxiliar.

Caso (b): Orientación coherente y malla cerrada

En este escenario, tenemos una propiedad topológica fuerte: cada arista interna es compartida por exactamente dos triángulos. Debido a la orientación coherente, si la arista conecta los vértices A y B:

- En el Triángulo 1 aparecerá como secuencia $\cdots \to A \to B \to \ldots$
- En el Triángulo 2 aparecerá como secuencia $\cdots \to B \to A \to \ldots$

Estrategia: Para evitar duplicados sin usar memoria extra (diccionarios), podemos aplicar una regla de selección simple: **Solo añadimos la arista si el índice de origen es menor que el índice de destino ($i < j$).**

- Cuando procesemos el par (i, j) donde $i < j$, lo guardamos.
- Cuando procesemos el par (j, i) (que existirá obligatoriamente en el triángulo vecino), como $j > i$, lo ignoramos.

Esto garantiza que cada arista se añade exactamente una vez.

Código GDScript:

```
func calcular_aristas_caso_b(triangulos: Array[Vector3i]) ->
    Array[Vector2i]:
    var ari: Array[Vector2i] = []
    for t in triangulos:
        # Definimos los 3 pares tal cual aparecen en el orden
    del triángulo
        # Arista 0-1
        if t[0] < t[1]:
            ari.append(Vector2i(t[0], t[1]))
        # Arista 1-2
        if t[1] < t[2]:
            ari.append(Vector2i(t[1], t[2]))
        # Arista 2-0
        if t[2] < t[0]:
            ari.append(Vector2i(t[2], t[0]))
    return ari
```

Complejidad:

- Tiempo: $O(N_t)$. Es extremadamente rápido porque solo implica comparaciones de enteros.
- Memoria: $O(1)$ de memoria auxiliar (no necesitamos estructuras intermedias como diccionarios, escribimos directamente en el resultado).

Ejercicio 1.3.6. Escribe el pseudo-código de la función para calcular el área total de una malla indexada de triángulos, a partir de la tabla de vértices y de la tabla de triángulos.

Será una función GDScript que acepta ambas tablas:

- `vertices`: un array de tipo `Vector3` que contiene las posiciones espaciales.
- `triangulos`: un array de tipo `Vector3i`, donde cada elemento contiene los tres índices enteros que forman una cara.

La función debe devolver el área total como un valor de punto flotante (`float`).

Solución 1.3.6. Para resolver este problema, debemos basarnos en la geometría vectorial. El área de cualquier polígono complejo en 3D (la malla) es la suma de las áreas de sus primitivas individuales (los triángulos).

Fundamento Matemático

El área de un triángulo en el espacio 3D definido por tres puntos P_0, P_1, P_2 se puede calcular utilizando el **producto vectorial** (o producto cruz).

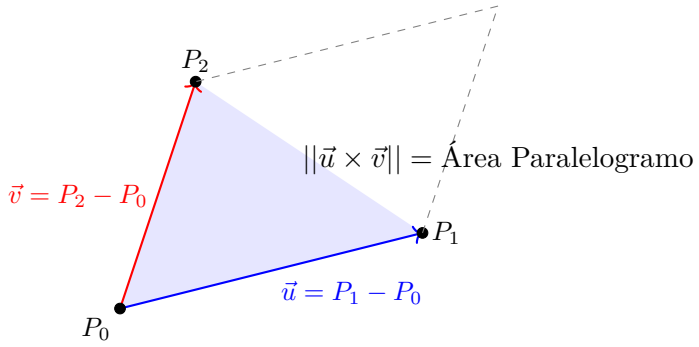
1) Definimos dos vectores que representen dos lados del triángulo partiendo de un vértice común, por ejemplo P_0:

$$\vec{u} = P_1 - P_0$$

$$\vec{v} = P_2 - P_0$$

2) El producto vectorial $\vec{w} = \vec{u} \times \vec{v}$ genera un vector perpendicular al plano del triángulo.
3) La magnitud (o longitud) de este vector resultante, $||\vec{w}||$, es igual al área del **paralelogramo** formado por los vectores $\vec{u}$ y $\vec{v}$.
4) Dado que un triángulo es la mitad de un paralelogramo, el área del triángulo es la mitad de dicha magnitud:

$$\text{Área}_{tri} = \frac{1}{2}||\vec{u} \times \vec{v}||$$



Implementación en GDScript

El algoritmo consiste en iterar sobre la tabla de triángulos, recuperar las coordenadas de los vértices usando los índices, calcular el área de cada triángulo individual y acumularla en una variable total.

```
func calcular_area_malla(vertices: Array[Vector3], triangulos:
    Array[Vector3i]) -> float:
    var area_total: float = 0.0
    for t in triangulos:
        var p0: Vector3 = vertices[t[0]]
        var p1: Vector3 = vertices[t[1]]
        var p2: Vector3 = vertices[t[2]]
        var u: Vector3 = p1 - p0
        var v: Vector3 = p2 - p0
        var vector_area: Vector3 = u.cross(v)
        var area_triangulo: float = vector_area.length() * 0.5
        area_total += area_triangulo
    return area_total
```

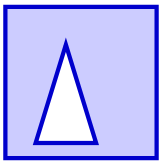
Análisis de complejidad: Si N_t es el número de triángulos (longitud del array `triangulos`), la

complejidad temporal es $O(N_t)$, ya que realizamos un número constante de operaciones matemáticas (restas y producto vectorial) por cada cara de la malla.

1.4 Sesión 5

Ejercicio 1.4.1. Implementa un proyecto cuya escena principal tenga un nodo de tipo `Node2D` con varios nodos hijos, que formen la figura con un cuadrado de lado 2, centrado en el origen, y con un triángulo inscrito.

El cuadrado debe estar relleno de azul claro, el triángulo de blanco, y las aristas deben verse de color azul oscuro.



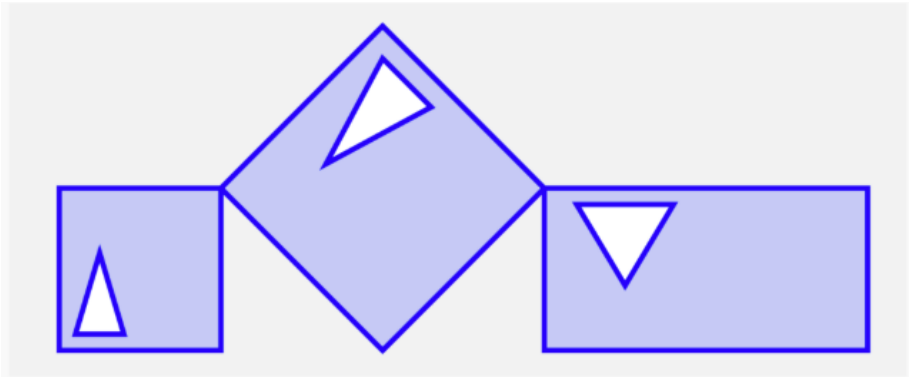
Solución 1.4.1. Solución al problema 5.1:

```
# Problema 5.1:
# Implementa un proyecto cuya escena principal tenga un de tipo Node2D con
# varios nodos hijos, que formen la figura con un cuadrado de lado 2, centrado
# en el origen, y con un triángulo inscrito. El cuadrado debe estar relleno de azul
# claro, el triángulo de blanco, y las aristas deben verse de color azul oscuro.


extends Node2D

# Referencia al Singleton (Autoload) que contiene las herramientas
# const Utils = preload("res://FuncionesAuxiliaresT5.gd") # otra opción posible
# NOTA: Si ya lo tenemos configurado como Autoload global, puedes usar directamente
# el nombre 'FuncionesAuxiliaresT5' en lugar de la variable 'Utils'.

func _ready():
  # ================================
  # DEFINICIÓN DE GEOMETRÍA
  # Cuadrado de lado 2 centrado en el origen: va de -1 a 1 en X e Y.
  # Triángulo inscrito: definimos vértices que quepan dentro del cuadrado.
  # ================================
```

```
21
22  # Vértices para el RELLENO del cuadrado (2 triángulos para
      formar un quad)
23  var v_cuadrado_relleno = PackedVector2Array([
24    Vector2(-1, -1), Vector2(1, -1), Vector2(-1, 1), # Triángulo
      1
25    Vector2(1, -1), Vector2(1, 1), Vector2(-1, 1)    # Triángulo
      2
26  ])
27
28  # Vértices para el BORDE del cuadrado (Polilínea cerrada)
29  var v_cuadrado_borde = PackedVector2Array([
30    Vector2(-1, -1), Vector2(1, -1),
31    Vector2(1, 1), Vector2(-1, 1),
32    Vector2(-1, -1) # Repetimos el primero para cerrar
33  ])
34
35  # Vértices para el RELLENO del triángulo (1 triángulo simple)
36  # Lo hacemos un poco más pequeño para que se vea "dentro"
      claramente
37  var v_triangulo_relleno = PackedVector2Array([
38    Vector2(-0.5, -0.5), Vector2(0.5, -0.5), Vector2(0, 0.8)
39  ])
40
41  # Vértices para el BORDE del triángulo (Polilínea cerrada)
42  var v_triangulo_borde = PackedVector2Array([
43    Vector2(-0.5, -0.5), Vector2(0.5, -0.5),
44    Vector2(0, 0.8), Vector2(-0.5, -0.5)
45  ])
46
47  # ================================
48  # CREACIÓN DE MALLAS
49  # ================================
50
51  # 1. Crear el cuadrado relleno (Azul Claro)
52  # Usamos una función local porque el Autoload solo crea
      LINE_STRIP
53  var mesh_cuadrado_relleno = FuncionesAuxiliaresT5.
      _crear_malla_rellena(v_cuadrado_relleno)
54  var nodo_cuad_relleno = FuncionesAuxiliaresT5.
      CrearMeshInstance2D(mesh_cuadrado_relleno, Transform2D())
55  nodo_cuad_relleno.modulate = Color(0.6, 0.8, 1.0) # Azul claro
56  add_child(nodo_cuad_relleno)
57
58  # 2. Crear el borde del cuadrado (Azul Oscuro)
59  # Usamos la función del Autoload (genera líneas)
60  var mesh_cuadrado_borde = FuncionesAuxiliaresT5.CrearArrayMesh
      (v_cuadrado_borde)
```

```
var nodo_cuad_borde = FuncionesAuxiliaresT5.CrearMeshInstance2D(mesh_cuadrado_borde, Transform2D())
nodo_cuad_borde.modulate = Color(0.0, 0.0, 0.5) # Azul oscuro
# Opcional: aumentar grosor de línea si se usa un material específico,
# pero por defecto Godot dibuja líneas de 1px.
add_child(nodo_cuad_borde)

# 3. Crear el triángulo relleno (Blanco)
var mesh_tri_relleno = FuncionesAuxiliaresT5._crear_malla_rellena(v_triangulo_relleno)
var nodo_tri_relleno = FuncionesAuxiliaresT5.CrearMeshInstance2D(mesh_tri_relleno, Transform2D())
nodo_tri_relleno.modulate = Color.WHITE # Blanco
add_child(nodo_tri_relleno)

# 4. Crear el borde del triángulo (Azul Oscuro)
var mesh_tri_borde = FuncionesAuxiliaresT5.CrearArrayMesh(v_triangulo_borde)
var nodo_tri_borde = FuncionesAuxiliaresT5.CrearMeshInstance2D(mesh_tri_borde, Transform2D())
nodo_tri_borde.modulate = Color(0.0, 0.0, 0.5) # Azul oscuro
add_child(nodo_tri_borde)

# Ajuste de visualización: Mover todo al centro de la pantalla para verlo mejor
# position = get_viewport_rect().size / 2
# scale = Vector2(100, 100) # Escalamos porque 2 pixels es muy pequeño # al activarlo se ve justo en el medio de toda la vista por lo que hay que hacer muy pequeña la misma
```

Ejercicio 1.4.2. Crea un proyecto Godot con una escena principal con un nodo raíz compuesto. Ese nodo tendrá tres hijos, cada uno es una instancia de la escena del problema anterior, pero con una transformación distinta.



Solución 1.4.2. Solución al problema 5.2:

```
extends Node2D

# Cargamos la escena del Problema 5.1 para instanciarla
const ESCENA_BASE = preload("res://escenaHijos/problema_5_1.tscn")

# Definimos una escala base para que se vea bien en pantalla
const S = 1

func _ready():
  # Centramos todo el conjunto en la pantalla
  # position = Vector2(200, 300)

  # ==========================
  # INSTANCIA 1: CUADRADO ORIGINAL
  # ==========================
  var ins1 = ESCENA_BASE.instantiate()
  ins1.scale = Vector2(S, S)
  ins1.position = Vector2(0, 0)
  add_child(ins1)

  # CALCULO DEL PUNTO DE CONEXIÓN 1 -> 2
  # La esquina superior derecha del cuadrado (ins1) en coordenadas locales es (1, -1).
  # En coordenadas globales (relativas al padre) es: (S, -S).

  # ==========================
  # INSTANCIA 2: ROMBO (ROTADO 45 GRADOS)
  # ==========================
  var ins2 = ESCENA_BASE.instantiate()
  ins2.scale = Vector2(S, S)
  ins2.rotation_degrees = 135

  # CÁLCULO DE POSICIÓN PARA CONECTAR:
  # El rombo tiene su vértice IZQUIERDO a una distancia de 'sqrt(2) * S' de su centro.
  # Queremos que ese vértice coincida con la esquina (S, -S) del cuadrado.
  # PosX = (Posición Borde Cuadrado) + (Distancia al centro del Rombo)
  # PosX = S + (S * sqrt(2))
  # PosY = S (para alinearse con la parte superior del cuadrado)

  var offset_rombo = S * sqrt(2)
  ins2.position = Vector2(S + offset_rombo, S)

  add_child(ins2)

```

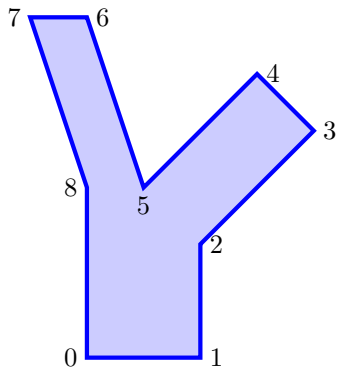
```
# CALCULO DEL PUNTO DE CONEXIÓN 2 -> 3
# El vértice DERECHO del rombo está a 'offset_rombo' a la derecha de su centro.
# Posición Global Vértice Derecho = ins2.position + (offset_rombo, 0)

# ==========================
# INSTANCIA 3: RECTÁNGULO INVERTIDO (ESCALADO)
# ==========================
var ins3 = ESCENA_BASE.instantiate()
# Escalado (2, -1):
# X = 2 (Doble de ancho)
# Y = -1 (Reflexión vertical, el triángulo apunta abajo)
ins3.scale = Vector2(2 * S, -S)

# CÁLCULO DE POSICIÓN PARA CONECTAR:
# La "esquina superior izquierda" visual de este rectángulo corresponde
# geométricamente a (-1, 1) local antes de escalar, o (-2S, -S) después de escalar.
# Queremos que (-2S, -S) coincida con el vértice derecho del rombo.

# Coordenada X del vértice derecho del rombo:
var x_conexion = ins2.position.x + offset_rombo

# La posición del centro de ins3 debe ser tal que su izquierda (-2S) toque la conexión.
ins3.position.x = x_conexion + (2 * S)

# Coordenada Y: Queremos que la parte superior (-S visual) toque la conexión (ins2.y = -S)
# Como ins2 está en Y = -S y su vértice derecho está en Y=0 relativo a él...
# Espera, el vértice derecho del rombo está en la misma Y que su centro (ins2.position.y).
# ins2.position.y es -S.
# La parte superior del rectángulo está en -S relativo a su centro (0).
# Por tanto, si ponemos el centro de ins3 en Y=0, su parte superior estará en -S.
ins3.position.y = 0

add_child(ins3)
```

Ejercicio 1.4.3. Implementa un proyecto Godot con una función `Tronco` que crea y devuelve un `Node2D` con dos nodos hijos que forman la figura de aquí abajo (uno para el relleno y otro para las

aristas).

Tabla de coordenadas:

0	$(+0{,}0, +0{,}0)$
1	$(+1{,}0, +0{,}0)$
2	$(+1{,}0, +1{,}0)$
3	$(+2{,}0, +2{,}0)$
4	$(+1{,}5, +2{,}5)$
5	$(+0{,}5, +1{,}5)$
6	$(+0{,}0, +3{,}0)$
7	$(-0{,}5, +3{,}0)$
8	$(+0{,}0, +1{,}5)$



Solución 1.4.3. Solución al problema 5.3:

```
# Problema 5.3:
# Implementa un proyecto Godot con una función Troncoque crea y devuelve
# un Node2D con dos nodos hijos que forman la figura de aquí abajo (uno para
# el relleno y otro para las aristas).

# PARTIMOS DE QUE EN 2D EL SENTIDO DA IGUAL, PERO SABEMOS QUE ES HORARIO

extends Node2D

func _ready():
  # Generamos el tronco
  var tronco = Tronco()

  # Lo añadimos a la escena
  add_child(tronco)

# ======================
# Función: Tronco
```

```
# Descripción: Crea un Node2D compuesto por dos mallas (relleno y borde)
# siguiendo la tabla de coordenadas de la página 60.
# ======================
func Tronco() -> Node2D:
  var n = Node2D.new()

  # 1. Definición de Vértices según la tabla del PDF
  var v0 = Vector2(0.0, 0.0)
  var v1 = Vector2(1.0, 0.0)
  var v2 = Vector2(1.0, 1.0)
  var v3 = Vector2(2.0, 2.0)
  var v4 = Vector2(1.5, 2.5)
  var v5 = Vector2(0.5, 1.5)   # El "entrepierna" o bifurcación
  var v6 = Vector2(0.0, 3.0)
  var v7 = Vector2(-0.5, 3.0)
  var v8 = Vector2(0.0, 1.5)

  # ------------------------
  # A. MALLA DE RELLENO (Triángulos)
  # ------------------------
  # Como el polígono es cóncavo, debemos triangularlo manualmente.
  # Dividimos la figura en 7 triángulos para cubrir toda la superficie.
  var vertices_relleno = PackedVector2Array([
    # Base del tronco (Cuadrilátero 0-1-2-8 dividido en dos)
    v0, v1, v2,
    v0, v2, v8,

    # Triángulo central de unión (conecta tronco con ramas)
    v8, v2, v5,

    # Rama Derecha (Cuadrilátero 2-3-4-5 dividido)
    v2, v3, v4,
    v2, v4, v5,

    # Rama Izquierda (Cuadrilátero 5-6-7-8 dividido)
    v5, v6, v7,
    v5, v7, v8
  ])

  # Usamos la función local auxiliar para crear malla de tipo TRIANGLES
  var malla_relleno = FuncionesAuxiliaresT5._crear_malla_rellena(vertices_relleno)

  # Instanciamos usando la función del autoload y asignamos
```

```
    color lavanda/azul claro
62  var inst_relleno = FuncionesAuxiliaresT5.CrearMeshInstance2D(
     malla_relleno, Transform2D())
63  inst_relleno.modulate = Color(0.7, 0.7, 1.0)
64  n.add_child(inst_relleno)
65
66  # ----------------------
67  # B. MALLA DE BORDE (Línea)
68  # ----------------------
69  # Recorremos el perímetro exterior en orden
70  # var vertices_borde = PackedVector2Array([
71  #   v0, v1, v2, v3, v4, v5, v6, v7, v8s # Cerramos volviendo a
      v0
72  # ])
73
74  var vertices_borde = PackedVector2Array([
75    v1, v2,  # Lado derecho tronco
76    v2, v3,  # Lado derecho rama derecha
77    # Saltamos 3-4 (punta derecha)
78    v4, v5,  # Lado izquierdo rama derecha (interior V)
79    v5, v6,  # Lado derecho rama izquierda (interior V)
80    # Saltamos 6-7 (punta izquierda)
81    v7, v8,  # Lado izquierdo rama izquierda
82    v8, v0   # Lado izquierdo tronco
83    # Saltamos 0-1 (base+)
84  ])
85
86  # Usamos la función del autoload (genera LINE SIN STRIP ya que
      este los conecta todos)
87  var malla_borde = FuncionesAuxiliaresT5.
     _crear_malla_lineas_pares(vertices_borde)
88
89  # Instanciamos y asignamos color azul oscuro
90  var inst_borde = FuncionesAuxiliaresT5.CrearMeshInstance2D(
     malla_borde, Transform2D())
91  inst_borde.modulate = Color(0.0, 0.0, 0.0)
92  n.add_child(inst_borde)
93
94  return n
```

Ejercicio 1.4.4. Implementa otro proyecto Godot que use la función del problema anterior para otra función, `Arbol(n)`, que genera un árbol de escena con la figura de aquí abajo, que incluye múltiples instancias de `Tronco`, situadas recursivamente unas adyacentes a otras, hasta un nivel de recursividad dado por n.

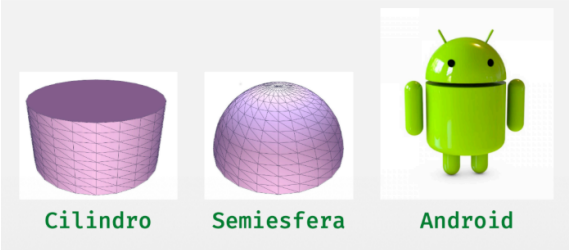
Solución 1.4.4. Solución al problema 5.4:

```
# Problema 5.4:
# Implementa otro proyecto Godot que use la función del problema anterior
# para otra función, Arbol(n) que genera un árbol de escena con la figura
# de aquí abajo, que incluye múltiples instancias de Tronco, situadas recursi3
# vamente unas adyacentes a otras, hasta un nivel de recursividad dado por n.

extends Node2D

# Configuración del árbol
var niveles_recursividad : int = 7

func _ready():
  # Generamos el árbol recursivo
  var arbol = Arbol(niveles_recursividad)
  add_child(arbol)

# ==================
# FUNCIÓN RECURSIVA: Arbol(n)
# Crea un nodo tronco y, si n > 0, le añade dos árboles más pequeños (n-1)
# en las puntas, transformados geométricamente para encajar.
# ==================
func Arbol(n: int) -> Node2D:
  # 1. Creamos la geometría de este nivel (el tronco base)
  var nodo_actual = Tronco()

  # 2. Caso Base: Si n es 0, terminamos aquí (solo devolvemos el tronco)
  if n <= 0:
    return nodo_actual

  # 3. Caso Recursivo: Crear ramas hijas

  # --- RAMA IZQUIERDA ---
  # Debe encajar en el segmento superior izquierdo (v7 -> v6) del padre.
  var hijo_izq = Arbol(n - 1) # Recursión
  hijo_izq.position = Vector2(-0.5, 3.0) # Vértice 7 del padre
  hijo_izq.scale = Vector2(0.5, 0.5)     # Reduce al 50%, ya que el ancho de 6-7 es 0.5 cuando la base del hijo mide 1
  hijo_izq.rotation = 0                  # Sin rotación (alineado con X)
  nodo_actual.add_child(hijo_izq)

```

```
  # --- RAMA DERECHA ---
  # Debe encajar en el segmento superior derecho (v4 -> v3) del padre.
  var hijo_der = Arbol(n - 1) # Recursión
  hijo_der.position = Vector2(1.5, 2.5)  # Vértice 4 del padre
  hijo_der.scale = Vector2(0.707, 0.707) # 1 / sqrt(2) approx, ya que la distancia entre los vértices 3-4 es su módulo, es decir, la diferencia de ambos puntos al cuadrado y sumada, luego aplicamos la raíz para saber el factor de escala
  hijo_der.rotation_degrees = -45        # Rotar -45 para encajar, ya que vamos bajando y la base es horizontal
  nodo_actual.add_child(hijo_der)

  return nodo_actual

# ==================
# GEOMETRÍA DEL TRONCO (Del ejercicio 5.3, con tapas abiertas)
# ==================
func Tronco() -> Node2D:
  var n = Node2D.new()

  # Vértices (Tabla Pág. 60)
  var v0 = Vector2(0.0, 0.0); var v1 = Vector2(1.0, 0.0)
  var v2 = Vector2(1.0, 1.0); var v3 = Vector2(2.0, 2.0)
  var v4 = Vector2(1.5, 2.5); var v5 = Vector2(0.5, 1.5)
  var v6 = Vector2(0.0, 3.0); var v7 = Vector2(-0.5, 3.0)
  var v8 = Vector2(0.0, 1.5)

  # --- RELLENO (Triángulos) ---
  var vertices_relleno = PackedVector2Array([
    v0, v1, v2, v0, v2, v8, # Tronco
    v8, v2, v5,             # Centro
    v2, v3, v4, v2, v4, v5, # Rama Der
    v5, v6, v7, v5, v7, v8  # Rama Izq
  ])
  var m_relleno = _crear_malla_triangulos(vertices_relleno)
  var inst_relleno = FuncionesAuxiliaresT5.CrearMeshInstance2D(m_relleno, Transform2D())
  inst_relleno.modulate = Color(0.7, 0.7, 1.0)
  n.add_child(inst_relleno)

  # --- BORDE (Líneas Discontinuas) ---
  # NO incluimos las tapas (0-1, 3-4, 6-7) para que la recursión fluya visualmente
  var vertices_borde = PackedVector2Array([
    v1, v2, v2, v3, # Lado derecho
    v4, v5, v5, v6, # Interior V
    v7, v8, v8, v0  # Lado izquierdo
```

```
81    ])
82    var m_borde = _crear_malla_segmentos(vertices_borde)
83    var inst_borde = FuncionesAuxiliaresT5.CrearMeshInstance2D(
        m_borde, Transform2D())
84    inst_borde.modulate = Color(0.0, 0.0, 0.8)
85    n.add_child(inst_borde)
86
87    return n
88
89  # ==================
90  # HELPERS LOCALES (las dejamos aquí para que a la hora de
        estudiar poder tenerlas más a mano)
91  # ==================
92  func _crear_malla_triangulos(v: PackedVector2Array) -> ArrayMesh
        :
93    var tablas: Array = []; tablas.resize(Mesh.ARRAY_MAX)
94    tablas[Mesh.ARRAY_VERTEX] = v
95    var am = ArrayMesh.new()
96    am.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES, tablas)
97    return am
98
99  func _crear_malla_segmentos(v: PackedVector2Array) -> ArrayMesh:
100   var tablas: Array = []; tablas.resize(Mesh.ARRAY_MAX)
101   tablas[Mesh.ARRAY_VERTEX] = v
102   var am = ArrayMesh.new()
103   am.add_surface_from_arrays(Mesh.PRIMITIVE_LINES, tablas)
104   return am
```

Ejercicio 1.4.5. En un proyecto Godot 3D (puedes usar la práctica 2), crea una figura como el logo de Android, usando únicamente dos objetos `ArrayMesh`, uno con un cilindro y otro con una semiesfera.



Solución 1.4.5. Solución al problema 5.5:

```
# Problema 5.5:
# En un proyecto Godot 3D (puedes usar la práctica 2) para crear una figura
# como el logo de Android, usando únicamente dos objetos ArrayMesh, uno
# con un cilindro y otro con una semiesfera.

extends Node3D

# Variables para almacenar las dos únicas mallas permitidas
var malla_cilindro: ArrayMesh
var malla_semiesfera: ArrayMesh

# Materiales
var material_verde: StandardMaterial3D
var material_negro: StandardMaterial3D

func _ready():
  # 1. Crear los recursos (Materiales y Mallas)
  _crear_materiales()
  _generar_mallas()

  # 2. Construir la jerarquía (Ensamblaje)
  construir_android()

func _crear_materiales():
  # Verde corporativo de Android
  material_verde = StandardMaterial3D.new()
  material_verde.albedo_color = Color(0.24, 0.86, 0.35) # Aprox Android Green

  # Negro para los ojos
  material_negro = StandardMaterial3D.new()
  material_negro.albedo_color = Color.BLACK

func _generar_mallas():
  # --- Generar Cilindro ---
  # Usamos primitivas de Godot para simplificar el código del
```

```
    ejemplo,
36  # pero conceptualmente es un ArrayMesh generado.
37  var cilindro = CylinderMesh.new()
38  cilindro.top_radius = 0.5
39  cilindro.bottom_radius = 0.5
40  cilindro.height = 1.0
41  # Convertimos a ArrayMesh para cumplir estrictamente el
    enunciado
42  # que pide "objetos ArrayMesh" (aunque PrimitiveMesh hereda de
    Mesh).
43  malla_cilindro = ArrayMesh.new()
44  malla_cilindro.add_surface_from_arrays(Mesh.
    PRIMITIVE_TRIANGLES, cilindro.get_mesh_arrays())
45
46  # --- Generar Semiesfera ---
47  var esfera = SphereMesh.new()
48  esfera.radius = 0.5
49  esfera.height = 0.5 # Hacemos que sea media esfera
50  esfera.is_hemisphere = true # Propiedad específica para
    semiesfera
51  malla_semiesfera = ArrayMesh.new()
52  malla_semiesfera.add_surface_from_arrays(Mesh.
    PRIMITIVE_TRIANGLES, esfera.get_mesh_arrays())
53
54 func construir_android():
55  # --- CUERPO (Cilindro) ---
56  var cuerpo = MeshInstance3D.new()
57  cuerpo.mesh = malla_cilindro
58  cuerpo.material_override = material_verde
59  # El cilindro por defecto tiene altura 1. Lo escalamos para
    que sea el cuerpo.
60  cuerpo.scale = Vector3(1.5, 1.5, 1.5)
61  cuerpo.position = Vector3(0, 0.75, 0) # Subirlo un poco
62  add_child(cuerpo)
63
64  # --- CABEZA (Semiesfera) ---
65  var cabeza = MeshInstance3D.new()
66  cabeza.mesh = malla_semiesfera
67  cabeza.material_override = material_verde
68  cabeza.scale = Vector3(1.5, 1.5, 1.5) # Misma escala X/Z que
    el cuerpo
69  cabeza.position = Vector3(0, 1.6, 0) # Sobre el cuerpo
70  add_child(cabeza)
71
72  # --- OJOS (Cilindros transformados) ---
73  # Ojo Izquierdo
74  var ojo_izq = MeshInstance3D.new()
75  ojo_izq.mesh = malla_cilindro
```

```
76  ojo_izq.material_override = material_negro
77  # Transformación clave: Escalar el cilindro para que parezca
     un disco plano
78  ojo_izq.scale = Vector3(0.1, 0.02, 0.1)
79  # Rotarlo para que mire al frente (El cilindro está orientado
     en Y por defecto)
80  ojo_izq.rotation_degrees.x = 90
81  ojo_izq.position = Vector3(-0.25, 0.25, 0.45) # Posición
     relativa a la cabeza
82  cabeza.add_child(ojo_izq) # ¡Hijo de la cabeza!
83
84  # Ojo Derecho (Instancia idéntica, distinta posición)
85  var ojo_der = MeshInstance3D.new()
86  ojo_der.mesh = malla_cilindro
87  ojo_der.material_override = material_negro
88  ojo_der.scale = Vector3(0.1, 0.02, 0.1)
89  ojo_der.rotation_degrees.x = 90
90  ojo_der.position = Vector3(0.25, 0.25, 0.45)
91  cabeza.add_child(ojo_der)
92
93  # --- ANTENAS (Cilindros finos) ---
94  var antena_izq = MeshInstance3D.new()
95  antena_izq.mesh = malla_cilindro
96  antena_izq.material_override = material_verde
97  antena_izq.scale = Vector3(0.05, 0.4, 0.05) # Fina y larga
98  antena_izq.position = Vector3(-0.3, 0.4, 0)
99  antena_izq.rotation_degrees.z = 30 # Inclinación
100 cabeza.add_child(antena_izq)
101
102 var antena_der = MeshInstance3D.new()
103 antena_der.mesh = malla_cilindro
104 antena_der.material_override = material_verde
105 antena_der.scale = Vector3(0.05, 0.4, 0.05)
106 antena_der.position = Vector3(0.3, 0.4, 0)
107 antena_der.rotation_degrees.z = -30
108 cabeza.add_child(antena_der)
109
110 # --- EXTREMIDADES (Cilindros) ---
111 # Aquí aplicamos lo aprendido en Session 5: Reutilización
112
113 # Brazo Izquierdo
114 var brazo_izq = MeshInstance3D.new()
115 brazo_izq.mesh = malla_cilindro
116 brazo_izq.material_override = material_verde
117 brazo_izq.scale = Vector3(0.3, 1.0, 0.3)
118 brazo_izq.position = Vector3(-0.9, 0.6, 0) # Al lado del
     cuerpo
119 add_child(brazo_izq)
```

```
120
121  # Brazo Derecho
122  var brazo_der = MeshInstance3D.new()
123  brazo_der.mesh = malla_cilindro
124  brazo_der.material_override = material_verde
125  brazo_der.scale = Vector3(0.3, 1.0, 0.3)
126  brazo_der.position = Vector3(0.9, 0.6, 0)
127  add_child(brazo_der)
128
129  # Pierna Izquierda
130  var pierna_izq = MeshInstance3D.new()
131  pierna_izq.mesh = malla_cilindro
132  pierna_izq.material_override = material_verde
133  pierna_izq.scale = Vector3(0.3, 0.6, 0.3)
134  pierna_izq.position = Vector3(-0.4, -0.4, 0) # Debajo del
      cuerpo
135  add_child(pierna_izq)
136
137  # Pierna Derecha
138  var pierna_der = MeshInstance3D.new()
139  pierna_der.mesh = malla_cilindro
140  pierna_der.material_override = material_verde
141  pierna_der.scale = Vector3(0.3, 0.6, 0.3)
142  pierna_der.position = Vector3(0.4, -0.4, 0)
143  add_child(pierna_der)
```

1.5 Sesión 6

Ejercicio 1.5.1. Escribe el código GDScript para adjuntar a un nodo de tipo Camera3D, de forma que en cada frame la cámara apunte a un objeto móvil objetivo (por ejemplo un coche), con estos requerimientos:

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

Solución 1.5.1. Nuestro objetivo móvil va a ser un coche. La resolución detallada es la siguiente:

Requerimientos Geométricos:

1) **Punto de Atención (Look At):** La cámara debe apuntar al objetivo. Esto significa que el eje $-Z$ de la cámara (en Godot, la cámara "mira" hacia $-Z$ local) debe alinearse con el vector que va desde la cámara hasta el objetivo. El punto $\vec{p}_{obj}$ se proyectará en el centro del

viewport.

2) **Posición Relativa:**
   - **"Detrás" (Horizontal):** 3 unidades por detrás. "Detrás" se define en relación con el movimiento. Si el coche se mueve hacia adelante, "detrás" es la dirección opuesta a la velocidad. Debemos considerar solo la componente horizontal para evitar que la cámara se incline hacia el suelo si el coche sube una pendiente.
   - **"Arriba" (Vertical):** 2 unidades por encima del objetivo (eje Y global).

Fundamentación Teórica

Para resolver esto, utilizamos conceptos de **Espacios Afines** y **Operaciones con Vectores** (tratados en el pdf `ig-s03.pdf`):

1) **Definición de "Atrás":** El vector velocidad $\vec{v}_{obj}$ nos da la dirección del movimiento. Para situarnos "detrás" horizontalmente:
   - Tomamos $\vec{v}_{obj}$ y anulamos su componente Y (para que sea puramente horizontal): $\vec{d}_{hz} = (v_x, 0, v_z)$.
   - Normalizamos este vector para obtener una dirección unitaria: $\hat{d}_{hz} = \vec{d}_{hz}/|\vec{d}_{hz}|$.
   - El vector "hacia atrás" es $-\hat{d}_{hz}$.
   - El desplazamiento horizontal deseado es $-3 \cdot \hat{d}_{hz}$.
2) **Composición de la Posición de la Cámara ($\vec{p}_{cam}$):**

$$\vec{p}_{cam} = \vec{p}_{obj} + (0, 2, 0) - 3 \cdot \hat{d}_{hz}$$

   Donde:
   - $(0, 2, 0)$: 2 unidades arriba.
   - $-3 \cdot \hat{d}_{hz}$: 3 unidades atrás.
3) **Transformación de Vista (LookAt):** Una vez tenemos $\vec{p}_{cam}$, necesitamos construir la matriz de vista. En Godot, la clase `Node3D` (de la cual hereda `Camera3D`) tiene métodos auxiliares para esto. El método `look_at(target, up)` ajusta la transformación del nodo para que mire a `target` manteniendo el vector `up` orientado hacia arriba tanto como sea posible.

Solución: Código GDScript

```
1  extends Camera3D
2
3  # Asumimos que 'objetivo' es un singleton (AutoLoad) o una clase
       global accesible.
4  # Si no fuera global, habría que obtener la referencia al nodo (
       ej. get_node(''../Coche''))
5
6  func _process(delta: float):
7      # 1. Obtener datos del objetivo (en coordenadas de mundo)
8      # Según el enunciado, existen estas funciones globales.
9      var p_obj: Vector3 = objetivo.posicion()
10     var v_obj: Vector3 = objetivo.velocidad()
11
```

```
    # 2. Calcular la dirección horizontal del movimiento
    # Creamos un vector con la velocidad pero ignorando la componente Y
    var direccion_hz: Vector3 = Vector3(v_obj.x, 0.0, v_obj.z)

    # IMPORTANTE: Si el coche está parado (velocidad casi 0), no podemos normalizar
    # (división por cero). En un caso real, mantendríamos la última dirección válida.
    # Para el ejercicio, asumimos movimiento o usamos una dirección por defecto (ej. eje Z).
    if direccion_hz.length_squared() > 0.001:
        direccion_hz = direccion_hz.normalized()
    else:
        # Fallback: si está quieto, asumimos que ''detrás'' es el eje Z positivo (por ejemplo)
        # O idealmente, usaríamos la orientación del nodo objetivo (basis.z)
        direccion_hz = Vector3(0, 0, 1)

    # 3. Calcular la posición deseada de la cámara
    # - Situada en la posición del objetivo
    # - Desplazada 2 unidades hacia ARRIBA (Eje Y global)
    # - Desplazada 3 unidades hacia ATRÁS (opuesto a la dirección horizontal)
    var nueva_posicion: Vector3 = p_obj + Vector3(0, 2, 0) - (direccion_hz * 3.0)

    # 4. Aplicar la posición a la cámara
    # Usamos global_position para asegurar que estamos en coords de mundo
    global_position = nueva_posicion

    # 5. Orientar la cámara (Transformación de Vista)
    # Hacemos que la cámara mire al punto objetivo.
    # El vector ''Arriba'' (Up) suele ser el eje Y global (Vector3.UP)
    look_at(p_obj, Vector3.UP)
```

Explicación detallada de la implementación

1) **`extends Camera3D`**: El script hereda de la clase base de cámaras en Godot, permitiendo controlar la proyección y vista.
2) **`_process(delta)`**: Usamos este método del bucle principal (`MainLoop`) porque el enunciado pide que la cámara se actualice ”en cada frame”.
3) **Cálculo del vector dirección**:

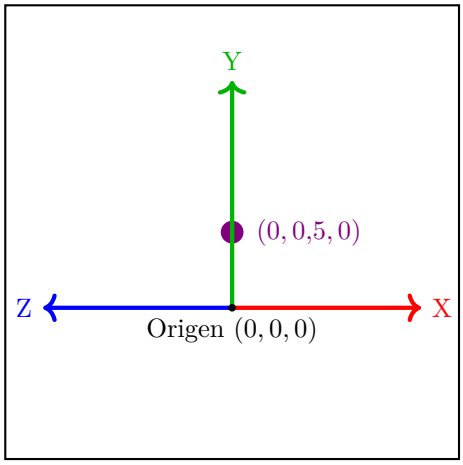
- El enunciado especifica ”3 unidades en horizontal”. Esto es crucial. Si usáramos el vector velocidad completo (incluyendo Y) para calcular el ”atrás”, y el coche subiera una rampa muy empinada, la cámara se metería bajo tierra. Por eso proyectamos sobre el plano XZ haciendo $v_{obj}.y = 0$ y luego normalizamos $v.normalized()$.

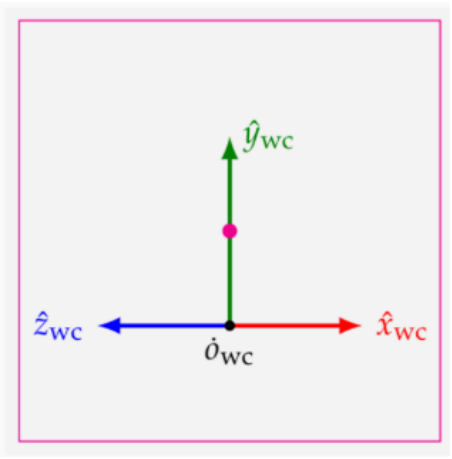
4) **Posicionamiento (`global_position`)**:
   - Calculamos la posición final sumando vectores. Matemáticamente: $\vec{p}_{cam} = \vec{p}_{obj} + (0, 2, 0) - 3 \cdot \hat{d}_{hz}$.
5) **Orientación (`look_at`)**:
   - Este método es fundamental en la **Transformación de Vista**. Recalcula la matriz de transformación del nodo (`transform`) para que su eje $-Z$ (visión) apunte a $\vec{p}_{obj}$ y su eje Y se alinee con `Vector3.UP`. Esto resuelve la parte compleja de crear la matriz de rotación ortonormal manualmente.

Ejercicio 1.5.2. Supongamos una escena que contiene una representación visible del marco de coordenadas del mundo como tres flechas (roja X, verde Y y azul Z), como ocurre en las prácticas. Queremos visualizar esa escena en pantalla, de forma que:

1) El eje Y aparezca vertical, hacia arriba, el eje X horizontal, hacia la derecha, el eje Z horizontal, hacia la izquierda (los ejes X y Z se visualizan con la misma longitud aparente).
2) El punto de coordenadas $(0, 0{,}5, 0)$ (aparece como un disco de color morado en la figura) debe aparecer en el centro del viewport.
3) El observador (foco de la proyección) estará a 3 unidades de distancia del punto $(0, 0{,}5, 0)$.

Escribe unos valores que podríamos usar para **a**, **u** y **n** de forma que se cumplan estos requisitos. En la figura se observa una vista esquemática de cómo quedaría la figura en un viewport cuadrado.





Solución 1.5.2. Para determinar los parámetros de la matriz de vista (**a**, **u**, **n**), analizamos cada requerimiento paso a paso:

1) **Determinación del punto de atención (a):** El enunciado establece que el punto de coordenadas $(0, 0{,}5, 0)$ debe aparecer en el centro del viewport. Por definición, el punto de atención **a** (Look-At point) es el punto hacia el que apunta la cámara y que se proyecta en el centro del plano de imagen.
   Por lo tanto:
   $$\mathbf{a} = (0, 0{,}5, 0)$$
2) **Determinación del vector hacia arriba (u):** Se requiere que el eje Y del mundo aparezca vertical y hacia arriba en la imagen. Dado que el eje Y del mundo es $(0, 1, 0)$, la forma más

directa de conseguir que se proyecte verticalmente es alineando el vector *view-up* (**u**) con el eje Y del mundo (siempre que la dirección de vista no sea paralela a este eje, lo cual verificaremos en el siguiente paso).
Por lo tanto:

$$\mathbf{u} = (0, 1, 0)$$

3) **Determinación del vector normal de vista (n):** El vector **n** define la dirección desde el punto de atención hacia el observador (es decir, la inversa de la dirección de la vista). También determina la posición del observador $\mathbf{o}_{ec}$ mediante la relación $\mathbf{o}_{ec} = \mathbf{a} + \mathbf{n}$.
Analizamos las condiciones para $\mathbf{n} = (n_x, n_y, n_z)$:
   - **Longitud:** El observador debe estar a 3 unidades de distancia de **a**. Como **n** es el vector que une **a** con el observador, su norma debe ser 3:

$$||\mathbf{n}|| = 3$$

   - **Orientación Horizontal:** Para que el eje Y se vea perfectamente vertical y centrado, la cámara debe estar a la misma altura o el vector de visión debe estar contenido en un plano vertical que contenga al eje Y. Sin embargo, la condición crítica proviene de los ejes X y Z.
   - **Orientación de X y Z:**
     - El eje X debe verse horizontal hacia la derecha.
     - El eje Z debe verse horizontal hacia la izquierda.
     - Ambos deben tener la misma longitud aparente.

     Esto implica que el observador debe situarse en una posición simétrica respecto a los ejes X e Z positivos (primer cuadrante del plano XZ respecto a **a**), de forma que la línea de visión biseque el ángulo de 90 grados entre X y Z.
     Si nos situamos en la bisectriz del primer cuadrante del plano XZ, el vector de dirección tendrá componentes X y Z iguales y positivas. El eje X (derecha) y el eje Z (adelante) formarán ambos un ángulo de 45° con el plano de proyección, proyectándose hacia lados opuestos (derecha e izquierda) con la misma deformación (longitud aparente).
     Por tanto, la dirección de **n** debe ser $(1, 0, 1)$.

   Calculamos **n**:
   1) Tomamos el vector director base: $\vec{d} = (1, 0, 1)$.
   2) Calculamos su norma: $||\vec{d}|| = \sqrt{1^2 + 0^2 + 1^2} = \sqrt{2}$.
   3) Normalizamos y escalamos por la distancia requerida (3 unidades):

$$\mathbf{n} = 3 \cdot \frac{\vec{d}}{||\vec{d}||} = 3 \cdot \frac{(1, 0, 1)}{\sqrt{2}} = \left(\frac{3}{\sqrt{2}}, 0, \frac{3}{\sqrt{2}}\right)$$

   Aproximando los valores:

$$\frac{3}{\sqrt{2}} = \frac{3\sqrt{2}}{2} \approx 2{,}1213$$

   Así, $\mathbf{n} \approx (2{,}12, 0, 2{,}12)$.

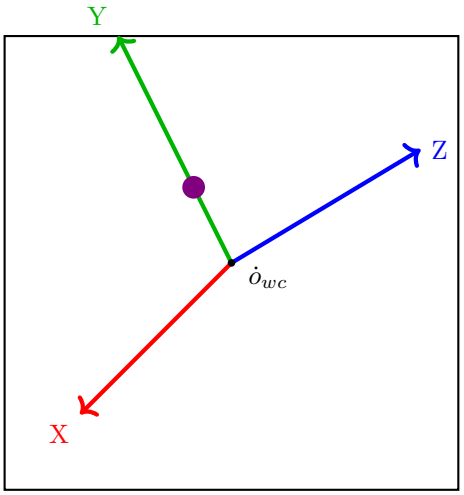
Resultado Final: Los valores que cumplen los requisitos son:

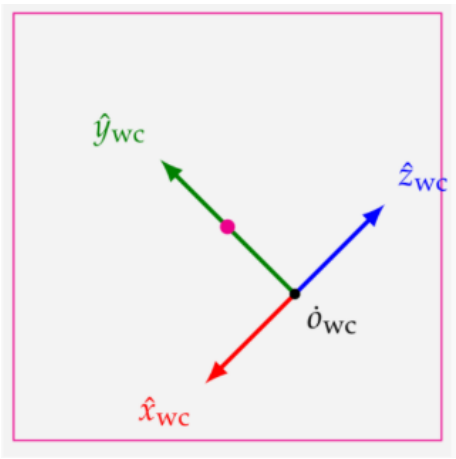
$$\mathbf{a} = (0, 0{,}5, 0)$$

$$\mathbf{u} = (0, 1, 0)$$

$$\mathbf{n} = \left(\frac{3}{\sqrt{2}}, 0, \frac{3}{\sqrt{2}}\right)$$

Ejercicio 1.5.3. Repite el problema anterior 6.2, pero ahora para esta vista (ver figura). Usa una rotación del marco de vista entorno a uno de sus propios ejes.





Escribe los valores para **a**, **u** y **n**.

Solución 1.5.3. Para obtener la configuración visual mostrada en la figura, partimos de la solución del ejercicio 6.2 y aplicamos las transformaciones necesarias.

1) **Punto de atención (a):** Al igual que en el ejercicio anterior, el punto $(0,0{,}5,0)$ (disco morado) debe aparecer en el centro del viewport. Por tanto:

$$\mathbf{a} = (0,0{,}5,0)$$

2) **Vector normal de vista (n):** Observamos la orientación de los ejes X y Z:
   - El eje X (rojo) apunta hacia la izquierda y abajo.
   - El eje Z (azul) apunta hacia la derecha y arriba.

   En el ejercicio 6.2, mirábamos desde el primer cuadrante $(+X, +Z)$, viendo el eje X a la derecha y Z a la izquierda. Aquí la situación horizontal se ha invertido (X a la izquierda, Z a la derecha), lo que implica que el observador se ha movido a la posición opuesta ("detrás" de la escena), mirando desde el cuadrante $(-X, -Z)$.
   El vector de dirección base sería $(-1, 0, -1)$. Normalizando y aplicando la distancia de 3 unidades:

$$\mathbf{n} = 3 \cdot \frac{(-1,0,-1)}{\sqrt{(-1)^2 + 0^2 + (-1)^2}} = 3 \cdot \left(\frac{-1}{\sqrt{2}}, 0, \frac{-1}{\sqrt{2}}\right)$$

   Aproximando:

$$\mathbf{n} \approx (-2{,}12, 0, -2{,}12)$$

3) **Vector hacia arriba (u):** Observamos el eje Y (verde). En lugar de apuntar verticalmente hacia arriba (como haría con $\mathbf{u} = (0,1,0)$), apunta hacia arriba a la izquierda. Esto indica una rotación de la cámara (Roll) alrededor del eje de visión **n**.
   Si usáramos $\mathbf{u}_{base} = (0,1,0)$ desde la posición trasera, veríamos el eje Y vertical. Para que el eje Y se incline hacia la izquierda en la pantalla, la cámara debe rotar en sentido horario (CW). Una rotación de 45 grados en sentido horario del vector $\mathbf{u}_{base}$ alrededor del eje de visión nos da el vector necesario.

Calculamos **u** como una combinación lineal que se incline hacia el eje Z negativo y X negativo (para mantener la ortogonalidad con **n**):

$$\mathbf{u} = (-1, 1, 1)$$

(Nota: Se puede normalizar a $(-1/\sqrt{3}, 1/\sqrt{3}, 1/\sqrt{3})$).
Verificación rápida: $\mathbf{n} \cdot \mathbf{u} = (-1)(-1) + (0)(1) + (-1)(1) = 1 + 0 - 1 = 0$. Son perpendiculares.

¿Cómo se calcula u exactamente?
El vector **u** (View-Up) indica la dirección de "arriba" para la cámara. El procedimiento ordenado para deducir $\mathbf{u} = (-1, 1, 1)$ es:

1) **Definir la base sin rotar:**
   - Nos situamos "detrás" de la escena (lado opuesto al ejercicio 6.2), ya que el eje X va a la izquierda y el Z a la derecha.
   - Vector de vista ideal: $\mathbf{n} = (-1, 0, -1)$.
   - Vector arriba estándar: $\mathbf{u}_{base} = (0, 1, 0)$.
2) **Calcular el vector "Derecha":**

$$\text{Derecha} = \mathbf{u}_{base} \times \mathbf{n} = (0, 1, 0) \times (-1, 0, -1) = (-1, 0, 1)$$

   Sabemos que Arriba × Atrás = Derecha. Para el caso de la izquierda sería el opuesto. (n es atrás y u es arriba).
3) **Aplicar la rotación (mezclar arriba y derecha):**
   - Para rotar la cámara hacia la derecha (sentido horario), sumamos el vector arriba original y el vector derecha:

$$\mathbf{u} = \mathbf{u}_{base} + \text{Derecha} = (0, 1, 0) + (-1, 0, 1) = (-1, 1, 1)$$

   - Este vector tiene componente en Y (arriba), pero también en X y Z, inclinando el "arriba" de la cámara hacia la derecha de la pantalla, logrando el efecto de rotación deseado.

Valores Finales:

$$\mathbf{a} = (0,0{,}5, 0)$$

$$\mathbf{u} = (-1, 1, 1) \quad (\text{o normalizado } \approx (-0{,}577, 0{,}577, 0{,}577))$$

$$\mathbf{n} = \left(\frac{-3}{\sqrt{2}}, 0, \frac{-3}{\sqrt{2}}\right) \approx (-2{,}12, 0, -2{,}12)$$

Ejercicio 1.5.4. Escribe el código GDScript para calcular los vectores de coordenadas o_{ec}, x_{ec}, y_{ec} y z_{ec} que definen el marco de vista a partir de los vectores de coordenadas a, u y n (todos estos vectores de coordenadas de mundo, en objetos de tipo Vector3).

Solución 1.5.4. Para construir el marco de referencia de vista (view reference frame) a partir de los vectores dados, seguimos el procedimiento estándar de la transformación de cámara en gráficos 3D:

1) **Cálculo del origen del marco (o_{ec}):** El origen del marco de cámara (posición del observador) se obtiene sumando el punto de atención a y el vector normal n:

$$o_{ec} = a + n$$

2) **Cálculo del eje z_{ec}:** El eje z_{ec} es la dirección de la vista (normalizada) y se obtiene normalizando el vector n:
$$z_{ec} = \frac{n}{\|n\|}$$

3) **Cálculo del eje x_{ec}:** El eje x_{ec} (derecha de la cámara) se obtiene como el producto vectorial entre el vector hacia arriba u y el vector normal n, normalizado:
$$x_{ec} = \frac{u \times n}{\|u \times n\|}$$

4) **Cálculo del eje y_{ec}:** El eje y_{ec} (arriba de la cámara) se obtiene como el producto vectorial entre z_{ec} y x_{ec}:
$$y_{ec} = z_{ec} \times x_{ec}$$

El siguiente código GDScript implementa estos pasos, suponiendo que a, u y n son objetos de tipo `Vector3`:

```
# a, u, n: Vector3 (coordenadas de mundo)

# 1. Origen del marco de cámara
var o_ec : Vector3 = a + n

# 2. Eje Z (dirección de la vista, normalizado)
var z_ec : Vector3 = n.normalized()

# 3. Eje X (derecha, ortogonal a u y n, normalizado)
var x_ec : Vector3 = u.cross(n).normalized()

# 4. Eje Y (arriba, ortogonal a z_ec y x_ec)
var y_ec : Vector3 = z_ec.cross(x_ec)
```

Este procedimiento garantiza que los vectores x_{ec}, y_{ec} y z_{ec} forman una base ortonormal adecuada para definir el sistema de referencia de la cámara.

Ejercicio 1.5.5. Partiendo de los vectores de coordenadas o_{ec}, x_{ec}, y_{ec} y z_{ec} que se calculan en el problema anterior, escribe el código que calcula explícitamente la matriz de vista, es una variable de tipo `Transform3D`.

Solución 1.5.5. Para construir la matriz de vista (*View Matrix*) a partir del marco de cámara definido por o_{ec} (origen), x_{ec}, y_{ec} y z_{ec} (vectores ortonormales), seguimos el procedimiento estándar de gráficos 3D:

1) **Definición:** La matriz de vista transforma coordenadas del mundo al sistema de la cámara. Se compone de una rotación (alineando los ejes del mundo con los de la cámara) y una traslación (llevando el origen de la cámara al origen del sistema).

2) **Expresión matricial:**

$$V = \begin{pmatrix} x_{ec}.x & x_{ec}.y & x_{ec}.z & -(x_{ec} \cdot o_{ec}) \\ y_{ec}.x & y_{ec}.y & y_{ec}.z & -(y_{ec} \cdot o_{ec}) \\ z_{ec}.x & z_{ec}.y & z_{ec}.z & -(z_{ec} \cdot o_{ec}) \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3) **Implementación en Godot (`Transform3D`):** En Godot, la clase `Transform3D` almacena la base (rotación) y el origen (traslación). La base se define por columnas, por lo que debemos transponer la matriz formada por x_{ec}, y_{ec} y z_{ec} como filas.

Código GDScript:

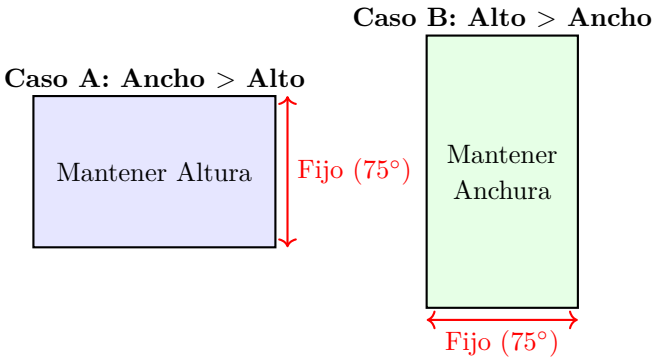
```
# Suponemos disponibles: x_ec, y_ec, z_ec, o_ec (Vector3)

# 1. Construir la base (rotación): columnas de la base son los ejes de cámara
var R := Basis(x_ec, y_ec, z_ec)
var vista_basis := R.transposed()

# 2. Calcular la traslación (origen) según la fórmula de la matriz de vista
var d_x = -x_ec.dot(o_ec)
var d_y = -y_ec.dot(o_ec)
var d_z = -z_ec.dot(o_ec)
var vista_origin = Vector3(d_x, d_y, d_z)

# 3. Construir la matriz de vista final
var matriz_vista = Transform3D(vista_basis, vista_origin)

# La función de Transform3D lo que es empaqueta todo en un solo objeto, en este caso, lo que hace es crear una matriz de 4x4 a partir de una matriz de 3x3 (Basis) y un vector de traslación (origin).
```

Explicación: La matriz de vista es la inversa de la transformación de la cámara en el mundo. La base ortonormal se transpone para invertir la rotación, y la traslación se obtiene proyectando el origen del marco de cámara sobre cada eje y cambiando el signo, lo que equivale a trasladar el mundo al sistema de la cámara. Usamos cross para construir el marco de referencia, y dot para situar puntos dentro de ese marco, en este caso como lo que se busca es proyectar usamos dot.

Ejercicio 1.5.6. En una copia independiente del código de prácticas, modifica el nodo de la cámara orbital simple para conseguir que el fov mínimo (vertical u horizontal) sea siempre de 75°. Esto servirá, por ejemplo, para ver el cubo de las prácticas siempre completo independientemente del ancho y alto de la ventana.

Para ello:

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



Solución 1.5.6. Para resolver este problema, debemos manipular la propiedad `keep_aspect` de la clase `Camera3D` en Godot. Esta propiedad determina qué eje (horizontal o vertical) mantiene el ángulo de visión (`fov`) fijo cuando cambia la relación de aspecto de la ventana.

El objetivo es asegurar que el objeto siempre sea visible. Si la ventana se estrecha horizontalmente, debemos fijar el FOV horizontal. Si se estrecha verticalmente, debemos fijar el FOV vertical.

1) **Lógica del algoritmo:**
   - Obtenemos el tamaño del viewport: w (ancho) y h (alto).
   - Calculamos la relación de aspecto $r = w/h$.
   - Si $r \geq 1$ (formato apaisado o cuadrado): El ancho es suficiente para contener la escena si fijamos la altura. Usamos `KEEP_HEIGHT`.
   - Si $r < 1$ (formato vertical o "retrato"): El ancho es el factor limitante. Para evitar que se recorte la escena lateralmente, debemos fijar el ángulo horizontal. Usamos `KEEP_WIDTH`.
2) **Implementación en GDScript:** Añadimos la función `_actualiza_proyeccion` y la conectamos a la señal `size_changed` del viewport raíz en la función `_ready`.

```
extends Camera3D

# -------------
# constantes y variables de instancia

const at   := 2.5   # angulo de rot. con teclas
const ar   := 0.5   # angulo de rot. con raton
var   bdrp := false # boton derecho del raton presionado si
   /no
var   dz   := 3.0   # distancia en Z de la camara al origen
var   dxy  := Vector2( 0.0, 0.0 ) # angulos hor. y vert.

# -------------
# actualiza la variable 'transform' de este nodo camara
```

```
14
15 func _actualiza_transf_vista(  ) -> void :
16     var ahr  := ((45.0+float(dxy.x))*2.0*PI)/360.0
17     var avr  := ((30.0+float(dxy.y))*2.0*PI)/360.0
18     var tras := Transform3D().translated( Vector3( 0.0,
    0.0, dz))
19     var rotx := Transform3D().rotated( Vector3.RIGHT, -avr
    )
20     var roty := Transform3D().rotated( Vector3.UP, ahr )
21     transform = roty*rotx*tras
22
23 # -------------
24 # NUEVA FUNCION: Ajuste dinamico de la proyeccion (Problema
     6.6)
25 func _actualiza_proyeccion() -> void:
26     # 1. Obtener tamano del viewport
27     var vp_size := get_viewport().size
28
29     # Evitamos division por cero si la ventana se minimiza
    completamente
30     if vp_size.y == 0: return
31
32     # 2 y 3. Calcular relacion de aspecto (ancho / alto)
33     var aspect_ratio := float(vp_size.x) / float(vp_size.y)
34
35     # 4. Ajuste segun la forma de la ventana
36     if aspect_ratio < 1.0:
37         # Si es mas alto que ancho (Portrait), fijamos el
    ancho
38         keep_aspect = Camera3D.KEEP_WIDTH
39     else:
40         # Si es mas ancho que alto (Landscape), fijamos el
    alto (por defecto)
41         keep_aspect = Camera3D.KEEP_HEIGHT
42
43     # Aseguramos que el FOV base sea siempre 75 grados
44     fov = 75.0
45
46 # -------------
47 func _ready() -> void :
48     _actualiza_transf_vista()
49
50     # Conectamos la senal de redimensionado a nuestra nueva
     funcion
51     get_tree().root.size_changed.connect(
    _actualiza_proyeccion)
52
53     # Llamamos a la funcion una vez al inicio para
```

```
    configurar el estado inicial
54      _actualiza_proyeccion()
55
56  # -------------
57  # procesa evento de entrada (sin cambios respecto al
        original)
58
59  func _input( event : InputEvent ):
60      var av : bool = true
61
62      if event is InputEventKey and event.pressed:
63          match event.keycode:
64              KEY_UP:    dxy += Vector2( 0, -at )
65              KEY_DOWN:  dxy += Vector2( 0, +at )
66              KEY_RIGHT: dxy += Vector2( -at, 0 )
67              KEY_LEFT:  dxy += Vector2( at, 0 )
68              KEY_MINUS, KEY_PAGEDOWN, KEY_KP_SUBTRACT: dz *=
        1.05
69              KEY_PLUS, KEY_PAGEUP, KEY_KP_ADD: dz = max( dz
       /1.05, 0.1 )
70              _: av = false
71
72      elif event is InputEventMouseButton:
73          match event.button_index:
74              MOUSE_BUTTON_RIGHT:
75                  bdrp = event.pressed
76                  av = false
77              MOUSE_BUTTON_WHEEL_DOWN: dz *= 1.05
78              MOUSE_BUTTON_WHEEL_UP:   dz = max( dz/1.05, 0.1
        )
79              _: av = false
80
81      elif event is InputEventMouseMotion and bdrp:
82          dxy += ar * Vector2( -event.relative.x, event.
       relative.y )
83
84      else:
85          av = false
86
87      if av:
88          _actualiza_transf_vista( )
```

Para simplificar, lo que se hace es añadir esta función y usarla en el `_ready` y cada vez que se redimensiona la ventana:

```
1   # NUEVA FUNCION: Ajuste dinamico de la proyeccion (Problema
        6.6)
```

```
func _actualiza_proyeccion() -> void:
    # 1. Obtener tamano del viewport
    var vp_size := get_viewport().size

    # Evitamos division por cero si la ventana se minimiza completamente
    if vp_size.y == 0: return

    # 2 y 3. Calcular relacion de aspecto (ancho / alto)
    var aspect_ratio := float(vp_size.x) / float(vp_size.y)

    # 4. Ajuste segun la forma de la ventana
    if aspect_ratio < 1.0:
        # Si es mas alto que ancho (Portrait), fijamos el ancho
        keep_aspect = Camera3D.KEEP_WIDTH
    else:
        # Si es mas ancho que alto (Landscape), fijamos el alto (por defecto)
        keep_aspect = Camera3D.KEEP_HEIGHT

    # Aseguramos que el FOV base sea siempre 75 grados
    fov = 75.0
```

Ejercicio 1.5.7. Se desea calcular los parámetros de la matriz de proyección perspectiva (l, r, b, t, n, f) para visualizar una escena compuesta por un cubo de lado s.

Datos conocidos:

- El cubo tiene lado s.
- El centro del cubo está en coordenadas del mundo $c = (c_x, c_y, c_z)$.
- La cámara (observador) se sitúa en $o_{ec} = (c_x, c_y, c_z + s + 2)$.
- La cámara mira hacia el centro del cubo ($a = c$) y el vector arriba es $(0, 1, 0)$.

Requerimientos:

- Ajustar la vista para que el objeto se vea lo más grande posible sin recortarse (zoom máximo).
- Ajustar los planos de recorte *near* y far lo más ceñidos posible al objeto.
- Mantener la proporción (sin deformación) en un viewport cuadrado.

Solución 1.5.7. Para resolver esto, imaginemos que trasladamos todo el sistema para que la cámara sea el centro del universo $(0, 0, 0)$. Analizaremos distancias relativas desde la cámara hasta el objeto.

1) **Paso 1: Entender la posición relativa (Distancia D).**
   La cámara y el cubo están alineados en los ejes X e Y (tienen las mismas coordenadas c_x, c_y). La única diferencia es la profundidad (eje Z).
   Calculamos la distancia D desde el ojo hasta el **centro** del cubo:

$$D = Z_{ojo} - Z_{cubo} = (c_z + s + 2) - c_z = s + 2$$

La cámara mira hacia el eje $-Z$, por lo que el cubo está flotando delante de nosotros a una distancia de $s + 2$ unidades.

2) **Paso 2: Calcular los planos de profundidad (n y f).**
Los parámetros n (near/cerca) y f (far/lejos) definen qué "rebanada" del mundo ve la cámara. Queremos que esta rebanada empiece justo en la cara frontal del cubo y termine justo en la cara trasera.
Sabemos que el cubo mide s de profundidad. Por tanto, desde su centro, se extiende $s/2$ hacia adelante (hacia la cámara) y $s/2$ hacia atrás.
   - **Plano Near (n):** Es la distancia desde el ojo hasta la cara más cercana del cubo.

$$n = \text{Distancia al centro} - \text{Mitad del cubo}$$

$$n = (s + 2) - \frac{s}{2} = \frac{s}{2} + 2$$

   - **Plano Far (f):** Es la distancia desde el ojo hasta la cara más lejana del cubo.

$$f = \text{Distancia al centro} + \text{Mitad del cubo}$$

$$f = (s + 2) + \frac{s}{2} = \frac{3s}{2} + 2$$

3) **Paso 3: Calcular el marco de la ventana (l, r, b, t).**
Estos parámetros definen el tamaño del "marco de la ventana" a través del cual miramos, situado en la distancia n.
Queremos que el objeto ocupe toda la pantalla. En perspectiva, si la cara delantera del cubo entra justa en la ventana, la cara trasera (que está más lejos) se verá más pequeña y entrará seguro. Por tanto, ajustamos la ventana al tamaño de la cara delantera.
La cara delantera del cubo es un cuadrado de lado s. Como la cámara está centrada:
   - La mitad del cubo va hacia la derecha y la mitad hacia la izquierda.
   - La mitad va hacia arriba y la mitad hacia abajo.

Por tanto, en el plano de proyección (que hemos situado pegado a la cara delantera, en n):

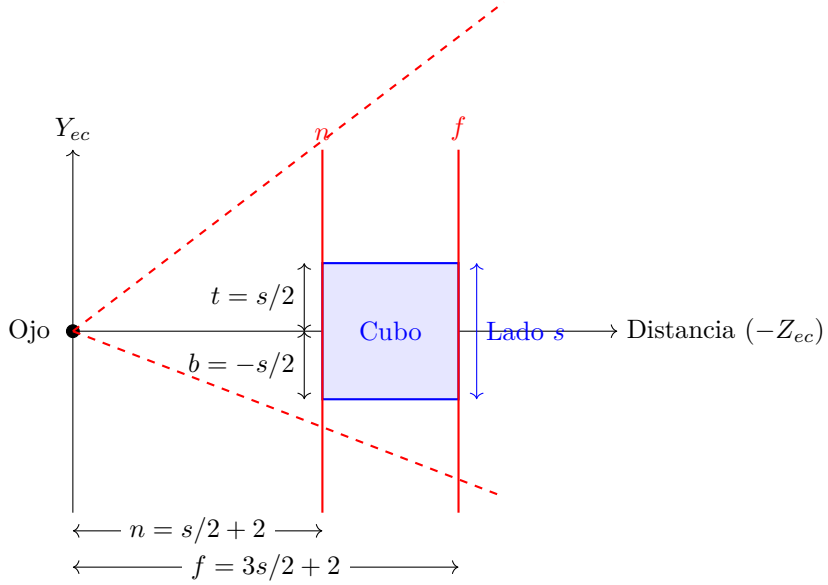
$$r = \text{mitad del ancho} = \frac{s}{2}$$

$$l = -\text{mitad del ancho} = -\frac{s}{2}$$

$$t = \text{mitad de la altura} = \frac{s}{2}$$

$$b = -\text{mitad de la altura} = -\frac{s}{2}$$

4) **Esquema Gráfico de la Solución:**
El siguiente diagrama muestra la vista lateral (perfil). El ojo está en el origen. El cubo (azul) está delimitado por los planos n y f (rojo). Las líneas discontinuas muestran el campo de visión.



5) **Resultado Final:** Los valores calculados únicamente en función de s son:

$$n = \frac{s}{2} + 2, \quad f = \frac{3s}{2} + 2$$

$$r = \frac{s}{2}, \quad l = -\frac{s}{2}, \quad t = \frac{s}{2}, \quad b = -\frac{s}{2}$$

Ejercicio 1.5.8. Repetimos el problema 6.7 con los mismos requerimientos y suposiciones, pero ahora la escena está contenida en una esfera de radio r con centro en $c = (c_x, c_y, c_z)$, en lugar de un cubo.

Datos y Adaptación del Enunciado:

- Objeto: Esfera de radio r.
- Centro: $c = (c_x, c_y, c_z)$.
- Cámara: Para mantener la equivalencia con el ejercicio anterior (donde la distancia dependía del tamaño del objeto s), sustituimos el lado del cubo s por el diámetro de la esfera $2r$.
- Posición de la cámara: $o_{ec} = (c_x, c_y, c_z + 2r + 2)$.
- Orientación: Mira hacia c, vector arriba $(0, 1, 0)$.

Requerimientos:

- n y f ajustados al máximo al objeto.
- Tamaño aparente máximo sin recortar (la esfera debe entrar completa en la imagen).
- Viewport cuadrado (aspect ratio 1).

Solución 1.5.8. Procederemos de forma análoga al caso del cubo, utilizando la **caja englobante** (bounding box) de la esfera para asegurar que esta quede completamente dentro del volumen de vista. Una esfera de radio r cabe perfectamente dentro de un cubo de lado $s = 2r$.

1) **Paso 1: Análisis de Distancias en el Eje Z.**
Transformamos el centro de la esfera a coordenadas de cámara (poniendo la cámara en el origen). La distancia D desde el ojo hasta el centro c es la diferencia en la coordenada Z:

$$D = Z_{ojo} - Z_{centro} = (c_z + 2r + 2) - c_z = 2r + 2$$

La esfera se extiende una distancia r (el radio) hacia adelante y hacia atrás desde su centro.

2) **Paso 2: Cálculo de los planos de recorte (n y f).**
   - **Plano Near (n):** Debe situarse justo delante del punto más cercano de la esfera.

$$n = D - \text{radio} = (2r + 2) - r = r + 2$$

   - **Plano Far (f):** Debe situarse justo detrás del punto más lejano de la esfera.

$$f = D + \text{radio} = (2r + 2) + r = 3r + 2$$

3) **Paso 3: Cálculo de la ventana de proyección (l, r, b, t).**
   Para asegurar que la esfera se vea completa y lo más grande posible, ajustaremos el frustum para que englobe el cuadrado frontal de la "caja imaginaria" que contiene a la esfera.
   Si el plano de proyección está en n, la sección de la caja englobante en ese plano tiene una altura y anchura igual al diámetro de la esfera ($2r$). Sin embargo, debido a la perspectiva, si ajustamos la ventana para cubrir el tamaño del objeto en el plano *near*, garantizamos que cualquier parte del objeto detrás de ese plano también será visible (ya que el frustum se ensancha).
   La "cara delantera" de nuestra caja imaginaria en $z = -n$ tendría un tamaño de $2r \times 2r$.
   Como la cámara apunta al centro:
   - Ancho total $= 2r \implies$ Del centro a la derecha $= r$.
   - Alto total $= 2r \implies$ Del centro hacia arriba $= r$.

   Por tanto:

$$r = r \quad \text{(coincide con el radio)}$$

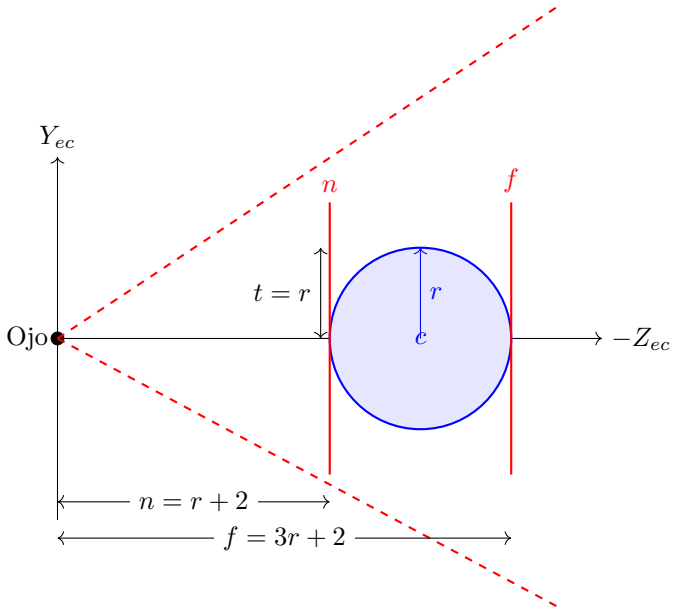
$$t = r$$

$$l = -r$$

$$b = -r$$

   Nota: Al usar $t = r$ en el plano n, estamos definiendo un frustum que pasa exactamente por los bordes de la esfera en su punto más cercano. Como la esfera se curva "hacia adentro", esto garantiza holgura y que la esfera completa sea visible.

4) **Representación Gráfica:**
   El esquema muestra la esfera (azul) y cómo los planos n y f la encierran (rojo).



5) **Resumen de resultados:** Los parámetros en función de r son:

$$n = r + 2, \quad f = 3r + 2$$

$$r_{param} = r, \quad l = -r, \quad t = r, \quad b = -r$$

(Donde r_{param} es el parámetro *right* del frustum y r es el radio de la esfera).

Ejercicio 1.5.9. Repetimos el problema 6.7 (visualización de un cubo de lado s), con los mismos requerimientos de optimización (tamaño máximo, sin recortes, n y f ajustados), pero con una diferencia importante: El viewport (la ventana donde se dibuja la imagen) ya no es necesariamente cuadrado. Tiene dimensiones de w píxeles de ancho y h píxeles de alto.

Datos conocidos:

- Objeto: Cubo de lado s, centrado en c.
- Cámara: Posición $o_{ec} = (c_x, c_y, c_z + s + 2)$, mirando a c.
- Viewport: Resolución $w \times h$. Relación de aspecto $aspect = w/h$.

Objetivo: Calcular n, f, l, r, b, t para que el cubo llene la pantalla lo máximo posible sin perder la proporción (sin deformarse) y sin recortarse.

Solución 1.5.9. Este problema introduce el concepto de **Relación de Aspecto (Aspect Ratio)**. Si la ventana de nuestro programa es rectangular, el volumen de vista (frustum) también debe ser rectangular con la misma proporción, o de lo contrario el cubo se verá estirado o aplastado.

1) **Paso 1: Planos de profundidad (n y f).**
La forma del viewport (rectangular o cuadrada) no afecta a la profundidad. La distancia de la cámara al objeto sigue siendo la misma que en el problema 6.7.
Distancia al centro: $D = s + 2$.
Los planos n y f dependen solo de la coordenada Z del cubo:

$$n = \frac{s}{2} + 2$$

$$f = \frac{3s}{2} + 2$$

(Estos valores son idénticos al problema 6.7).

2) **Paso 2: Relación de Aspecto.**
Definimos la relación de aspecto del viewport como:

$$a = \frac{\text{ancho}}{\text{alto}} = \frac{w}{h}$$

Para evitar deformaciones, las dimensiones físicas de la ventana de proyección ($r - l$ y $t - b$) deben mantener esta misma proporción:

$$\frac{r-l}{t-b} = \frac{2r}{2t} = \frac{r}{t} = a \implies r = t \cdot a$$

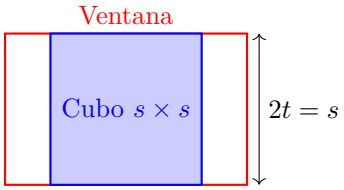
(Asumiendo simetría $r = -l$ y $t = -b$).

3) **Paso 3: Cálculo de la ventana (l, r, b, t).**
La cara del cubo que debemos encuadrar es un **cuadrado de lado** s. Tenemos que meter ese cuadrado de tamaño $s \times s$ dentro de un rectángulo de proporción $w \times h$.
Debemos distinguir dos casos posibles para garantizar que el cubo entre entero ("tamaño aparente mayor posible" significa ajustar a la dimensión más restrictiva).
CASO A: Viewport Apaisado o "Landscape" ($w \geq h$)
   - La ventana es más ancha que alta.
   - Si ajustamos el ancho de la ventana al ancho del cubo ($2r = s$), la altura de la ventana ($2t$) sería proporcionalmente menor a s, y cortaríamos el cubo por arriba y abajo.
   - **Solución:** El factor limitante es la **altura**. Debemos igualar la altura de la ventana a la altura del cubo.
   $$t = \frac{s}{2}, \quad b = -\frac{s}{2}$$
   - El ancho se ajusta automáticamente para mantener la proporción (será mayor que s, dejando espacio libre a los lados):
   $$r = t \cdot \frac{w}{h} = \frac{s}{2} \cdot \frac{w}{h}$$
   $$l = -r = -\frac{s}{2} \cdot \frac{w}{h}$$

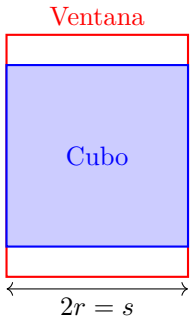
CASO B: Viewport Vertical o "Portrait" ($w < h$)
   - La ventana es más alta que ancha.
   - Si ajustamos la altura de la ventana a la altura del cubo ($2t = s$), el ancho ($2r$) sería menor que s, y cortaríamos el cubo por los lados.
   - **Solución:** El factor limitante es el **ancho**. Debemos igualar el ancho de la ventana al ancho del cubo.
   $$r = \frac{s}{2}, \quad l = -\frac{s}{2}$$
   - La altura se ajusta automáticamente (será mayor que s, dejando espacio libre arriba y abajo):
   $$t = \frac{r}{a} = r \cdot \frac{h}{w} = \frac{s}{2} \cdot \frac{h}{w}$$
   $$b = -t = -\frac{s}{2} \cdot \frac{h}{w}$$

4) **Resumen Gráfico de los Casos:**

Caso A: $w > h$ (Apaisado)



Caso B: $w < h$ (Vertical)



5) **Resultado General Unificado:** Podemos expresar la solución usando la función máximo para cubrir ambos casos:

$$n = \frac{s}{2} + 2, \quad f = \frac{3s}{2} + 2$$

$$r = \frac{s}{2} \cdot \text{máx}\left(1, \frac{w}{h}\right), \quad t = \frac{s}{2} \cdot \text{máx}\left(1, \frac{h}{w}\right)$$

$$l = -r, \quad b = -t$$

Ejercicio 1.5.10. Alta complejidad. Posicionamiento de cámara dado un FOV (β).

Repetimos el problema 6.7 (cubo de lado s centrado en c), manteniendo los requerimientos de optimización (viewport cuadrado, sin recortes, n máximo, f mínimo).

Nueva condición: En lugar de darnos la posición de la cámara, se nos da el ángulo de apertura vertical (Field of View) β. Debemos calcular:

1) La coordenada Z de la posición del observador (o_{ec}), sabiendo que $o_x = c_x$ y $o_y = c_y$.
2) Los parámetros de la proyección l, r, t, b, n, f en función de β, s y c.

Solución 1.5.10. Este problema es "inverso" al anterior en cierto sentido. Antes fijábamos la distancia y calculábamos qué apertura necesitábamos (implícitamente). Ahora, fijamos la apertura (el ángulo de la lente) y tenemos que calcular a qué distancia ponernos para que el cubo llene la pantalla perfectamente.

1) **Paso 1: Entender la geometría del FOV (β).**
El ángulo β es la apertura total vertical. La mitad de ese ángulo es $\beta/2$. En un triángulo rectángulo formado por la línea de visión, el plano de proyección y el borde superior del frustum:
$$\tan(\beta/2) = \frac{\text{altura del marco}}{\text{distancia al marco}} = \frac{t}{n}$$
Queremos que el cubo llene la pantalla. Esto ocurre cuando el "marco" de visión en el plano más cercano (n) coincide exactamente con la cara delantera del cubo.
La cara delantera del cubo tiene altura s. Por tanto, desde el centro hacia arriba mide $s/2$. Esto fija nuestro valor de t:
$$t = \frac{s}{2}$$

2) **Paso 2: Calcular la distancia al plano Near (n).**
Sustituimos t en la ecuación del FOV y despejamos n:
$$\tan(\beta/2) = \frac{s/2}{n}$$

$$n = \frac{s/2}{\tan(\beta/2)} = \frac{s}{2} \cdot \cot(\beta/2)$$

Ahora ya sabemos cuánto espacio debe haber entre el ojo y la cara delantera del cubo (n).

3) **Paso 3: Calcular la posición de la cámara (o_z).**
Sabemos dónde está el cubo en el mundo (en c_z).
   - El centro del cubo está en c_z.
   - La cara delantera está en $c_z + s/2$ (hacia nosotros).
   - El ojo está una distancia n más allá de la cara delantera.

$$o_z = \text{Posición cara delantera} + n$$

$$o_z = (c_z + \frac{s}{2}) + n$$

Sustituyendo el valor de n calculado antes:

$$o_z = c_z + \frac{s}{2} + \frac{s}{2}\cot(\beta/2) = c_z + \frac{s}{2}\left(1 + \cot\left(\frac{\beta}{2}\right)\right)$$

Por tanto, la posición del observador es:

$$o_{ec} = \left(c_x, c_y, c_z + \frac{s}{2}\left(1 + \cot\left(\frac{\beta}{2}\right)\right)\right)$$

4) **Paso 4: Calcular el resto de parámetros (f, l, r, b).**
   - f **(Far):** Es la distancia desde el ojo hasta la cara trasera. La cara trasera está a una distancia s (la profundidad del cubo) más lejos que la cara delantera (n).

$$f = n + s$$

$$f = \frac{s}{2}\cot(\beta/2) + s$$

   - t, b, l, r**:** Como el viewport es cuadrado (según enunciado 6.7) y queremos ajustar a la cara delantera ($s \times s$):

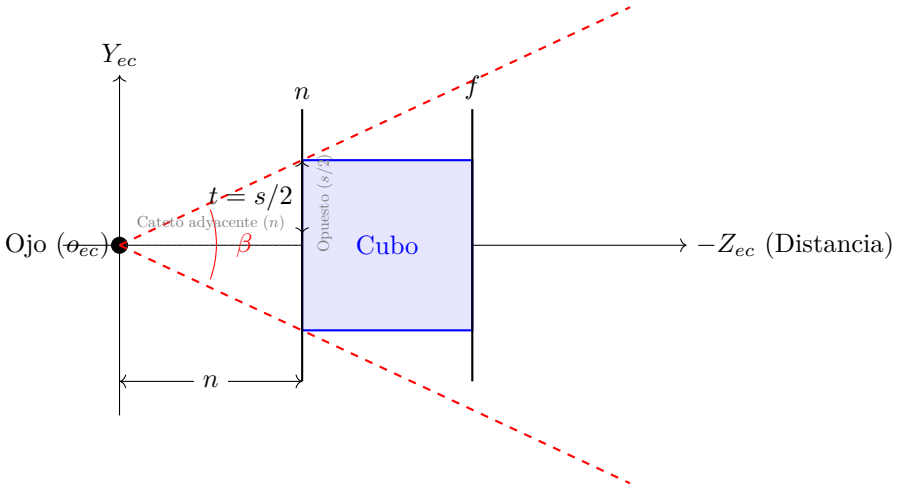
$$t = \frac{s}{2}$$

$$b = -\frac{s}{2}$$

$$r = \frac{s}{2}$$

$$l = -\frac{s}{2}$$

5) **Esquema Gráfico:**
El diagrama muestra cómo el ángulo β determina la distancia n para que el frustum coincida con la altura $s/2$.



6) **Resumen de Fórmulas:**

$$o_z = c_z + \frac{s}{2}\left(1 + \cot\frac{\beta}{2}\right)$$

$$n = \frac{s}{2}\cot\frac{\beta}{2}$$

$$f = n + s$$

$$r = t = \frac{s}{2}, \quad l = b = -\frac{s}{2}$$

1.6 Sesión 7

Ejercicio 1.6.1. Implementación de Componentes Especulares (Phong y Blinn-Phong).

Escribe el código en GDScript para dos funciones que calculen la reflectividad debida a la componente pseudo-especular de los modelos de iluminación local:

1) **Modelo de Phong:** Evaluar la expresión f_{ph} (Ecuación 6).
2) **Modelo de Blinn-Phong:** Evaluar la expresión f_{bp} (Ecuación 7).

Ambas funciones recibirán como parámetros:

- Los vectores unitarios: Normal en el punto ($\mathbf{n}_p$), vector hacia el observador ($\mathbf{v}$) y vector hacia la fuente de luz ($\mathbf{l}_i$).
- El exponente de brillo e (shininess).
- El coeficiente especular k_s (o k_{ph}/k_{bp}).

La función debe devolver un valor de tipo `float` que represente la intensidad de la luz reflejada especularmente.

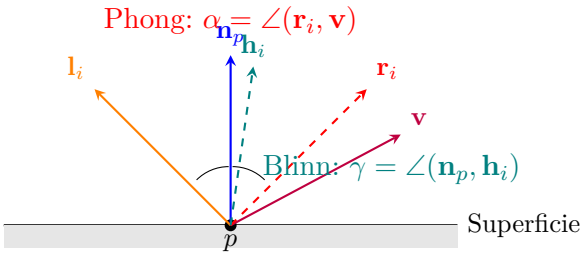


Figura 1.1: *Esquema de vectores para Phong ($\mathbf{r}_i$) y Blinn-Phong ($\mathbf{h}_i$).*

Solución 1.6.1. A continuación se detalla el procedimiento geométrico y la implementación en código GDScript para ambos modelos.

1) **Modelo de Sombreado de Phong (f_{ph})**
El modelo de Phong calcula el brillo especular basándose en el ángulo entre el vector de visión $\mathbf{v}$ y el vector de reflexión perfecta de la luz $\mathbf{r}_i$.
Fórmulas requeridas:
   - Vector de reflexión: $\mathbf{r}_i = 2(\mathbf{n}_p \cdot \mathbf{l}_i)\mathbf{n}_p - \mathbf{l}_i$.
   - Condición de luz incidente: $d_i = 1$ si $\mathbf{n}_p \cdot \mathbf{l}_i > 0$, de lo contrario 0.
   - Intensidad: $I = k_{ph} \cdot (\text{máx}(0, \mathbf{r}_i \cdot \mathbf{v}))^e$.

   Código GDScript:

```
func calcular_phong_especular(n: Vector3, v: Vector3, l:
    Vector3, e: float, k_ph: float) -> float:
    # 1. Calcular el producto punto entre la normal y la
    luz (Lambert)
    var n_dot_l : float = n.dot(l)

    # 2. Si la luz está detrás de la superficie, no hay
    especularidad
    if n_dot_l <= 0.0:
        return 0.0

    # 3. Calcular el vector reflejado r
    # Fórmula: r = 2 * (n . l) * n - l
    # En GDScript se puede usar reflect(), pero ojo:
    reflect devuelve
    # el vector reflejado dada la dirección incidente y la
    normal.
    # La fórmula manual es más explícita para teoría.
    var r : Vector3 = (2.0 * n_dot_l * n - l).normalized()

    # 4. Calcular el factor especular (r . v)^e
    var r_dot_v : float = max(0.0, r.dot(v))
    var specular : float = pow(r_dot_v, e)

    # 5. Devolver intensidad final ponderada por k_ph
    return k_ph * specular
```

2) **Modelo de Blinn-Phong (f_{bp})**
El modelo de Blinn-Phong optimiza el cálculo y suaviza el resultado utilizando el vector intermedio o *halfway vector* $\mathbf{h}_i$, que es la bisectriz entre la luz $\mathbf{l}_i$ y la visión $\mathbf{v}$.
Fórmulas requeridas:
   - Vector Halfway: $\mathbf{h}_i = \frac{\mathbf{l}_i + \mathbf{v}}{||\mathbf{l}_i + \mathbf{v}||}$.
   - Intensidad: $I = k_{bp} \cdot (\mathbf{n}_p \cdot \mathbf{h}_i)^e$.

   Código GDScript:

```gdscript
func calcular_blinn_phong_especular(n: Vector3, v: Vector3, l: Vector3, e: float, k_bp: float) -> float:
    # 1. Calcular el producto punto N.L para descartar luz trasera
    var n_dot_l : float = n.dot(l)

    if n_dot_l <= 0.0:
        return 0.0

    # 2. Calcular el vector halfway (bisectriz) h
    # Es la suma de L y V, normalizada
    var h : Vector3 = (l + v).normalized()

    # 3. Calcular el producto punto entre la normal y el halfway vector
    var n_dot_h : float = max(0.0, n.dot(h))

    # 4. Elevar a la potencia (exponente de brillo)
    var specular : float = pow(n_dot_h, e)

    # 5. Devolver resultado ponderado
    return k_bp * specular
```

Nota técnica: En GDScript, la clase `Vector3` asume que los vectores ya están normalizados si el enunciado dice "vectores unitarios". Si no se garantiza, se debería llamar a `.normalized()` sobre los parámetros de entrada antes de operar.

Ejercicio 1.6.2. Cálculo de máximos de intensidad y visibilidad en una esfera.

Supongamos una esfera de radio unidad centrada en el origen.

- Se ilumina con una fuente de luz puntual en $\mathbf{p} = (0, 2, 0)$.
- El observador está situado en $\mathbf{o} = (2, 0, 0)$.

Determinar razonadamente el punto de la superficie donde el brillo será máximo y si dicho punto es visible para el observador para los siguientes casos:

1) Componente difusa (Lambertiana).
2) Componente pseudo-especular de Phong.
3) Componente pseudo-especular de Blinn-Phong.

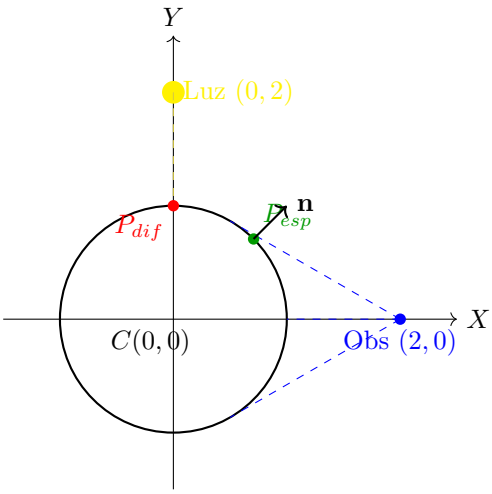


Figura 1.2: *Diagrama de la escena en el plano XY* ($z = 0$).

Solución 1.6.2. Analizaremos cada caso paso a paso. Dado que tanto la luz como el observador están en el plano XY ($z = 0$) y la esfera está centrada en el origen, los puntos de máximo brillo estarán necesariamente en el círculo máximo del plano XY.

Datos geométricos generales para un punto $P(x, y, z)$ en la superficie de la esfera unitaria:

- Radio $R = 1$, Centro $C = (0, 0, 0)$.
- La normal en la superficie es $\mathbf{n}_p = P - C = (x, y, z)$.
- Vector hacia la luz: $\mathbf{l} = \text{normalizar}(\mathbf{p} - P)$.
- Vector hacia el observador: $\mathbf{v} = \text{normalizar}(\mathbf{o} - P)$.

Condición de Visibilidad: Un punto P es visible si el ángulo entre la normal $\mathbf{n}_p$ y el vector de visión $\mathbf{v}$ es menor de 90 grados, es decir, $\mathbf{n}_p \cdot \mathbf{v} > 0$.

Analicemos el horizonte de visibilidad para el observador en $(2, 0, 0)$:

$$\mathbf{v}_{aprox} \approx (2, 0, 0) - (x, y, z) = (2 - x, -y, -z)$$

$$\mathbf{n}_p \cdot \mathbf{v}_{aprox} \propto (x, y, z) \cdot (2 - x, -y, -z) = 2x - (x^2 + y^2 + z^2) = 2x - 1$$

La condición $\mathbf{n}_p \cdot \mathbf{v} > 0 \implies 2x - 1 > 0 \implies x > 0{,}5$. *Cualquier punto con coordenada* $x \leq 0{,}5$ *está oculto por el horizonte de la esfera.*

1) **Componente Difusa (Lambertiana)**
   La intensidad difusa es proporcional a $\mathbf{n}_p \cdot \mathbf{l}$. El brillo es máximo cuando la normal apunta directamente a la luz ($\mathbf{n}_p \parallel \mathbf{l}$).
   - Dirección desde el centro a la luz: $(0, 2, 0) - (0, 0, 0) = (0, 2, 0)$.
   - El punto de la superficie en esa dirección es $P_{dif} = (0, 1, 0)$.
   - **Visibilidad:** La coordenada x de P_{dif} es 0.
   - Como $0 \leq 0{,}5$, el punto **NO es visible**. Está en la parte superior de la esfera, pero el observador, situado a la derecha, solo ve hasta $x > 0{,}5$.
2) **Componente Pseudo-especular (Phong)**
   La intensidad es proporcional a $(\mathbf{r} \cdot \mathbf{v})^e$, donde $\mathbf{r}$ es el reflejo de la luz sobre la normal. El máximo ocurre cuando $\mathbf{r} = \mathbf{v}$ (reflexión perfecta). Esto implica que la normal $\mathbf{n}_p$ debe ser la bisectriz del ángulo formado por el vector luz $\mathbf{l}$ y el vector visión $\mathbf{v}$.
   Debido a la simetría del problema (Luz en eje Y, Observador en eje X, distancias iguales al origen), el punto debe estar en la bisectriz del primer cuadrante ($x = y$).

- Punto en la esfera a 45 grados: $P_{esp} = (\cos(45°), \sin(45°), 0) = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right) \approx (0{,}707, 0{,}707, 0)$.
- Comprobación geométrica: La normal en este punto apunta a $(1, 1)$. La luz está en $(0, 2)$ y el ojo en $(2, 0)$. El vector normal divide simétricamente el ángulo entre la luz y el ojo.
- **Visibilidad:** La coordenada x de P_{esp} es 0,707.
- Como $0{,}707 > 0{,}5$, el punto **SÍ es visible**. El brillo especular aparecerá en el "hombro" de la esfera mirando hacia el observador.

3) **Modelo de Blinn-Phong**
   La intensidad es proporcional a $(\mathbf{n}_p \cdot \mathbf{h})^e$, donde $\mathbf{h}$ (halfway vector) es la bisectriz entre $\mathbf{l}$ y $\mathbf{v}$. El máximo ocurre cuando la normal $\mathbf{n}_p$ coincide con $\mathbf{h}$.
   - Geométricamente, la condición "la normal coincide con la bisectriz de L y V" es idéntica a la condición de reflexión perfecta del modelo de Phong descrita arriba.
   - Por tanto, el punto de máximo brillo es el mismo: $P_{bp} = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right)$.
   - **Visibilidad:** Al ser el mismo punto, **SÍ es visible**.

Ejercicio 1.6.3. Evaluación de la BRDF de Microfacetas (GGX).

Escribe el código en GDScript de una función para calcular la reflectividad debida a la BRDF de microfacetas GGX, evaluando la expresión de f_{ggx} (Ecuación 10).

La función recibirá los siguientes parámetros:

- Vectores unitarios: Dirección de iluminación ($\mathbf{w}_i$), dirección de visión ($\mathbf{w}_o$), tangente X ($\mathbf{t}_x$), tangente Y ($\mathbf{t}_y$) y normal de la macrosuperficie ($\mathbf{n}_x$).
- Valores de rugosidad: α_x y α_y (tipo float).

La función debe devolver un valor de tipo `float`.

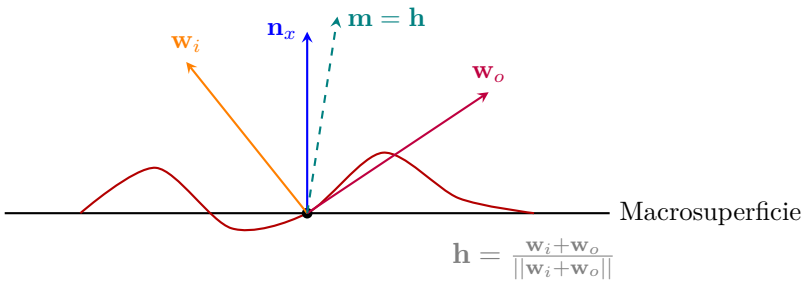


Figura 1.3: *Geometría de microfacetas: El vector* $\mathbf{h}$ *actúa como la normal de la microfaceta* ($\mathbf{m}$) *que refleja* $\mathbf{w}_i$ *hacia* $\mathbf{w}_o$.

Solución 1.6.3. Para implementar la BRDF GGX completa, debemos desglosar la Ecuación 10 en sus tres componentes principales: la Distribución de Normales (D), el Enmascaramiento-Sombreado (G) y el término de Fresnel (F).

1) **Cálculo del Vector Halfway (h):** Es la bisectriz entre el vector de luz y el de visión. Representa la orientación que debe tener una microfaceta para reflejar la luz perfectamente hacia el observador.
$$\mathbf{h} = \frac{\mathbf{w}_i + \mathbf{w}_o}{||\mathbf{w}_i + \mathbf{w}_o||}$$

2) **Distribución de Normales Anisotrópica (D):** Evaluamos la probabilidad de que una microfaceta esté alineada con $\mathbf{h}$. Usamos la fórmula GGX anisotrópica (Ecuación de transpa-

rencia 75):

$$D(\mathbf{h}) = \frac{1}{\pi\alpha_x\alpha_y\left(\left(\frac{\mathbf{h}\cdot\mathbf{t}_x}{\alpha_x}\right)^2 + \left(\frac{\mathbf{h}\cdot\mathbf{t}_y}{\alpha_y}\right)^2 + (\mathbf{h}\cdot\mathbf{n}_x)^2\right)^2}$$

3) **Enmascaramiento y Sombreado (G_2):** Usamos la aproximación *Height Correlated Masking and Shadowing* (Ecuación de transparencia 77). Se define mediante una función auxiliar $\Lambda(w)$:

$$\Lambda(\mathbf{w}) = \frac{1}{2}\left(-1 + \sqrt{1 + \frac{\alpha_x^2 x^2 + \alpha_y^2 y^2}{z^2}}\right)$$

Donde x, y, z son las proyecciones del vector $\mathbf{w}$ sobre $\mathbf{t}_x, \mathbf{t}_y, \mathbf{n}_x$.

$$G_2 = \frac{1}{1 + \Lambda(\mathbf{w}_i) + \Lambda(\mathbf{w}_o)}$$

4) **Término de Fresnel (F):** Usamos la aproximación de Schlick (Ecuación de transparencia 78). Aunque el enunciado no proporciona el índice de refracción (f_0), es necesario para la ecuación. Asumiremos un valor estándar de 0,04 (dieléctrico común) para completar el cálculo.

$$F \approx f_0 + (1 - f_0)(1 - (\mathbf{w}_i \cdot \mathbf{h}))^5$$

5) **Combinación Final (f_{ggx}):**

$$f_{ggx} = \frac{F \cdot D \cdot G_2}{4(\mathbf{w}_i \cdot \mathbf{n}_x)(\mathbf{w}_o \cdot \mathbf{n}_x)}$$

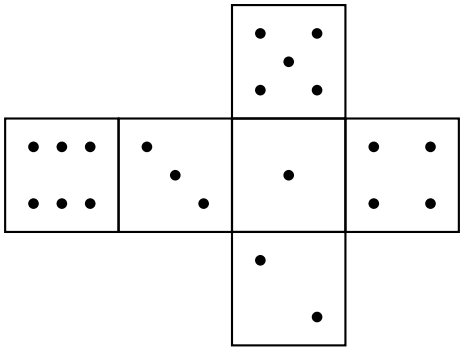
Implementación en GDScript:

```
func calcular_brdf_ggx(wi: Vector3, wo: Vector3, tx: Vector3, ty
    : Vector3, nx: Vector3, ax: float, ay: float) -> float:
    # 1. Calcular el vector Halfway (h)
    var h: Vector3 = (wi + wo).normalized()

    # Pre-cálculo de productos punto necesarios
    var n_dot_wi = max(0.0001, nx.dot(wi)) # Evitar división por
     cero
    var n_dot_wo = max(0.0001, nx.dot(wo))
    var n_dot_h  = max(0.0, nx.dot(h))
    var h_dot_wi = max(0.0, h.dot(wi))

    # Proyecciones para anisotropía
    var h_dot_tx = h.dot(tx)
    var h_dot_ty = h.dot(ty)

    # 2. Calcular Distribución D (GGX Anisotrópica)
    var term_x = pow(h_dot_tx / ax, 2)
    var term_y = pow(h_dot_ty / ay, 2)
    var term_z = pow(n_dot_h, 2)

    var denom_d = PI * ax * ay * pow(term_x + term_y + term_z,
```

```
    2)
21  var D = 1.0 / max(0.0001, denom_d)
22
23  # 3. Calcular Geometría G2 (Height Correlated)
24  # Función Lambda auxiliar inline para wi
25  var wi_x = wi.dot(tx) * ax
26  var wi_y = wi.dot(ty) * ay
27  var wi_z = n_dot_wi
28  var lambda_wi = 0.5 * (-1.0 + sqrt(1.0 + (pow(wi_x, 2) + pow
    (wi_y, 2)) / pow(wi_z, 2)))
29
30  # Función Lambda auxiliar inline para wo
31  var wo_x = wo.dot(tx) * ax
32  var wo_y = wo.dot(ty) * ay
33  var wo_z = n_dot_wo
34  var lambda_wo = 0.5 * (-1.0 + sqrt(1.0 + (pow(wo_x, 2) + pow
    (wo_y, 2)) / pow(wo_z, 2)))
35
36  var G2 = 1.0 / (1.0 + lambda_wi + lambda_wo)
37
38  # 4. Calcular Fresnel F (Aproximación de Schlick)
39  var f0 = 0.04 # Valor asumido para dieléctricos si no se
    provee
40  var F = f0 + (1.0 - f0) * pow(1.0 - h_dot_wi, 5)
41
42  # 5. Resultado final combinado
43  var numerador = F * D * G2
44  var denominador = 4.0 * n_dot_wi * n_dot_wo
45
46  return numerador / max(0.0001, denominador)
```

1.7 Sesión 8

Ejercicio 1.7.1. Supongamos que se desea crear una malla indexada para un cubo, de forma que deseamos aplicar una textura que incluya las caras de un dado. Para ello disponemos de una imagen de textura que tiene una relación de aspecto 4:3.



1) Describe razonadamente cuántos vértices (como mínimo) tendrá el modelo.
2) Escribe la tabla de coordenadas de vértices, la tabla de coordenadas de textura y la tabla de triángulos. Ten en cuenta que el cubo tiene lado unidad y su centro está en $(0{,}5, 0{,}5, 0{,}5)$.
3) Dibuja un esquema de la textura en la cual cada vértice del modelo aparezca etiquetado con su número de vértice más sus coordenadas de textura.

Solución 1.7.1. La resolución del ejercicio es la siguiente:

1) Número de Vértices del Modelo
Aunque un cubo geométrico estándar tiene 8 vértices espaciales (esquinas), en informática gráfica, un vértice en una malla indexada se define como una tupla única de atributos: $(x, y, z, u, v, \dots)$. Si un mismo punto geométrico (esquina del cubo) necesita tener dos coordenadas de textura distintas (por ejemplo, en una costura donde la textura se corta), el vértice debe duplicarse.
Observando la distribución de la textura en cruz proporcionada en las diapositivas, la imagen tiene un aspect ratio 4:3, lo que implica una rejilla de 4×3 caras. La disposición es:
   - Fila superior: Cara 5.
   - Fila media: Caras 6, 3, 1, 4.
   - Fila inferior: Cara 2.

   Para calcular el número mínimo de vértices, analizamos la conectividad en el espacio UV:
   - Si tratamos cada cara como un cuadrado independiente, tendríamos $6 \times 4 = 24$ vértices.
   - Restamos los vértices que se comparten en las aristas continuas en la textura (donde no hay corte UV). Las conexiones visuales son: 6-3, 3-1, 1-4, 5-1 y 1-2.
   - Hay 5 aristas compartidas. Cada arista fusiona 2 pares de vértices.
   - Total vértices $= 24 - (5 \text{ aristas} \times 2 \text{ vértices}) = 14$.

   Por tanto, el modelo necesita **14 vértices** únicos.

2) Tablas de Definición del Modelo
Asumimos el sistema de referencia donde la cara 1 es el Frontal ($z = 1$), la cara 5 es Arriba ($y = 1$), la cara 2 es Abajo ($y = 0$), la cara 3 es Izquierda ($x = 0$), la cara 4 es Derecha ($x = 1$) y la cara 6 es Atrás ($z = 0$). El cubo va de $(0, 0, 0)$ a $(1, 1, 1)$.
Dividimos el dominio de textura $u \in [0, 1], v \in [0, 1]$ según la rejilla 4x3[2]:
   - Paso en u: $1/4 = 0{,}25$. Columnas: $0, 0{,}25, 0{,}5, 0{,}75, 1{,}0$.
   - Paso en v: $1/3 \approx 0{,}333$. Filas: $0, 0{,}33, 0{,}66, 1{,}0$.

   Nota: Las divisiones que se hacen de u y v corresponden a cada vértice, de manera que tan solo tenemos que imaginar que la textura es como una tabla, si vemos en la cara 5 (arriba) esta entre u=0.5 a u=0.75 y v=0.66 a v=1.0. En la tabla se hace referencia a top-esquina, lo que se conoce como top-left en inglés, por ende, debemos debemos de tener en cuenta que la coordenada v=1.0 es la parte superior de la textura y v=0.0 es la parte inferior. Se le debe de atribuir u=0.5 y v=1.0 a la esquina superior izquierda de la cara 5 (arriba).

   Tabla de Vértices (Geometría + Textura)

   Ordenamos los vértices recorriendo la textura de arriba a abajo y de izquierda a derecha.

[2]Es en base al enunciado.

Índice (i)	**Posición** (x, y, z)	**Coord. Textura** (u, v)	**Descripción (UV)**
0	$(0, 1, 0)$	(0,50, 1,00)	Top-Esq Cara 5
1	$(1, 1, 0)$	(0,75, 1,00)	Top-Der Cara 5
2	$(1, 1, 0)$	(0,00, 0,66)	Top-Esq Cara 6
3	$(0, 1, 0)$	(0,25, 0,66)	Top-Der 6 / Top-Esq 3
4	$(0, 1, 1)$	(0,50, 0,66)	Top-Der 3 / Top-Esq 1 / Bot-Esq 5
5	$(1, 1, 1)$	(0,75, 0,66)	Top-Der 1 / Top-Esq 4 / Bot-Der 5
6	$(1, 1, 0)$	(1,00, 0,66)	Top-Der 4
7	$(1, 0, 0)$	(0,00, 0,33)	Bot-Esq Cara 6
8	$(0, 0, 0)$	(0,25, 0,33)	Bot-Der 6 / Bot-Esq 3
9	$(0, 0, 1)$	(0,50, 0,33)	Bot-Der 3 / Bot-Esq 1 / Top-Esq 2
10	$(1, 0, 1)$	(0,75, 0,33)	Bot-Der 1 / Bot-Esq 4 / Top-Der 2
11	$(1, 0, 0)$	(1,00, 0,33)	Bot-Der 4
12	$(0, 0, 0)$	(0,50, 0,00)	Bot-Esq Cara 2
13	$(1, 0, 0)$	(0,75, 0,00)	Bot-Der Cara 2

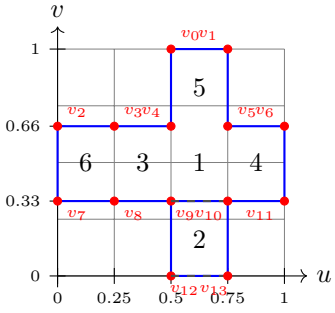
Tabla de Triángulos

Definimos dos triángulos por cara (sentido antihorario visto desde fuera).

Cara (Dado)	**Triángulo 1** (v_a, v_b, v_c)	**Triángulo 2** (v_a, v_c, v_d)
5 (Arriba)	$(0, 1, 4)$	$(1, 5, 4)$
6 (Atrás)	$(2, 3, 7)$	$(3, 8, 7)$
3 (Izq)	$(3, 4, 8)$	$(4, 9, 8)$
1 (Frente)	$(4, 5, 9)$	$(5, 10, 9)$
4 (Der)	$(5, 6, 10)$	$(6, 11, 10)$
2 (Abajo)	$(9, 10, 12)$	$(10, 13, 12)$

Nota: Usamos orden horario.

3) Esquema de la Textura
A continuación se muestra el espacio de coordenadas de textura (u, v) con los vértices etiquetados según la tabla anterior.



El código GDScript para definir las tablas de vértices, coordenadas de textura y triángulos es el siguiente:

```
# Definición de los vértices: posición y coordenadas de textura
var vertices = [
    Vector3(0, 1, 0),   # v0
    Vector3(1, 1, 0),   # v1
    Vector3(1, 1, 0),   # v2
    Vector3(0, 1, 0),   # v3
    Vector3(0, 1, 1),   # v4
    Vector3(1, 1, 1),   # v5
    Vector3(1, 1, 0),   # v6
    Vector3(1, 0, 0),   # v7
    Vector3(0, 0, 0),   # v8
    Vector3(0, 0, 1),   # v9
    Vector3(1, 0, 1),   # v10
    Vector3(1, 0, 0),   # v11
    Vector3(0, 0, 0),   # v12
    Vector3(1, 0, 0),   # v13
]

var uvs = [
    Vector2(0.50, 1.00),   # v0
    Vector2(0.75, 1.00),   # v1
    Vector2(0.00, 0.66),   # v2
    Vector2(0.25, 0.66),   # v3
    Vector2(0.50, 0.66),   # v4
    Vector2(0.75, 0.66),   # v5
    Vector2(1.00, 0.66),   # v6
    Vector2(0.00, 0.33),   # v7
    Vector2(0.25, 0.33),   # v8
    Vector2(0.50, 0.33),   # v9
    Vector2(0.75, 0.33),   # v10
    Vector2(1.00, 0.33),   # v11
    Vector2(0.50, 0.00),   # v12
    Vector2(0.75, 0.00),   # v13
]

# Definición de los triángulos (índices de vértices) en orden
    horario (sentido antihorario visto desde fuera)
var triangles = [
    # Cara 5 (Arriba)
    0, 1, 4,
    1, 5, 4,
    # Cara 6 (Atrás)
    2, 3, 7,
    3, 8, 7,
    # Cara 3 (Izquierda)
    3, 4, 8,
    4, 9, 8,
    # Cara 1 (Frente)
```

```
    4, 5, 9,
    5, 10, 9,
    # Cara 4 (Derecha)
    5, 6, 10,
    6, 11, 10,
    # Cara 2 (Abajo)
    9, 10, 12,
    10, 13, 12,
]
```

Ejercicio 1.7.2. Considera de nuevo el cubo y la textura del problema anterior (un cubo de lado unidad con centro en (0,5, 0,5, 0,5) y una textura de imagen con relación de aspecto 4:3 que despliega las caras de un dado). Supón que ahora queremos visualizar el cubo iluminado, para lo cual debemos asignar normales a los vértices.

Responde a estas cuestiones:

1) Describe razonadamente si sería posible usar la misma tabla de vértices y la misma tabla de coordenadas de textura que en el problema anterior (donde se buscaba el número mínimo de vértices), o si es necesario usar tablas distintas.
2) Si has respondido que no es posible usar las mismas tablas, escribe la nueva tabla de vértices, la nueva tabla de coordenadas de textura y la tabla de normales.

Solución 1.7.2. La resolución del ejercicio es la siguiente:

1. Análisis de la reutilización de la tabla de vértices

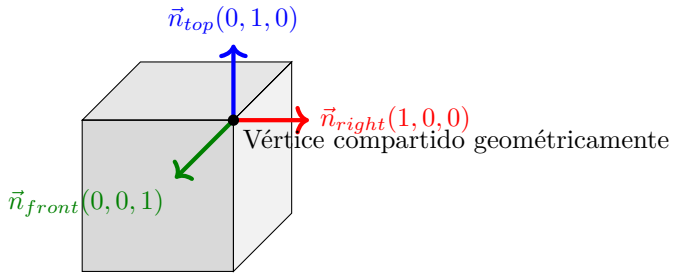
Para responder a esta cuestión, debemos entender qué define un *vértice* en el contexto del cauce gráfico (pipeline) cuando aplicamos iluminación.

En el problema anterior (8.1), buscábamos minimizar el espacio geométrico. Un cubo tiene geométricamente 8 esquinas. Si solo nos importara la posición (x, y, z), podríamos definir solo 8 vértices y reutilizarlos mediante índices.

Sin embargo, para la iluminación (sombreado), necesitamos asociar un **vector normal** ($\vec{n}$) a cada vértice. El vector normal indica hacia dónde "mira" la superficie en ese punto para calcular cómo rebota la luz.

- **El problema de la continuidad:** En una esfera suave, la normal en un vértice es el promedio de las caras adyacentes, permitiendo un sombreado suave (Gouraud).
- **El caso del cubo (aristas vivas):** Un cubo tiene aristas afiladas (no suaves). Consideremos una esquina del cubo, por ejemplo, la superior-derecha-frontal $(1, 1, 1)$.
  - Para la cara **Frontal**, la normal debe apuntar hacia adelante: $\vec{n} = (0, 0, 1)$.
  - Para la cara **Superior**, la normal debe apuntar hacia arriba: $\vec{n} = (0, 1, 0)$.
  - Para la cara **Derecha**, la normal debe apuntar a la derecha: $\vec{n} = (1, 0, 0)$.

Como un vértice en la memoria de la GPU es una estructura de datos única que contiene $\{Posición, Normal, UV\}$, no podemos tener un solo vértice con tres normales distintas simultáneamente.



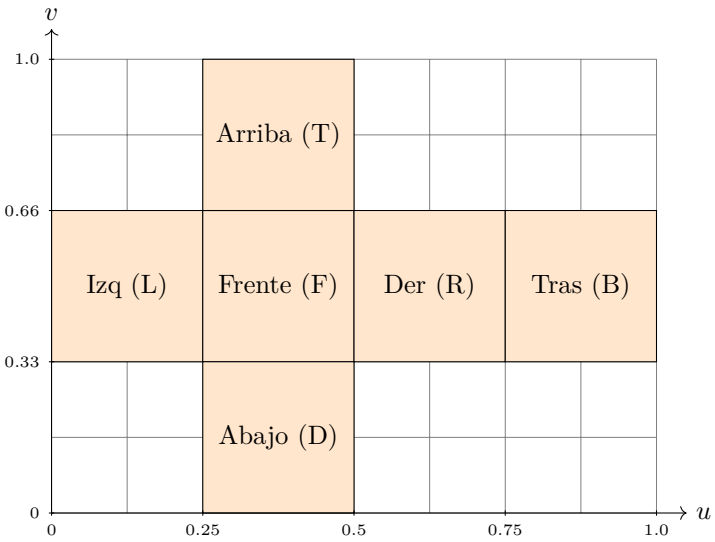
Conclusión: No es posible usar la misma tabla reducida de 8 vértices. Es necesario duplicar los vértices en las costuras de las aristas. Necesitaremos vértices independientes para cada cara del cubo.
Total de vértices necesarios: 6 caras × 4 vértices/cara = **24 vértices**.

2. Definición de las nuevas tablas

Para construir las tablas, asumiremos la disposición de textura "en cruz" típica para una relación de aspecto 4:3, tal como sugiere el enunciado del Problema 8.1.

Esquema de la Textura (Relación 4:3): Dividimos la textura en una cuadrícula de 4×3.

- Ancho de celda (u): $1/4 = 0{,}25$
- Alto de celda (v): $1/3 \approx 0{,}333$



A continuación, definimos las tablas. Dado que el cubo tiene lado 1 y centro en $(0{,}5, 0{,}5, 0{,}5)$, las coordenadas van de 0,0 a 1,0 en los ejes X, Y, Z.

Nota de notación:

- **Posición:** (x, y, z)
- **Normal:** (nx, ny, nz)
- **Textura:** (u, v)

Desglosaremos la tabla cara por cara (cada cara genera 4 vértices únicos).

Tabla Completa de Vértices (Datos combinados)

1) **Cara Frontal (Z = 1)**: Corresponde a la celda ($u \in [0{,}25, 0{,}5], v \in [0{,}33, 0{,}66]$). Normal $\vec{n} = (0, 0, 1)$.

Índice	**Posición** (x, y, z)	**Normal** (nx, ny, nz)	**Textura** (u, v)
0	$(0, 0, 1)$	$(0, 0, 1)$	$(0{,}25, 0{,}33)$
1	$(1, 0, 1)$	$(0, 0, 1)$	$(0{,}50, 0{,}33)$
2	$(1, 1, 1)$	$(0, 0, 1)$	$(0{,}50, 0{,}66)$
3	$(0, 1, 1)$	$(0, 0, 1)$	$(0{,}25, 0{,}66)$

2) **Cara Derecha (X = 1)**: Corresponde a la celda ($u \in [0{,}5, 0{,}75], v \in [0{,}33, 0{,}66]$). Normal $\vec{n} = (1, 0, 0)$.

Índice	**Posición** (x, y, z)	**Normal** (nx, ny, nz)	**Textura** (u, v)
4	$(1, 0, 1)$	$(1, 0, 0)$	$(0{,}50, 0{,}33)$
5	$(1, 0, 0)$	$(1, 0, 0)$	$(0{,}75, 0{,}33)$
6	$(1, 1, 0)$	$(1, 0, 0)$	$(0{,}75, 0{,}66)$
7	$(1, 1, 1)$	$(1, 0, 0)$	$(0{,}50, 0{,}66)$

3) **Cara Trasera (Z = 0)**: Corresponde a la celda ($u \in [0{,}75, 1{,}0], v \in [0{,}33, 0{,}66]$). Normal $\vec{n} = (0, 0, -1)$.

Índice	**Posición** (x, y, z)	**Normal** (nx, ny, nz)	**Textura** (u, v)
8	$(1, 0, 0)$	$(0, 0, -1)$	$(0{,}75, 0{,}33)$
9	$(0, 0, 0)$	$(0, 0, -1)$	$(1{,}00, 0{,}33)$
10	$(0, 1, 0)$	$(0, 0, -1)$	$(1{,}00, 0{,}66)$
11	$(1, 1, 0)$	$(0, 0, -1)$	$(0{,}75, 0{,}66)$

4) **Cara Izquierda (X = 0)**: Corresponde a la celda ($u \in [0{,}0, 0{,}25], v \in [0{,}33, 0{,}66]$). Normal $\vec{n} = (-1, 0, 0)$.

Índice	**Posición** (x, y, z)	**Normal** (nx, ny, nz)	**Textura** (u, v)
12	$(0, 0, 0)$	$(-1, 0, 0)$	$(0{,}00, 0{,}33)$
13	$(0, 0, 1)$	$(-1, 0, 0)$	$(0{,}25, 0{,}33)$
14	$(0, 1, 1)$	$(-1, 0, 0)$	$(0{,}25, 0{,}66)$
15	$(0, 1, 0)$	$(-1, 0, 0)$	$(0{,}00, 0{,}66)$

5) **Cara Superior (Y = 1)**: Corresponde a la celda superior central ($u \in [0{,}25, 0{,}5], v \in [0{,}66, 1{,}0]$). Normal $\vec{n} = (0, 1, 0)$.

Índice	**Posición** (x, y, z)	**Normal** (nx, ny, nz)	**Textura** (u, v)
16	$(0, 1, 1)$	$(0, 1, 0)$	$(0{,}25, 0{,}66)$
17	$(1, 1, 1)$	$(0, 1, 0)$	$(0{,}50, 0{,}66)$
18	$(1, 1, 0)$	$(0, 1, 0)$	$(0{,}50, 1{,}00)$
19	$(0, 1, 0)$	$(0, 1, 0)$	$(0{,}25, 1{,}00)$

6) **Cara Inferior (Y = 0)**: Corresponde a la celda inferior central ($u \in [0{,}25, 0{,}5], v \in [0{,}0, 0{,}33]$). Normal $\vec{n} = (0, -1, 0)$.

Índice	**Posición** (x, y, z)	**Normal** (nx, ny, nz)	**Textura** (u, v)
20	$(0, 0, 0)$	$(0, -1, 0)$	$(0{,}25, 0{,}00)$
21	$(1, 0, 0)$	$(0, -1, 0)$	$(0{,}50, 0{,}00)$
22	$(1, 0, 1)$	$(0, -1, 0)$	$(0{,}50, 0{,}33)$
23	$(0, 0, 1)$	$(0, -1, 0)$	$(0{,}25, 0{,}33)$

El código GDScript para definir las nuevas tablas de vértices, normales y coordenadas de textura es el siguiente:

```
# Tabla de posiciones (24 vértices: 6 caras x 4 vértices)
var vertices = [
    # Cara Frontal (Z=1)
    Vector3(0,0,1), Vector3(1,0,1), Vector3(1,1,1), Vector3(0,1,1),
    # Cara Derecha (X=1)
    Vector3(1,0,1), Vector3(1,0,0), Vector3(1,1,0), Vector3(1,1,1),
    # Cara Trasera (Z=0)
    Vector3(1,0,0), Vector3(0,0,0), Vector3(0,1,0), Vector3(1,1,0),
    # Cara Izquierda (X=0)
    Vector3(0,0,0), Vector3(0,0,1), Vector3(0,1,1), Vector3(0,1,0),
    # Cara Superior (Y=1)
    Vector3(0,1,1), Vector3(1,1,1), Vector3(1,1,0), Vector3(0,1,0),
    # Cara Inferior (Y=0)
    Vector3(0,0,0), Vector3(1,0,0), Vector3(1,0,1), Vector3(0,0,1),
]

# Tabla de normales (una por vértice, constante por cara)
var normals = [
    # Frontal
    Vector3(0,0,1), Vector3(0,0,1), Vector3(0,0,1), Vector3(0,0,1),
    # Derecha
    Vector3(1,0,0), Vector3(1,0,0), Vector3(1,0,0), Vector3(1,0,0),
    # Trasera
    Vector3(0,0,-1), Vector3(0,0,-1), Vector3(0,0,-1), Vector3(0,0,-1),
    # Izquierda
    Vector3(-1,0,0), Vector3(-1,0,0), Vector3(-1,0,0), Vector3(-1,0,0),
    # Superior
    Vector3(0,1,0), Vector3(0,1,0), Vector3(0,1,0), Vector3(0,1,0),
```

```
    # Inferior
    Vector3(0,-1,0), Vector3(0,-1,0), Vector3(0,-1,0), Vector3(0,-1,0),
]

# Tabla de coordenadas de textura (UV)
var uvs = [
    # Frontal (u: 0.25-0.5, v: 0.33-0.66)
    Vector2(0.25,0.33), Vector2(0.50,0.33), Vector2(0.50,0.66), Vector2(0.25,0.66),
    # Derecha (u: 0.5-0.75, v: 0.33-0.66)
    Vector2(0.50,0.33), Vector2(0.75,0.33), Vector2(0.75,0.66), Vector2(0.50,0.66),
    # Trasera (u: 0.75-1.0, v: 0.33-0.66)
    Vector2(0.75,0.33), Vector2(1.00,0.33), Vector2(1.00,0.66), Vector2(0.75,0.66),
    # Izquierda (u: 0.0-0.25, v: 0.33-0.66)
    Vector2(0.00,0.33), Vector2(0.25,0.33), Vector2(0.25,0.66), Vector2(0.00,0.66),
    # Superior (u: 0.25-0.5, v: 0.66-1.0)
    Vector2(0.25,0.66), Vector2(0.50,0.66), Vector2(0.50,1.00), Vector2(0.25,1.00),
    # Inferior (u: 0.25-0.5, v: 0.00-0.33)
    Vector2(0.25,0.00), Vector2(0.50,0.00), Vector2(0.50,0.33), Vector2(0.25,0.33),
]

# Tabla de triángulos
var triangles = [
    # Frontal
    0,1,2, 0,2,3,
    # Derecha
    4,5,6, 4,6,7,
    # Trasera
    8,9,10, 8,10,11,
    # Izquierda
    12,13,14, 12,14,15,
    # Superior
    16,17,18, 16,18,19,
    # Inferior
    20,21,22, 20,22,23,
]
```

Ejercicio 1.7.3. Considera un cubo de lado unidad y con centro en $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$. Se quiere visualizar con una textura a partir de una única imagen (cuadrada) que se replicará en las 6 caras de dicho cubo. Asume que no se va a usar iluminación (no es necesario calcular la tabla de normales).

Escribe ahora la tabla de coordenadas de vértices y la tabla de coordenadas de textura necesarias para renderizar este objeto correctamente.

Solución 1.7.3. Para resolver este problema, debemos entender primero cómo funciona el mapeado de texturas en un motor gráfico (como OpenGL o el usado en Godot).

1) **Análisis de la Geometría:** El cubo tiene lado $L = 1$ y su centro es $C = (0{,}5, 0{,}5, 0{,}5)$. Esto implica que las coordenadas espaciales de los vértices varían desde:

$$x_{min} = 0{,}5 - 0{,}5 = 0, \quad x_{max} = 0{,}5 + 0{,}5 = 1$$

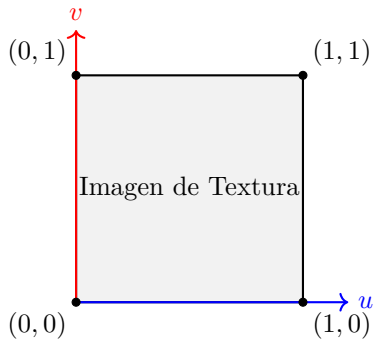
Lo mismo aplica para y y z. Por tanto, el cubo ocupa el volumen $[0, 1]^3$.

2) **El Problema de la Continuidad (Por qué necesitamos 24 vértices):** Un cubo geométrico tiene solo 8 esquinas (vértices físicos). Sin embargo, nos piden replicar la imagen completa en *cada una* de las 6 caras.
Imaginemos la esquina superior derecha de la cara frontal. Sus coordenadas espaciales son $(1, 1, 1)$.
   - Para la **Cara Frontal**, esta esquina corresponde a la coordenada de textura $(u, v) = (1, 1)$ (arriba-derecha de la imagen).
   - Para la **Cara Derecha**, esa misma esquina espacial $(1, 1, 1)$ corresponde a $(u, v) = (0, 1)$ (arriba-izquierda de la imagen).
   - Para la **Cara Superior**, esa esquina corresponde a $(u, v) = (1, 0)$ (abajo-derecha de la imagen, dependiendo de la orientación).

   En informática gráfica, un **vértice** se define por la tupla única de sus atributos: $(Posicion, UV)$. Como una misma posición espacial requiere distintos UVs según la cara que estemos dibujando, debemos **duplicar** los vértices. No podemos usar solo 8 vértices compartidos (mesh indexada simple); necesitamos definir 4 vértices únicos por cada una de las 6 caras.

$$\text{Total de vértices} = 6 \text{ caras} \times 4 \text{ vértices/cara} = 24 \text{ vértices}.$$

3) **Esquema Visual del Mapeado:** A continuación, representamos cómo se asignan las coordenadas (u, v) a una cara genérica para que la imagen se vea derecha (no rotada ni espejada).



4) **Tablas de Definición del Modelo:** Definiremos los vértices cara por cara. Asumiremos el orden de vértices estándar para formar dos triángulos (por ejemplo: 0-1-2 y 0-2-3 para un quad) en sentido antihorario (CCW).

Cara	Índice (i)	Posición (x, y, z)	Coord. Textura (u, v)
	0	$(0, 0, 1)$	$(0, 0)$
	1	$(1, 0, 1)$	$(1, 0)$
	2	$(1, 1, 1)$	$(1, 1)$
	3	$(0, 1, 1)$	$(0, 1)$
	4	$(1, 0, 0)$	$(0, 0)$
	5	$(0, 0, 0)$	$(1, 0)$
	6	$(0, 1, 0)$	$(1, 1)$
	7	$(1, 1, 0)$	$(0, 1)$
	8	$(1, 0, 1)$	$(0, 0)$
	9	$(1, 0, 0)$	$(1, 0)$
	10	$(1, 1, 0)$	$(1, 1)$
	11	$(1, 1, 1)$	$(0, 1)$
	12	$(0, 0, 0)$	$(0, 0)$
	13	$(0, 0, 1)$	$(1, 0)$
	14	$(0, 1, 1)$	$(1, 1)$
	15	$(0, 1, 0)$	$(0, 1)$
	16	$(0, 1, 1)$	$(0, 0)$
	17	$(1, 1, 1)$	$(1, 0)$
	18	$(1, 1, 0)$	$(1, 1)$
	19	$(0, 1, 0)$	$(0, 1)$
	20	$(0, 0, 0)$	$(0, 0)$
	21	$(1, 0, 0)$	$(1, 0)$
	22	$(1, 0, 1)$	$(1, 1)$
	23	$(0, 0, 1)$	$(0, 1)$

Cuadro 1.1: *Tabla Combinada de Vértices y Coordenadas de Textura*

Nota sobre la orientación: En la cara trasera y las laterales, el orden de los vértices y la asignación de (u, v) se ha elegido para mantener la coherencia visual (que la imagen no se vea "espejada") y el orden de los vértices (winding order) sea consistente para el "culling" de caras traseras.

El código GDScript para definir las tablas de vértices y coordenadas de textura es el siguiente:

```
# Tabla de posiciones (24 vértices: 6 caras x 4 vértices)
var vertices = [
    # Frontal (z=1)
    Vector3(0,0,1), Vector3(1,0,1), Vector3(1,1,1), Vector3
    (0,1,1),
    # Trasera (z=0)
    Vector3(1,0,0), Vector3(0,0,0), Vector3(0,1,0), Vector3
    (1,1,0),
    # Derecha (x=1)
    Vector3(1,0,1), Vector3(1,0,0), Vector3(1,1,0), Vector3
```

```
    (1,1,1),
    # Izquierda (x=0)
    Vector3(0,0,0), Vector3(0,0,1), Vector3(0,1,1), Vector3
    (0,1,0),
    # Superior (y=1)
    Vector3(0,1,1), Vector3(1,1,1), Vector3(1,1,0), Vector3
    (0,1,0),
    # Inferior (y=0)
    Vector3(0,0,0), Vector3(1,0,0), Vector3(1,0,1), Vector3
    (0,0,1),
]

# Tabla de coordenadas de textura (UV)
var uvs = [
    # Frontal
    Vector2(0,0), Vector2(1,0), Vector2(1,1), Vector2(0,1),
    # Trasera
    Vector2(0,0), Vector2(1,0), Vector2(1,1), Vector2(0,1),
    # Derecha
    Vector2(0,0), Vector2(1,0), Vector2(1,1), Vector2(0,1),
    # Izquierda
    Vector2(0,0), Vector2(1,0), Vector2(1,1), Vector2(0,1),
    # Superior
    Vector2(0,0), Vector2(1,0), Vector2(1,1), Vector2(0,1),
    # Inferior
    Vector2(0,0), Vector2(1,0), Vector2(1,1), Vector2(0,1),
]

# Tabla de triángulos
var triangles = [
    # Frontal
    0,1,2, 0,2,3,
    # Trasera
    4,5,6, 4,6,7,
    # Derecha
    8,9,10, 8,10,11,
    # Izquierda
    12,13,14, 12,14,15,
    # Superior
    16,17,18, 16,18,19,
    # Inferior
    20,21,22, 20,22,23,
]
```

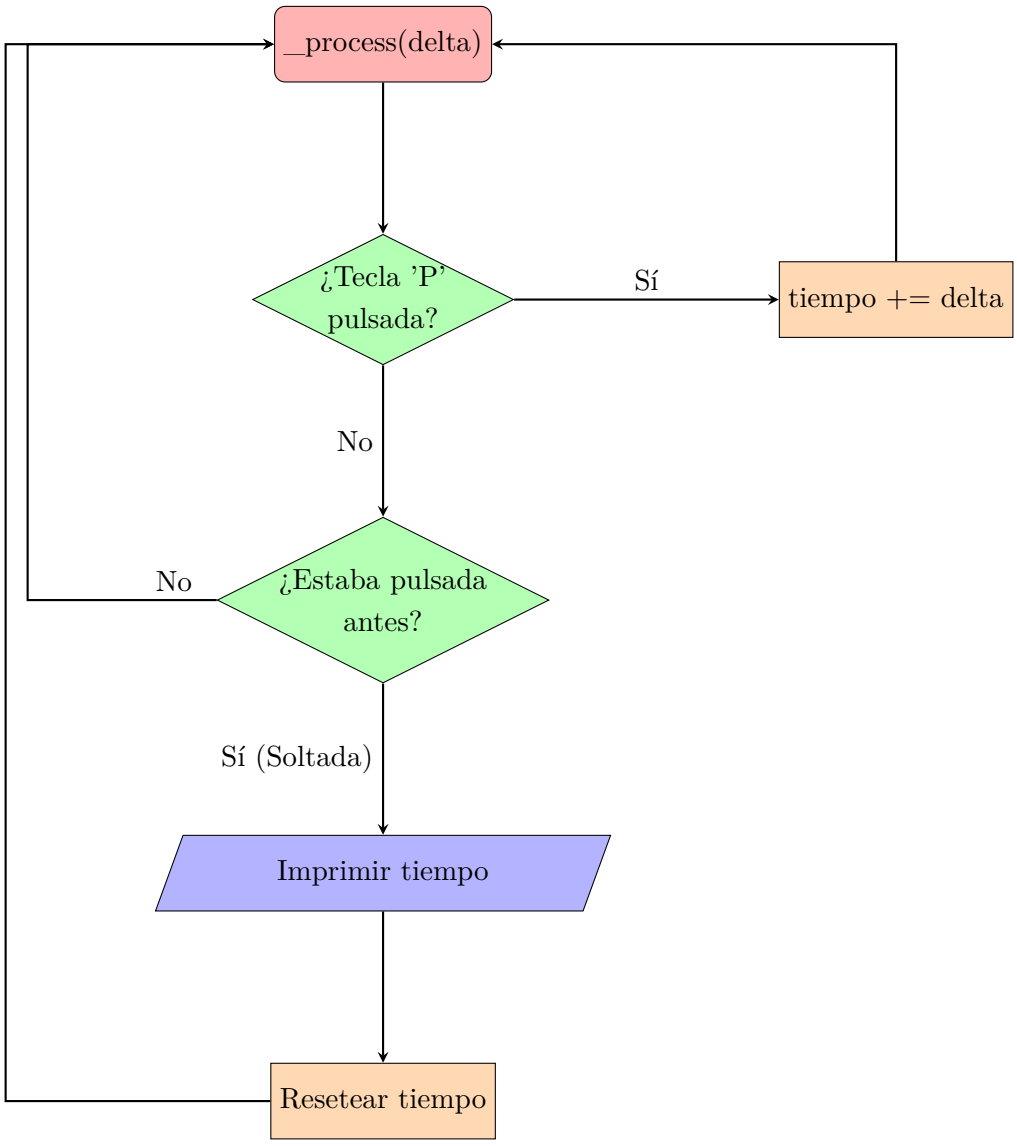
1.8 Sesión 9

Ejercicio 1.8.1. En una aplicación Godot cualquiera, añade código al nodo raíz de forma que cada vez que se pulse y luego se levante una tecla (por ejemplo la tecla P), se imprima en pantalla un mensaje con el tiempo total en segundos que dicha tecla ha estado pulsada, en los casos en los que ha permanecido pulsada al menos el tiempo de un frame.

Solución 1.8.1. Para resolver este problema, debemos comprender cómo funciona el ciclo de vida de un videojuego o aplicación gráfica interactiva en tiempo real. No basta con saber que una tecla ha sido pulsada; necesitamos cuantificar la duración temporal de ese estado.

En Godot, la función `_process(delta)` se ejecuta en cada fotograma (frame). El parámetro `delta` representa el tiempo transcurrido (en segundos) desde el fotograma anterior. Por lo tanto, la estrategia consiste en acumular este valor `delta` mientras la tecla esté presionada y, en el momento exacto en que se libera, mostrar el total acumulado.

A continuación, se presenta el diagrama de flujo lógico que seguiremos para implementar el algoritmo:



Implementación paso a paso:

1) **Definición de variables de estado:** Necesitamos una variable para acumular el tiempo (`tiempo_pulsado`) y una variable booleana (`tecla_activa`) para saber si estamos en medio de una acción de pulsación. Esto es necesario para detectar el evento "just released" (acaba de ser soltada) manualmente o mediante la lógica de estados.
2) **Uso del bucle de procesamiento:** Utilizaremos la función virtual `_process(delta)`, que Godot invoca continuamente.
3) **Lógica de entrada (Input):** Usaremos la clase `Input` para sondear (polling) el estado físico de la tecla 'P' (código `KEY_P`).
4) **Acumulación y Reporte:**
   - Si la tecla está pulsada: Sumamos `delta` a nuestra variable acumuladora.
   - Si la tecla NO está pulsada pero `tecla_activa` es verdadera: Significa que el usuario acaba de soltar la tecla. En ese momento imprimimos el valor y reiniciamos las variables.

Código GDScript Solución:

```
extends Node

# Variable para almacenar el tiempo acumulado en segundos

var tiempo_acumulado: float = 0.0

# Bandera para controlar el estado de la tecla (si se está
    manteniendo pulsada)

var tecla_esta_pulsada: bool = false

func _process(delta: float) -> void:
# Verificamos si la tecla P está siendo presionada en este frame
if Input.is_key_pressed(KEY_P):
# Marcamos que la tecla está activa
tecla_esta_pulsada = true

    # Acumulamos el tiempo transcurrido desde el último frame
    tiempo_acumulado += delta

else:
    # Si la tecla NO está pulsada, verificamos si lo estaba en
    el frame anterior
    # Esto indica el evento ''Just Released'' (Acaba de soltarse
    )
    if tecla_esta_pulsada:

        # Verificamos la condición del enunciado:
        # ''permanecido pulsada al menos el tiempo de un frame''
        # Si tiempo_acumulado > 0, significa que al menos un
    frame sumó delta.
```

```
28          if tiempo_acumulado > 0.0:
29              print(''La tecla P se mantuvo pulsada durante: '',
30                    tiempo_acumulado, '' segundos.'')
31
32          # Reiniciamos el estado para la próxima pulsación
33          tiempo_acumulado = 0.0
34          tecla_esta_pulsada = false
```

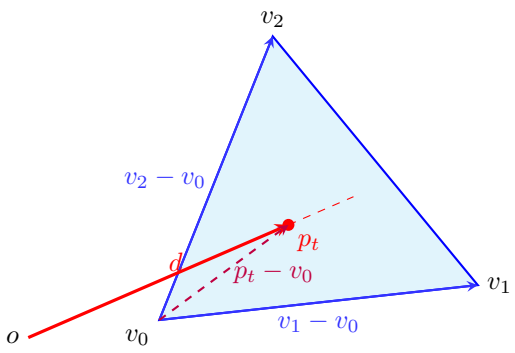
Ejercicio 1.8.2. Una posibilidad para hacer selección en mallas de triángulos es usar cálculo de intersecciones entre un rayo (una semirrecta que pasa por el centro de un píxel) y cada uno de los triángulos de la malla.

Diseña un algoritmo en pseudo-código para el cálculo de intersecciones entre un rayo y un triángulo:

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

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

- El rayo intersecta con el plano del triángulo si y solo si existe $t > 0$ tal que el punto $p_t = o + td$ está en el plano. Esto equivale a que el vector $p_t - v_0$ es perpendicular a la normal del plano n (es decir, su producto escalar es nulo).
- El punto p_t está dentro del triángulo si existen dos valores reales no negativos a y b (con $0 \leq a + b \leq 1$) tales que el vector $p_t - v_0 = a(v_1 - v_0) + b(v_2 - v_0)$. A los tres valores a, b y $c \equiv 1 - a - b$ se les llama coordenadas baricéntricas de p_t en el triángulo.



Solución 1.8.2. Para resolver el problema siguiendo estrictamente las condiciones dadas, el algoritmo se estructura en dos fases secuenciales: encontrar el punto en el plano (Condición 1) y validar si dicho punto está contenido en la región triangular (Condición 2).

Procedimiento detallado:

1) **Cálculo de la Normal del Plano:** Primero, definimos los vectores directores del plano del triángulo basándonos en sus aristas:

$$e_1 = v_1 - v_0$$

$$e_2 = v_2 - v_0$$

La normal n se obtiene mediante el producto vectorial:

$$n = e_1 \times e_2$$

2) **Condición 1: Intersección con el Plano:** Buscamos un t tal que el vector desde v_0 hasta el punto de impacto p_t sea ortogonal a la normal. La ecuación del plano es $(p - v_0) \cdot n = 0$. Sustituyendo la ecuación del rayo $p = o + t \cdot d$:

$$((o + t \cdot d) - v_0) \cdot n = 0$$

$$(o - v_0) \cdot n + t(d \cdot n) = 0$$

Despejando t:

$$t = \frac{(v_0 - o) \cdot n}{d \cdot n}$$

Si $d \cdot n \approx 0$, el rayo es paralelo al plano (no hay intersección). Si $t \leq 0$, el triángulo está detrás del origen.

3) **Condición 2: Inclusión en el Triángulo (Coordenadas Baricéntricas):** Una vez tenemos $p_t = o + t \cdot d$, definimos el vector $w = p_t - v_0$. Según el enunciado, debemos encontrar a y b tales que:

$$w = a \cdot e_1 + b \cdot e_2$$

Esto es un sistema de ecuaciones lineales. Para resolverlo eficientemente usando productos escalares, multiplicamos la ecuación por e_1 y por e_2:

1) $(w \cdot e_1) = a(e_1 \cdot e_1) + b(e_2 \cdot e_1)$
2) $(w \cdot e_2) = a(e_1 \cdot e_2) + b(e_2 \cdot e_2)$

Aplicando la regla de Cramer para despejar a y b:

$$a = \frac{(w \cdot e_1)(e_2 \cdot e_2) - (w \cdot e_2)(e_1 \cdot e_2)}{(e_1 \cdot e_1)(e_2 \cdot e_2) - (e_1 \cdot e_2)^2}$$

$$b = \frac{(e_1 \cdot e_1)(w \cdot e_2) - (e_1 \cdot e_2)(w \cdot e_1)}{(e_1 \cdot e_1)(e_2 \cdot e_2) - (e_1 \cdot e_2)^2}$$

Finalmente, verificamos si $a \geq 0$, $b \geq 0$ y $a + b \leq 1$.

Algoritmo en Pseudo-código:

```
Funcion IntersectarRayoTriangulo(o, d, v0, v1, v2):
// --- Pre-computo de vectores del triangulo ---
Vector3 e1 = v1 - v0
Vector3 e2 = v2 - v0
Vector3 n  = ProductoCruz(e1, e2) // Normal del plano


// --- Condicion 1: Interseccion Rayo-Plano ---

// Calculamos el denominador (d . n)
float det = ProductoPunto(d, n)

```

```
// Si es cercano a 0, el rayo es paralelo al triangulo
Si valor_absoluto(det) < EPSILON:
    Retornar {Falso, Nulo}

// Calculamos t usando la formula derivada: t = ((v0 - o) . n) / det
Vector3 origen_a_v0 = v0 - o
float t = ProductoPunto(origen_a_v0, n) / det

// Verificamos que la interseccion esta delante de la camara (t > 0)
Si t < EPSILON:
    Retornar {Falso, Nulo}

// Calculamos el punto de interseccion en el plano
Vector3 pt = o + (d * t)

// --- Condicion 2: Punto dentro del triangulo ---
// Debemos resolver: pt - v0 = a*e1 + b*e2

Vector3 w = pt - v0

// Calculo de productos punto para el sistema de Cramer
float uu = ProductoPunto(e1, e1)
float uv = ProductoPunto(e1, e2)
float vv = ProductoPunto(e2, e2)
float wu = ProductoPunto(w, e1)
float wv = ProductoPunto(w, e2)

// Denominador del sistema (determinante)
float denominador = (uu * vv) - (uv * uv)

// Si denominador es 0, el triangulo es degenerado (linea o punto)
Si valor_absoluto(denominador) < EPSILON:
    Retornar {Falso, Nulo}

// Calculo de coordenadas baricentricas a y b
float a = ((wu * vv) - (wv * uv)) / denominador
float b = ((uu * wv) - (wu * uv)) / denominador

// Verificacion final de limites baricentricos
// 0 <= a, 0 <= b, a + b <= 1
Si (a >= 0.0) Y (b >= 0.0) Y (a + b <= 1.0):
    Retornar {Verdadero, pt}
Sino:
    Retornar {Falso, Nulo}
```

Solución 1.8.2. Otra resolución alternativa y más detallada es la que se proporciona a continuación. Para resolver este problema, debemos traducir la geometría 3D a una serie de pasos lógicos. No basta con aplicar fórmulas; hay que entender qué significan. El proceso se divide en tres fases: definir la pared (plano), buscar el choque y verificar si el choque está dentro del triángulo.

1. Definición del Plano (La Pared)

Un triángulo es plano. Para saber si un rayo choca con él, primero necesitamos saber la orientación de la pared invisible donde está pegado el triángulo.

- Calculamos dos vectores que bordean el triángulo desde $\mathbf{v}_0$:

$$\mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0, \quad \mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0$$

- La orientación (el vector normal $\mathbf{n}$) es perpendicular a ambos bordes:

$$\mathbf{n} = \mathbf{e}_1 \times \mathbf{e}_2$$

2. El Choque (Cálculo de t)

El rayo es una línea que empieza en $\mathbf{o}$ y avanza en dirección $\mathbf{d}$. La fórmula del impacto en el plano es:

$$t = \frac{(\mathbf{v}_0 - \mathbf{o}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

¿Por qué $t < 0$ significa "detrás"? Imagina que el rayo son tus pasos.

- $\mathbf{t = 0}$ es donde estás parado (el origen).
- $\mathbf{t > 0}$ son pasos hacia adelante (lo que ves).
- $\mathbf{t < 0}$ son pasos hacia atrás (a tu espalda).

La fórmula matemática asume una recta infinita (hacia adelante y atrás). Si el cálculo da $t = -5$, significa que el plano está 5 pasos a tu espalda. Como una cámara solo "ve"hacia adelante, descartamos cualquier $t < 0$.

3. ¿Dentro o Fuera? (Coordenadas Baricéntricas)

Si $t > 0$, el rayo golpea la pared en el punto $\mathbf{p}$. Ahora usamos coordenadas baricéntricas (a, b) para ver si ese punto cae dentro del dibujo del triángulo. Es como preguntar: *"¿Puedo llegar al punto* $\mathbf{p}$ *dando pasos solo a lo largo de los bordes* $\mathbf{e}_1$ *y* $\mathbf{e}_2$ *sin salirme?"*.

Se resuelve el sistema $\mathbf{p} - \mathbf{v}_0 = a\mathbf{e}_1 + b\mathbf{e}_2$. Si $a \geq 0$, $b \geq 0$ y $a + b \leq 1$, estamos dentro.

Algorithm 1 Intersección Rayo-Triángulo

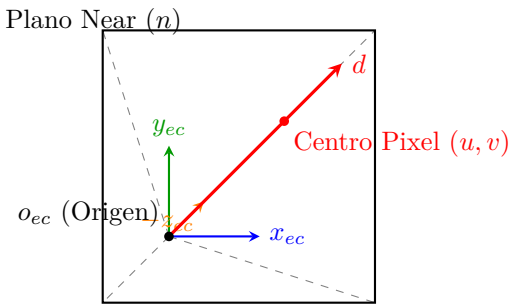
Entrada: Rayo $(\mathbf{o}, \mathbf{d})$, Triángulo $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2)$
Salida: Bool (¿Impacto?), Punto $(\mathbf{p})$
▷ — Fase 1: Preparar vectores —
$\mathbf{e}_1 \leftarrow \mathbf{v}_1 - \mathbf{v}_0$
$\mathbf{e}_2 \leftarrow \mathbf{v}_2 - \mathbf{v}_0$
$\mathbf{n} \leftarrow \mathbf{e}_1 \times \mathbf{e}_2$ ▷ Producto Vectorial (Normal)
▷ — Fase 2: Intersección con el plano —
$det \leftarrow \mathbf{d} \cdot \mathbf{n}$
if $|det| < \epsilon$ **then** ▷ ¿Es el rayo paralelo al plano?
    return Falso, Nulo
end if
vec_origen $\leftarrow \mathbf{v}_0 - \mathbf{o}$
$t \leftarrow (\mathbf{vec\_origen} \cdot \mathbf{n})/det$
if $t < 0$ **then** ▷ Si t es negativo, el triángulo está detrás
    return Falso, Nulo
end if
▷ — Fase 3: Test de inclusión (Baricéntricas) —
$\mathbf{p} \leftarrow \mathbf{o} + (t \cdot \mathbf{d})$ ▷ Punto de impacto en el plano
$\mathbf{w} \leftarrow \mathbf{p} - \mathbf{v}_0$
▷ Resolvemos sistema lineal usando prod. escalares (Cramer)
$uu \leftarrow \mathbf{e}_1 \cdot \mathbf{e}_1$; $uv \leftarrow \mathbf{e}_1 \cdot \mathbf{e}_2$; $vv \leftarrow \mathbf{e}_2 \cdot \mathbf{e}_2$
$wu \leftarrow \mathbf{w} \cdot \mathbf{e}_1$; $wv \leftarrow \mathbf{w} \cdot \mathbf{e}_2$
$D \leftarrow (uu \cdot vv) - (uv \cdot uv)$
$a \leftarrow ((wu \cdot vv) - (wv \cdot uv))/D$
$b \leftarrow ((uu \cdot wv) - (uv \cdot wu))/D$
if $a \geq 0 \wedge b \geq 0 \wedge (a + b \leq 1)$ **then**
    return Verdadero, $\mathbf{p}$ ▷ ¡Impacto confirmado!
else
    return Falso, Nulo ▷ Fuera del triángulo
end if

Ejercicio 1.8.3. Para implementar la selección usando intersecciones es necesario calcular el rayo que tiene como origen el observador y pasa por el centro del pixel donde se ha hecho click.

Escribe el pseudo-código del algoritmo que calcula el rayo a partir de las coordenadas del pixel donde se ha hecho click:

- Tenemos una vista perspectiva, y conocemos los 6 valores usados para construir la matriz de proyección (left, right, top, bottom, near, far).
- También conocemos el marco de coordenadas de vista, es decir, las tuplas y con los versores y la tupla con el punto origen (todos en coordenadas del mundo).
- El viewport tiene columnas y filas de pixels.
- Se ha hecho click en el pixel de coordenadas enteras e .

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



Solución 1.8.3. El objetivo de este ejercicio es realizar el proceso inverso a la rasterización: en lugar de proyectar un punto 3D a un pixel 2D, queremos proyectar un pixel 2D hacia el espacio 3D ("Un-project").

Para ello, debemos transformar las coordenadas del pixel desde el espacio de pantalla al espacio de la cámara (View Space), y finalmente rotar ese vector al espacio del mundo (World Space) usando la base de la cámara dada.

Procedimiento paso a paso:

1) **Mapeo de Pixel a Plano de Imagen (View Plane):** El plano de proyección se encuentra a una distancia (near) de la cámara. Los límites de este plano son en horizontal y en vertical. Los pixels se indexan generalmente desde la esquina superior izquierda $(0, 0)$ hasta $(w, filas)$. Sin embargo, el sistema de coordenadas de la cámara suele tener el eje Y apuntando hacia arriba. Debemos tener cuidado con esta inversión.
   Calculamos las coordenadas físicas (u, v) en el plano near correspondientes al centro del pixel:
   - Sumamos 0,5 a x_p y y_p para apuntar al *centro* del pixel, no a su esquina.
   - Interpolamos linealmente:

$$u = l + (r - l) \cdot \frac{x_p + 0{,}5}{w}$$

$$v = t - (t - b) \cdot \frac{y_p + 0{,}5}{filas}$$

   Nota: Asumimos que $y_p = 0$ es la parte superior (top) y $y_p = filas$ es la inferior (bottom), por eso restamos en v.
2) **Construcción del Vector en Espacio de Cámara:** En el sistema de referencia local de la cámara:
   - El origen del rayo es $(0, 0, 0)$.
   - El rayo atraviesa el plano near en $(u, v, -n)$. (Recordemos que en OpenGL/Godot la cámara mira hacia $-Z$).
   - El vector dirección local es $\vec{d}_{local} = (u, v, -n)$.
3) **Transformación al Espacio del Mundo:** Ahora usamos la base de la cámara dada (x_{ec}, y_{ec}, z_{ec}) para orientar este vector en el mundo.

$$\vec{d}_{mundo} = u \cdot \vec{x}_{ec} + v \cdot \vec{y}_{ec} + (-n) \cdot \vec{z}_{ec}$$

   El origen del rayo o es simplemente la posición de la cámara o_{ec}.
4) **Normalización:** Finalmente, normalizamos el vector dirección resultante.

Algoritmo en Pseudo-código:

```
Funcion CalcularRayoDesdePixel(xp, yp, w, filas, l, r, b, t, n,
    o_ec, x_ec, y_ec, z_ec):

// 1. Calcular coordenadas normalizadas del centro del pixel
    (0.0 a 1.0)
// Sumamos 0.5 para tomar el centro exacto del pixel
float ratio_x = (xp + 0.5) / w
float ratio_y = (yp + 0.5) / filas

// 2. Mapear al tamaño fisico del plano near (View Plane)
// Coordenada u (horizontal): interpolar entre left (l) y right
    (r)
float u = l + ((r - l) * ratio_x)

// Coordenada v (vertical): interpolar entre top (t) y bottom (b
    )
// IMPORTANTE: Asumimos que yp=0 es arriba (top) y yp=filas es
    abajo (bottom)
// Por tanto, a mayor yp, nos acercamos mas a 'b' y nos alejamos
     de 't'
float v = t - ((t - b) * ratio_y)

// 3. Construir el vector de direccion en coordenadas del mundo
// El vector en espacio camara es (u, v, -n)
// Lo transformamos multiplicando por los versores de la base de
     la camara
// d = u*Right + v*Up + (-n)*Back

Vector3 direccion_no_norm = (x_ec * u) + (y_ec * v) - (z_ec * n)

// 4. Normalizar la direccion
Vector3 d = Normalizar(direccion_no_norm)

// 5. El origen del rayo es la posicion de la camara (proyeccion
     perspectiva)
Vector3 o = o_ec

Retornar {o, d}
```

1.9 Sesión 10

Ejercicio 1.9.1. Supongamos que un rayo (una semirrecta en 3D) tiene como origen o extremo el punto cuyas coordenadas del mundo es la tupla **o**, y como vector de dirección la tupla **d** (la suponemos normalizada). Además, sabemos que un disco de radio r tiene como centro el punto de

coordenadas de mundo **c** y está en el plano perpendicular al vector **n**.

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

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

1) El rayo interseca con el plano que contiene al disco, es decir, existe $t > 0$ tal que el punto $p_t \equiv o + td$ está en dicho plano. Equivale a decir que el vector $p_t - c$ es perpendicular a la normal al plano n.
2) El punto p_t citado arriba está dentro del disco, es decir, su distancia a c es inferior al radio.

Solución 1.9.1. Para resolver este problema geométrico fundamental en el trazado de rayos (*ray-tracing*), se debe descomponer la situación en dos etapas lógicas secuenciales. Primero, se determina el punto donde el rayo infinito cruza el plano matemático que contiene al disco. Segundo, se verifica si dicho punto de cruce se encuentra dentro de los límites finitos del disco (es decir, dentro de su radio).

1) **Definición Algebraica del Rayo y el Plano:**
   Un rayo se define paramétricamente como una línea que parte de un origen o y avanza en la dirección d. Cualquier punto $p(t)$ sobre el rayo se puede expresar como:

$$p(t) = o + t \cdot d$$

   Donde t es un escalar real ($t \geq 0$) que representa la distancia desde el origen a lo largo del vector dirección.
   Por otro lado, un plano en el espacio 3D queda definido por un punto conocido (en este caso, el centro del disco c) y un vector normal n perpendicular a la superficie. La condición para que un punto genérico p pertenezca al plano es que el vector formado entre el centro y ese punto sea perpendicular a la normal. Matemáticamente, esto implica que su producto escalar (*dot product*) es cero:

$$(p - c) \cdot n = 0$$

2) **Cálculo del parámetro de intersección t:**
   Para encontrar la intersección, se sustituye la ecuación del rayo en la ecuación del plano:

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

   Aplicando la propiedad distributiva del producto escalar:

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

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

   Despejando t:

$$t(d \cdot n) = (c \cdot n) - (o \cdot n)$$

$$t(d \cdot n) = (c - o) \cdot n$$

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

   Aquí surgen consideraciones críticas de implementación:
   - Si el denominador $d \cdot n$ es igual a 0, significa que el rayo es perpendicular a la normal

del plano (es decir, el rayo es paralelo al plano), por lo que no hay intersección (o el rayo está contenido en el plano).
- Si $t < 0$, la intersección ocurre "detrás" del origen del rayo, por lo que no es visible y debe descartarse.

3) **Cálculo del punto de intersección p_t:**
Una vez obtenido un t válido ($t > 0$), se calcula la coordenada exacta del punto en el espacio:

$$p_t = o + t \cdot d$$

4) **Verificación de pertenencia al disco:**
El hecho de que p_t esté en el plano no garantiza que golpee el disco. Para que haya colisión, la distancia entre el punto de intersección p_t y el centro del disco c debe ser menor o igual al radio r.

$$\|p_t - c\| \leq r$$

Computacionalmente, calcular la raíz cuadrada para el módulo de un vector es costoso. Es preferible comparar los cuadrados de las distancias:

$$v = p_t - c$$

$$v \cdot v \leq r^2$$

$$(v_x^2 + v_y^2 + v_z^2) \leq r^2$$

A continuación, se presenta el algoritmo formal en pseudocódigo:

```
// Estructuras de datos:
// Vec3: tupla (x, y, z) con operaciones de suma, resta y producto punto
// Rayo: origen (Vec3), direccion (Vec3)
// Disco: centro (Vec3), normal (Vec3), radio (float)

bool IntersectaDisco(Rayo ray, Disco disco, float &t_salida) {
    // 1. Calcular el denominador (producto punto entre normal y direccion)
    float denom = dot(disco.normal, ray.direccion);


    // Si denom es cercano a 0, el rayo es paralelo al plano
    if (abs(denom) < 1e-6) {
        return false;
    }

    // 2. Calcular el vector desde el origen del rayo al centro del disco
    Vec3 vector_origen_centro = disco.centro - ray.origen;

    // 3. Calcular t
    float t = dot(vector_origen_centro, disco.normal) / denom;
```

```
21
22      // Verificar si la interseccion esta detras de la camara
23      if (t < 0) {
24          return false;
25      }
26
27      // 4. Calcular el punto exacto de interseccion en el plano
28      Vec3 p = ray.origen + (ray.direccion * t);
29
30      // 5. Verificar si el punto esta dentro del radio del disco
31      Vec3 v = p - disco.centro;
32      float dist_cuadrada = dot(v, v); // |v|^2
33
34      if (dist_cuadrada <= (disco.radio * disco.radio)) {
35          t_salida = t; // Guardamos la distancia a la colision
36          return true;  // Hay interseccion valida
37      }
38
39      return false; // Intersecta el plano, pero fuera del disco
40
41
42
43  }
```

Código 1.1: *Algoritmo de Intersección Rayo-Disco*

Otro formato del algoritmo en pseudocódigo es el siguiente:

Algorithm 2 Intersección Rayo-Disco

function IntersecciónRayoDisco(o, d, c, n, r)
  denom $\leftarrow d \cdot n$
  if |denom| $< \epsilon$ **then**
    return (**FALSO**, NULO)
  end if
  $t \leftarrow \frac{(c-o)\cdot n}{\text{denom}}$
  if $t < 0$ **then**
    return (**FALSO**, NULO)
  end if
  $p \leftarrow o + t \cdot d$
  if $(p-c)\cdot(p-c) \leq r^2$ **then**
    return (**VERDADERO**, p)
  else
    return (**FALSO**, NULO)
  end if
end function

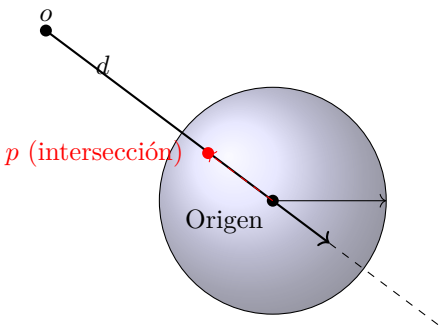
Observación. Sabemos que ε es un valor muy pequeño (por ejemplo, 10^{-6}) para evitar divisiones por cero numéricas.

Ejercicio 1.9.2. Diseña un algoritmo para calcular la primera intersección entre un rayo (con origen en o y vector d, normalizado) y una esfera de radio unidad y centro en el origen, si hay alguna.

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

$$F(p) \equiv p \cdot p - 1$$

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



Solución 1.9.2. Para resolver el problema de la intersección entre un rayo y una esfera unitaria centrada en el origen, se procede algebraicamente sustituyendo la ecuación paramétrica del rayo en la ecuación implícita de la esfera. El objetivo es hallar el valor del parámetro t (distancia desde el origen del rayo) donde ocurre el contacto.

1) **Planteamiento de las ecuaciones:**
   La ecuación del rayo es:
   $$p(t) = o + t \cdot d$$
   donde $t \geq 0$.
   La ecuación implícita de la esfera unitaria centrada en el origen es:
   $$p \cdot p - 1 = 0 \quad \text{(o bien } \|p\|^2 = 1\text{)}$$

2) **Sustitución:**
   Se sustituye $p(t)$ en la ecuación de la esfera:
   $$(o + t \cdot d) \cdot (o + t \cdot d) - 1 = 0$$
   Expandiendo el producto escalar (propiedad distributiva):
   $$(o \cdot o) + 2t(o \cdot d) + t^2(d \cdot d) - 1 = 0$$

3) **Simplificación:**
   Dado que el vector de dirección d está normalizado, sabemos que $d \cdot d = 1$. La ecuación se

convierte en una ecuación cuadrática de la forma $At^2 + Bt + C = 0$:

$$t^2 + 2(o \cdot d)t + (o \cdot o - 1) = 0$$

Identificamos los coeficientes:
- $A = 1$
- $B = 2(o \cdot d)$
- $C = o \cdot o - 1$

4) **Resolución de la ecuación cuadrática:** *Observación.* Aunque la resolución sea trivial, se detalla
Usamos la fórmula general para hallar t:

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

Sustituyendo $A = 1$, $B = 2(o \cdot d)$, $C = o \cdot o - 1$:

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

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

5) **Interpretación del discriminante (Δ):**
El término dentro de la raíz es el discriminante: $\Delta = (o \cdot d)^2 - (o \cdot o - 1)$.
- Si $\Delta < 0$: El rayo no toca la esfera (pasa de largo). No hay solución real.
- Si $\Delta = 0$: El rayo es tangente a la esfera (un punto de contacto).
- Si $\Delta > 0$: El rayo atraviesa la esfera (dos puntos de contacto, entrada y salida).

Se busca la **primera intersección**, que corresponde al menor valor positivo de t. Si ambos t son negativos, la esfera está detrás del origen del rayo.

6) **Generalización para Esfera Arbitraria (Centro C, Radio R):**
Para reutilizar el algoritmo de la esfera unitaria, se aplica una transformación al rayo para llevarlo al "espacio de la esfera unitaria".
La ecuación de una esfera genérica es $\|p - C\|^2 = R^2$, que se puede reescribir como:

$$\left\| \frac{p - C}{R} \right\|^2 = 1$$

Si definimos un nuevo origen de rayo transformado o':

$$o' = \frac{o - C}{R}$$

El problema se reduce a encontrar la intersección de un rayo que parte de o' con dirección d contra la esfera unitaria en el origen. Si el algoritmo base devuelve un parámetro de intersección t_{unit}, la distancia real t_{real} en el mundo original será:

$$t_{real} = t_{unit} \times R$$

Esto se debe a que hemos escalado el espacio dividiendo por R, por lo que las distancias calculadas están "comprimidas" y deben restaurarse multiplicando por R.

A continuación, se presenta el pseudocódigo que implementa esta lógica:

```
// Estructuras auxiliares
struct Rayo { Vec3 origen; Vec3 direccion; }; // direccion normalizada
struct Esfera { Vec3 centro; float radio; };

// Algoritmo Base: Interseccion con Esfera Unitaria en (0,0,0)
// Retorna true si hay colision, y guarda la distancia en t_out
bool IntersectaEsferaUnidad(Vec3 o, Vec3 d, float &t_out) {
    // Coeficientes de la ecuacion t^2 + Bt + C = 0
    // A es 1 porque d esta normalizado
    float B = 2.0f * dot(o, d);
    float C = dot(o, o) - 1.0f;

    float discriminante = (B * B) - (4.0f * C);

    if (discriminante < 0.0f) return false; // No hay interseccion

    float raiz = sqrt(discriminante);

    // Soluciones de la ecuacion
    float t0 = (-B - raiz) / 2.0f; // Entrada (mas cercana)
    float t1 = (-B + raiz) / 2.0f; // Salida (mas lejana)

    // Verificar orden y positividad para encontrar la primera valida
    if (t0 > 0.001f) {
        t_out = t0;
        return true;
    }
    if (t1 > 0.001f) {
        t_out = t1;
        return true; // El origen esta dentro de la esfera
    }

    return false; // Ambas intersecciones estan detras del rayo
}

// Algoritmo General: Reduccion al caso unitario
bool IntersectaEsferaGenerica(Rayo ray, Esfera esf, float &t_real) {
    // 1. Transformar el origen del rayo al espacio de la esfera unitaria
    // Se traslada el mundo para que el centro sea (0,0,0) y se escala por 1/R
    Vec3 o_prima = (ray.origen - esf.centro) / esf.radio;
```

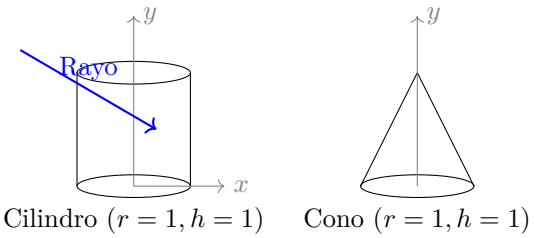
```
41
42      // La direccion d no se escala para mantener la coherencia
        geometrica
43      // del rayo, pero esto implica que el 't' resultante estara
        escalado.
44
45      float t_unit;
46      if (IntersectaEsferaUnidad(o_prima, ray.direccion, t_unit))
        {
47          // 2. Escalar la distancia resultante para volver al
        mundo real
48          t_real = t_unit * esf.radio;
49          return true;
50      }
51
52      return false;
53  }
```

Código 1.2: *Algoritmo de Intersección Rayo-Esfera*

Ejercicio 1.9.3. Se pide:

Parte 1: Cilindro. Describa cómo se puede definir el campo escalar cuyos ceros corresponden a los puntos de un cilindro de altura unidad y radio unidad (sin considerar las tapas). Utilizando esa definición, diseñe un algoritmo para calcular la intersección rayo-cilindro.

Parte 2: Cono. Describa asimismo el campo escalar y el algoritmo correspondientes a un cono de altura unidad y radio de la base unidad (sin considerar el disco de la base).



Cilindro $(r = 1, h = 1)$ Cono $(r = 1, h = 1)$

Solución 1.9.3. Este problema aborda la intersección con superficies cuádricas canónicas (cilindros y conos) acotadas espacialmente. A diferencia de la esfera, estas superficies son infinitas por definición algebraica, por lo que el algoritmo debe incorporar un paso adicional de ”recorte” (*clipping*) para respetar la altura finita. Asumiremos, por convención estándar en gráficos, que ambos objetos están alineados con el eje Y.

1) **Definición del Campo Escalar para el Cilindro:**
   Un cilindro infinito de radio $r = 1$ alineado con el eje Y cumple que, para cualquier punto

$p = (x, y, z)$ en su superficie, la distancia horizontal al eje Y es 1.

$$x^2 + z^2 = 1$$

Por lo tanto, el campo escalar $F_{cyl}(p)$ se define como:

$$F_{cyl}(p) \equiv x^2 + z^2 - 1$$

Los ceros de este campo ($F_{cyl}(p) = 0$) definen la superficie del cilindro infinito. Para obtener el cilindro de altura unidad, se añade la condición de restricción:

$$0 \leq y \leq 1$$

2) **Algoritmo de Intersección Rayo-Cilindro:**
Sea el rayo $p(t) = o + t \cdot d$, donde $o = (o_x, o_y, o_z)$ y $d = (d_x, d_y, d_z)$. Sustituimos las coordenadas del rayo en la ecuación implícita $x^2 + z^2 - 1 = 0$:

$$(o_x + td_x)^2 + (o_z + td_z)^2 - 1 = 0$$

Expandiendo y agrupando términos por potencias de t, obtenemos una ecuación cuadrática $At^2 + Bt + C = 0$:
   - $A = d_x^2 + d_z^2$
   - $B = 2(o_x d_x + o_z d_z)$
   - $C = o_x^2 + o_z^2 - 1$

   Se resuelve para t. Si existen soluciones reales t_0, t_1, se calcula el punto de impacto $p_{hit} = o + t \cdot d$. Finalmente, se descarta la intersección si la componente y de p_{hit} no cumple $0 \leq p_y \leq 1$.
3) **Definición del Campo Escalar para el Cono:**
Un cono infinito alineado con el eje Y, con vértice en el origen y que pasa por $(1, 1, 0)$, tiene una pendiente de 1 ($radio/altura = 1/1$). La relación es que el radio horizontal $\sqrt{x^2 + z^2}$ es igual a la altura y.

$$x^2 + z^2 = y^2$$

El campo escalar $F_{cone}(p)$ es:

$$F_{cone}(p) \equiv x^2 + z^2 - y^2$$

Con la restricción de altura $0 \leq y \leq 1$ (y asumiendo la hoja superior del cono, $y \geq 0$).
4) **Algoritmo de Intersección Rayo-Cono:**
Sustituyendo el rayo en $x^2 + z^2 - y^2 = 0$:

$$(o_x + td_x)^2 + (o_z + td_z)^2 - (o_y + td_y)^2 = 0$$

Esto genera nuevamente coeficientes para la ecuación cuadrática:
   - $A = d_x^2 + d_z^2 - d_y^2$
   - $B = 2(o_x d_x + o_z d_z - o_y d_y)$
   - $C = o_x^2 + o_z^2 - o_y^2$

   Se resuelve para t, se obtiene p_{hit} y se verifica que $0 \leq p_y \leq 1$.

A continuación, el pseudocódigo unificado para ambas estructuras:

```
// TipoObjeto: CILINDRO o CONO
bool IntersectaCuadrica(Rayo ray, TipoObjeto tipo, float &t_out) {
    float A, B, C;
    float ox = ray.origen.x, oz = ray.origen.z, oy = ray.origen.y;
    float dx = ray.direccion.x, dz = ray.direccion.z, dy = ray.direccion.y;
    if (tipo == CILINDRO) {
        // x^2 + z^2 - 1 = 0
        A = dx*dx + dz*dz;
        B = 2*(ox*dx + oz*dz);
        C = ox*ox + oz*oz - 1;
    } else { // CONO
        // x^2 + z^2 - y^2 = 0
        A = dx*dx + dz*dz - dy*dy;
        B = 2*(ox*dx + oz*dz - oy*dy);
        C = ox*ox + oz*oz - oy*oy;
    }

    float discrim = B*B - 4*A*C;
    if (discrim < 0) return false; // No hay interseccion con la superficie infinita

    float raiz = sqrt(discrim);
    float t0 = (-B - raiz) / (2*A);
    float t1 = (-B + raiz) / (2*A);

    // Buscar la interseccion mas cercana que este dentro de la altura
    float t_candidata = t0;
    if (t0 < 0.001) t_candidata = t1;
    if (t_candidata < 0.001) return false;

    // Calcular la altura del punto de impacto
    float y_impacto = oy + t_candidata * dy;

    // VALIDACION DE ALTURA (Clipping)
    // El cilindro y el cono tienen altura 1 (de y=0 a y=1)
    if (y_impacto >= 0.0 && y_impacto <= 1.0) {
        t_out = t_candidata;
        return true;
    }

    // Si t0 falla, probamos con t1 (podria ser que entramos por arriba/abajo)
    // Nota: Esto es necesario si estamos dentro del objeto o para el ''lado lejano''
```

```
    y_impacto = oy + t1 * dy;
    if (t1 > 0.001 && y_impacto >= 0.0 && y_impacto <= 1.0) {
        t_out = t1;
        return true;
    }

    return false;
}
```

Código 1.3: *Algoritmo Genérico para Cuádricas Acotadas*

1.10 Sesión 11

Ejercicio 1.10.1. Implementar un proyecto en Godot en el cual el nodo raíz tiene un script que define dos arrays con: una serie de n puntos $p_0, p_1, \ldots, p_{n-1}$ del plano $y = 0$, y una serie de instantes de tiempo $t_0, t_1, \ldots, t_{n-1}$ (en segundos) con $t_0 = 0$.

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

Solución 1.10.1. Se presenta a continuación la resolución detallada del problema, fundamentada en la teoría de curvas paramétricas y Splines Cúbicos de Hermite expuesta en las diapositivas del curso (páginas 63-91).

1. Fundamentos Teóricos: Interpolación de Hermite a Trozos

Para definir una trayectoria suave que pase por una secuencia de puntos $p_0, \ldots, p_{n-1}$ en tiempos específicos, se utiliza una curva definida a trozos. Para un instante de tiempo t que se encuentra en el intervalo $[t_i, t_{i+1}]$, la posición se obtiene interpolando entre p_i y p_{i+1} considerando las velocidades (tangentes) v_i y v_{i+1} en dichos puntos.

Se definen las siguientes variables auxiliares para el i-ésimo intervalo:

- La duración del intervalo: $s_i = t_{i+1} - t_i$.
- El parámetro normalizado de tiempo: $u = \frac{t - t_i}{s_i}$, donde $0 \leq u \leq 1$.

La ecuación vectorial para la posición $P(t)$ en este intervalo viene dada por la combinación lineal de las bases de Hermite:

$$P(t) = p_i h_{00}(u) + p_{i+1} h_{01}(u) + s_i v_i h_{10}(u) + s_i v_{i+1} h_{11}(u) \tag{1.9}$$

Es crucial notar que las velocidades v se multiplican por la duración del intervalo s_i para ajustar la

magnitud de la tangente al dominio normalizado $[0, 1]$. Las funciones base son:

$$
\begin{aligned}
h_{00}(u) &= 2u^3 - 3u^2 + 1 \\
h_{01}(u) &= -2u^3 + 3u^2 \\
h_{10}(u) &= u^3 - 2u^2 + u \\
h_{11}(u) &= u^3 - u^2
\end{aligned}
$$

2. Cálculo Automático de Velocidades (Tangentes)

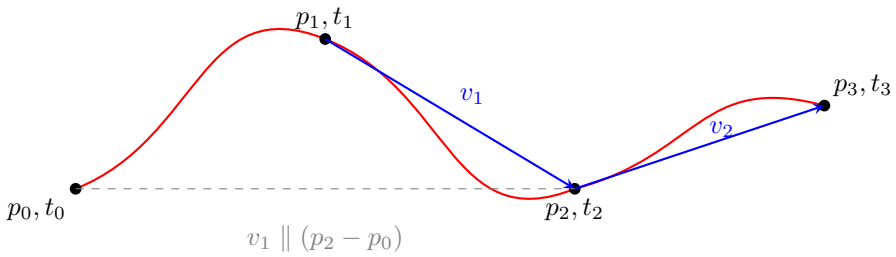
Dado que el enunciado no proporciona las velocidades explícitas, estas se calculan numéricamente para asegurar que la curva sea suave (continuidad C^1) en los puntos de unión. Se utiliza el método de diferencias finitas centradas (Catmull-Rom):

$$
v_i = \frac{p_{i+1} - p_{i-1}}{t_{i+1} - t_{i-1}}, \quad \text{para } 0 < i < n - 1 \tag{1.10}
$$

Para los puntos extremos ($i = 0$ e $i = n - 1$), se asume velocidad nula ($v = 0$) o se puede usar una diferencia simple, pero el enunciado sugiere seguir el ejemplo de suavizado estándar.

3. Representación Visual de la Trayectoria

La siguiente figura ilustra la geometría del problema: los puntos de control (rojos) definen el paso obligado, mientras que los vectores de velocidad calculados (azules) definen la curvatura en dichos puntos.



4. Implementación en GDScript

El siguiente código implementa la lógica completa. Se asume que este script se adjunta al nodo raíz de la escena y que existe un nodo hijo llamado "Coche" (MeshInstance3D o similar).

Código 1.4: *Script de Interpolación Hermite*

```
extends Node3D

# Datos de entrada: Puntos de paso y sus instantes de tiempo
var puntos = [
    Vector3(0, 0, 0),
    Vector3(4, 0, 4),
    Vector3(8, 0, -2),
```

```
    Vector3(12, 0, 5)
]
var tiempos = [0.0, 2.0, 5.0, 8.0] # t0 debe ser 0.0

# Almacen de velocidades calculadas
var velocidades = []

# Referencia al objeto visual (el coche)
onready var objeto_movil = $Coche
var tiempo_actual = 0.0

func _ready():
    # 1. Calcular tangentes automaticamente
    calcular_velocidades_hermite()

    # 2. Visualizar marcadores (discos)
    crear_marcadores_visuales()

func calcular_velocidades_hermite():
    var n = puntos.size()
    velocidades.resize(n)

    # Velocidad 0 en extremos (arranque y parada suave)
    velocidades[0] = Vector3.ZERO
    velocidades[n-1] = Vector3.ZERO

    # Calculo para puntos intermedios: v_i = (p_next - p_prev) /
    (t_next - t_prev)
    for i in range(1, n - 1):
        var dist_vector = puntos[i+1] - puntos[i-1]
        var intervalo_t = tiempos[i+1] - tiempos[i-1]
        velocidades[i] = dist_vector / intervalo_t

func crear_marcadores_visuales():
    for p in puntos:
        var marcador = CSGCylinder3D.new()
        marcador.radius = 0.3
        marcador.height = 0.1
        marcador.material = StandardMaterial3D.new()
        marcador.material.albedo_color = Color(1, 0, 0) # Rojo
        add_child(marcador)
        marcador.global_position = p

# Funcion principal de interpolacion
func obtener_posicion_velocidad(t):
    var n = puntos.size()

    # Caso limite: si t supera el tiempo final
```

```
55      if t >= tiempos[n-1]:
56          return {''pos'': puntos[n-1], ''dir'': Vector3.FORWARD}
57
58      # Buscar el intervalo [i, i+1] correspondiente al tiempo t
59      var i = 0
60      while i < n - 1 and t > tiempos[i+1]:
61          i += 1
62
63      # Datos del tramo actual
64      var p0 = puntos[i]
65      var p1 = puntos[i+1]
66      var v0 = velocidades[i]
67      var v1 = velocidades[i+1]
68      var t0 = tiempos[i]
69      var t1 = tiempos[i+1]
70
71      # Parametro u normalizado (0 a 1)
72      var s = t1 - t0 # Duracion del intervalo
73      var u = (t - t0) / s
74
75      # Pre-calculo de potencias de u
76      var u2 = u * u
77      var u3 = u2 * u
78
79      # Funciones base de Hermite (h00, h10, h01, h11)
80      var h00 = 2*u3 - 3*u2 + 1
81      var h10 = u3 - 2*u2 + u
82      var h01 = -2*u3 + 3*u2
83      var h11 = u3 - u2
84
85      # Interpolacion de la Posicion (notese v * s para escalar la
        tangente)
86      var pos = h00*p0 + h10*s*v0 + h01*p1 + h11*s*v1
87
88      # Calculo de la velocidad instantanea (Derivada de P
       respecto a t)
89      # Derivadas de las bases respecto a u:
90      var dh00 = 6*u2 - 6*u
91      var dh10 = 3*u2 - 4*u + 1
92      var dh01 = -6*u2 + 6*u
93      var dh11 = 3*u2 - 2*u
94
95      # v(t) = P'(u) * (du/dt) = P'(u) * (1/s)
96      var vel = (dh00*p0 + dh10*s*v0 + dh01*p1 + dh11*s*v1) / s
97
98      return {''pos'': pos, ''dir'': vel}
99
100 func _process(delta):
```

```
101     tiempo_actual += delta
102
103     # Reiniciar bucle si termina
104     if tiempo_actual > tiempos.back():
105         tiempo_actual = 0.0
106
107     # Calcular estado fisico
108     var estado = obtener_posicion_velocidad(tiempo_actual)
109
110     # Aplicar transformaciones
111     if objeto_movil:
112         objeto_movil.global_position = estado[''pos'']
113
114         # Orientar el objeto segun el vector de velocidad (
    tangente)
115         # Se evita el error si la velocidad es muy cercana a
    cero
116         if estado[''dir''].length_squared() > 0.001:
117             var objetivo_mirar = estado[''pos''] + estado[''dir'
    ']
118             objeto_movil.look_at(objetivo_mirar, Vector3.UP)
```

Explicación Paso a Paso del Código

1) **Inicialización (`_ready`):** Se calculan las velocidades (tangentes) en cada punto de control usando diferencias centradas, y se crean los marcadores visuales en la escena para cada punto.
2) **Cálculo de Velocidades:** Para los puntos intermedios, la velocidad se obtiene como el cociente entre la diferencia de posiciones y la diferencia de tiempos de los puntos anterior y siguiente. En los extremos, se asigna velocidad cero.
3) **Interpolación Hermite:** La función principal busca el intervalo de tiempo correspondiente y normaliza el parámetro temporal (u) al rango $[0, 1]$. Se aplican las bases polinómicas de Hermite para calcular la posición y la velocidad instantánea en ese tramo.
4) **Actualización por Frame (`_process`):** En cada fotograma, se incrementa el tiempo, se calcula la posición y dirección de la curva en ese instante, y se mueve el objeto (por ejemplo, un coche) a esa posición, orientándolo según la dirección de la curva usando `look_at`.

Ejercicio 1.10.2. Crea un proyecto Godot con una animación de una esfera cuya posición en X oscile periódicamente, con estas condiciones:

1) El centro de la esfera tiene coordenada Z igual a 0, su coordenada Y es igual al radio, y su coordenada X varía entre $-s$ y $+s$, donde $s > 0$ es una constante declarada en el script.
2) El período (tiempo en volver al mismo punto viajando en la misma dirección) es una constante $T > 0$ declarada en el script (con unidades de segundos).
3) La esfera se mueve siempre a velocidad constante en magnitud (es siempre s/T), y el signo depende de la dirección.
4) Tu animación debe producir esa velocidad constante, incluso teniendo en cuenta que los

sucesivos valores de delta pueden cambiar entre frames.

5) Especialmente, la magnitud de la velocidad debe ser constante aunque entre dos frames haya ocurrido un cambio de dirección en un extremo.

Solución 1.10.2. Se aborda la resolución de este problema mediante la programación de un script en GDScript, gestionando manualmente la actualización de la posición en cada fotograma para garantizar una velocidad constante y un rebote preciso en los extremos.

1. Análisis del Movimiento y Velocidad

El movimiento solicitado describe una onda triangular. La esfera oscila entre $-s$ y $+s$. Un ciclo completo (Período T) consiste en el recorrido:

$$0 \to +s \to 0 \to -s \to 0$$

La distancia total recorrida en un ciclo es $D = s + s + s + s = 4s$.

Para que este ciclo se complete exactamente en T segundos con velocidad uniforme, la magnitud de la velocidad (v) debe ser:

$$v = \frac{\text{Distancia Total}}{\text{Tiempo}} = \frac{4s}{T} \tag{1.11}$$

Nota técnica: El enunciado indica entre paréntesis que la velocidad es s/T. Sin embargo, matemáticamente, si la velocidad fuera s/T, el objeto tardaría $4T$ en completar el ciclo en lugar de T. En esta solución se prioriza el cumplimiento del Período T, por lo que se utilizará $v = 4s/T$.

2. Algoritmo de Actualización y Corrección de "Overshoot"

El reto principal en sistemas de tiempo real (como el método `_process` de Godot) es que el tiempo entre frames (`delta`) es variable. Si el objeto está cerca de un extremo (por ejemplo, $x = 4{,}9$ y $s = 5{,}0$) y el siguiente paso es grande (0.2), la posición teórica sería 5,1, excediendo el límite.

Para mantener la velocidad constante y la precisión:

1) Se calcula el desplazamiento propuesto: $\Delta x = v \cdot \delta$.
2) Se suma a la posición actual.
3) Si la nueva posición excede los límites (s o $-s$), se calcula el exceso (*overshoot*).
4) Se "refleja" el exceso hacia adentro del intervalo y se invierte la dirección. Esto simula que el rebote ocurrió en el instante exacto entre los frames.

3. Implementación en GDScript

El siguiente código se debe adjuntar a un nodo en la escena (por ejemplo, un `Node3D`) que contenga un hijo llamado "Esfera" (visualización).

Código 1.5: *Script de Oscilación Triangular Controlada*

```
1 extends Node3D
2
```

```
# Variables de configuracion (exportadas para editar en el
    inspector)
export var s: float = 5.0      # Amplitud maxima (metros)
export var T: float = 2.0      # Periodo completo (segundos)
export var radio: float = 0.5  # Radio visual de la esfera

# Variables de estado
var x_actual: float = 0.0
var direccion: int = 1         # 1: Derecha, -1: Izquierda
var velocidad: float = 0.0     # Magnitud de la velocidad

# Referencia al nodo visual
onready var esfera = $Esfera

func _ready():
    # Calculo de la velocidad necesaria para cumplir el periodo
    T
    # Distancia total por ciclo = 4 * s
    if T > 0:
        velocidad = (4.0 * s) / T
    else:
        velocidad = 0.0

    # Ajuste visual inicial
    if esfera:
        # Si es un CSGSphere3D, ajustamos el radio propiedad
        if ''radius'' in esfera:
            esfera.radius = radio
        # Posicion inicial
        esfera.position = Vector3(0, radio, 0)

func _process(delta):
    # 1. Calcular el paso teorico en este frame
    var distancia_paso = velocidad * delta

    # 2. Aplicar movimiento
    x_actual += distancia_paso * direccion

    # 3. Verificacion de limites y correccion de rebote

    # Limite derecho (+s)
    if x_actual > s:
        var exceso = x_actual - s
        x_actual = s - exceso   # Reflejar el exceso hacia atras
        direccion = -1          # Invertir direccion

    # Limite izquierdo (-s)
    elif x_actual < -s:
```

```
49          var exceso = -s - x_actual # Cuanto nos pasamos por la
    izquierda
50          x_actual = -s + exceso     # Reflejar el exceso hacia
    delante
51          direccion = 1              # Invertir direccion
52
53      # 4. Actualizar la posicion del nodo visual
54      if esfera:
55          esfera.position.x = x_actual
56          esfera.position.y = radio
57          esfera.position.z = 0.0
```

Este algoritmo asegura que la magnitud de la velocidad se mantenga constante en todo momento, respetando la física del rebote perfecto sin perder tiempo ni energía en los extremos.

Ejercicio 1.10.3. Desarrolla un proyecto Godot para el ejemplo de animación de un reloj con tres agujas. Las condiciones especificadas en la teoría son:

1) Se desea visualizar un reloj con tres agujas: horas, minutos y segundos.
2) Cada aguja se modela como una malla de polígonos en posición vertical (paralelo al eje Y), con el origen en el punto del eje del reloj.
3) Se usan matrices de rotación en torno al eje Z.
4) Los ángulos de rotación dependen linealmente del tiempo t (segundos transcurridos desde el comienzo del día).

Solución 1.10.3. Se procede a la implementación de un sistema de animación jerárquica para simular un reloj analógico funcional en tiempo real. La solución se basa en la aplicación directa de las transformaciones de rotación descritas en las diapositivas 32 a 35 del material de curso.

1. Modelo Matemático: Ángulos y Tiempo

Según la teoría, el estado de las agujas está determinado por tres ángulos $\theta_h, \theta_m, \theta_s$ que son funciones lineales del tiempo t. El tiempo t representa los segundos totales transcurridos en el ciclo actual (el ciclo de 12 horas para la aguja horaria).

Las relaciones angulares (en radianes) son:

- **Segundero (θ_s):** Da una vuelta completa (2π) cada 60 segundos.

$$\theta_s(t) = \frac{2\pi}{60} \cdot t \tag{1.12}$$

- **Minutero (θ_m):** Da una vuelta completa cada hora ($60^2 = 3600$ segundos).

$$\theta_m(t) = \frac{2\pi}{3600} \cdot t \tag{1.13}$$

- **Horario (θ_h):** Da una vuelta completa cada 12 horas ($12 \cdot 60^2 = 43200$ segundos).

$$\theta_h(t) = \frac{2\pi}{43200} \cdot t \tag{1.14}$$

Nota de implementación: En la convención estándar matemática y de Godot, una rotación positiva en el eje Z es antihoraria. Dado que los relojes giran en sentido horario, aplicaremos el signo negativo a estos ángulos en el código (rotación $= -\theta$).

2. Estructura del Grafo de Escena

Para cumplir con el requisito de que las agujas tengan su origen en el eje de rotación pero se extiendan a lo largo del eje Y positivo, utilizaremos una jerarquía de nodos:

1) **Nodo Raíz (Reloj):** Contenedor principal.
2) **Pivotes (Node3D):** Tres nodos hijos situados en $(0, 0, 0)$. Estos nodos serán los que roten.
3) **Mallas (MeshInstance3D):** Hijos de los pivotes. Se desplazarán verticalmente (offset) para que su base coincida con el pivote, logrando el efecto de girar desde el extremo.

3. Implementación en GDScript

El siguiente script crea la geometría procedimentalmente (para facilitar la prueba sin modelos externos) y aplica la lógica de rotación basada en la hora del sistema.

Código 1.6: *Script del Reloj Analógico*

```
extends Node3D

# Referencias a los nodos de las agujas (Pivotes)
var pivote_segundos: Node3D
var pivote_minutos: Node3D
var pivote_horas: Node3D

func _ready():
    # 1. Construccion procedimental de la escena
    crear_geometria_reloj()

func crear_geometria_reloj():
    # Creamos una esfera central como base
    var esfera = CSGSphere3D.new()
    esfera.radius = 0.5
    add_child(esfera)

    # Creamos las tres agujas.
    # Usamos una funcion auxiliar para configurar: (Nombre, Largo, Ancho, Color)
    pivote_horas = crear_aguja(''Horas'', 2.0, 0.2, Color.black)
    pivote_minutos = crear_aguja(''Minutos'', 3.0, 0.15, Color.darkgray)
    pivote_segundos = crear_aguja(''Segundos'', 3.5, 0.05, Color.red)

func crear_aguja(nombre, largo, ancho, color) -> Node3D:
```

```
25      # 1. El Pivote: Este nodo estara en (0,0,0) y es el que
        rotamos
26      var pivote = Node3D.new()
27      pivote.name = ''Pivote'' + nombre
28      add_child(pivote)
29
30      # 2. La Malla Visual: Hija del pivote
31      var mesh = CSGBox3D.new()
32      mesh.size = Vector3(ancho, largo, 0.1)
33
34      # IMPORTANTE: Desplazamos la malla hacia arriba (Y+) la
        mitad de su largo.
35      # Asi, el centro de rotacion (el pivote) queda en la base de
         la aguja.
36      mesh.position = Vector3(0, largo / 2.0, 0)
37
38      # Material
39      var material = StandardMaterial3D.new()
40      material.albedo_color = color
41      mesh.material = material
42
43      pivote.add_child(mesh)
44      return pivote
45
46  func _process(delta):
47      # 1. Obtener el tiempo actual del sistema
48      var tiempo = Time.get_time_dict_from_system()
49      var horas = tiempo[''hour'']
50      var minutos = tiempo[''minute'']
51      var segundos = tiempo[''second'']
52
53      # 2. Calcular t (segundos totales desde las 12:00)
54      # Ajustamos horas a formato 12h para la formula
55      horas = horas % 12
56
57      # Calculo de alta precision para movimiento suave (
        incluyendo milisegundos si se quisiera)
58      # t para segundos (ciclo 60s)
59      var t_sec = segundos
60      # t para minutos (ciclo 3600s). Sumamos segundos para
        movimiento continuo
61      var t_min = (minutos * 60.0) + segundos
62      # t para horas (ciclo 43200s). Sumamos minutos y segundos
63      var t_hour = (horas * 3600.0) + (minutos * 60.0) + segundos
64
65      # 3. Calcular angulos (Theta) usando las formulas de la
        teoria
66      # Theta = (2 * PI / Periodo) * t
```

```
67      # Usamos negativo para rotacion en sentido horario (
        Clockwise)
68
69      var theta_s = -(2.0 * PI / 60.0) * t_sec
70      var theta_m = -(2.0 * PI / 3600.0) * t_min
71      var theta_h = -(2.0 * PI / 43200.0) * t_hour
72
73      # 4. Aplicar rotacion en el eje Z
74      if pivote_segundos:
75          pivote_segundos.rotation.z = theta_s
76      if pivote_minutos:
77          pivote_minutos.rotation.z = theta_m
78      if pivote_horas:
79          pivote_horas.rotation.z = theta_h
```

4. Análisis del Código

1) **Generación de Geometría:** Se sigue la especificación de la diapositiva 35: un nodo raíz (la esfera central) y un nodo para cada aguja. Dentro de cada aguja, se separa la transformación (el Pivote) de la geometría (la Malla). El desplazamiento `mesh.position.y = largo / 2.0` es crítico para que la rotación ocurra en el extremo de la aguja y no en su centro geométrico.
2) **Cálculo del Tiempo (t):** En lugar de un acumulador simple `delta`, utilizamos `Time.get_time_dict_from_system()`. Esto sincroniza la animación con la hora real. Para las agujas de minutos y horas, se suman las fracciones correspondientes de las unidades menores (por ejemplo, a los minutos se le suman los segundos convertidos) para lograr un movimiento fluido y realista, en lugar de saltos discretos.
3) **Aplicación de la Rotación:** Se asignan los ángulos calculados a la propiedad `rotation.z`. El signo negativo asegura que el giro sea en el sentido de las agujas del reloj, corrigiendo la convención matemática estándar (antihoraria) del sistema de coordenadas de Godot.

Ejercicio 1.10.4. Desarrolla un proyecto Godot para el ejemplo de animación de un péndulo. Las condiciones teóricas especificadas son:

1) El péndulo consiste en una masa colgando de un punto fijo por una cuerda de longitud l.
2) El ángulo θ entre la cuerda y la vertical varía con el tiempo t.
3) La oscilación es periódica con un período $T > 0$ (tiempo en segundos para completar un ciclo).
4) El ángulo oscila entre $-\theta_m$ y θ_m.
5) La función que describe el ángulo es $\theta(t) = \theta_m \cdot \sin\left(\frac{2\pi t}{T}\right)$ (o una variante cosinusoidal equivalente).

Solución 1.10.4. Se detalla a continuación la implementación de un péndulo físico simple utilizando animación procedimental en Godot. La solución aplica las fórmulas de oscilación armónica descritas en las diapositivas 36 a 38 del material de referencia.

1. Modelo Matemático del Movimiento

El movimiento del péndulo se modela mediante una función sinusoidal que define el ángulo de rotación $\theta(t)$ en el eje Z.

Según la teoría proporcionada:

- Se define una función base oscilante $f(t) = \sin(\pi t)$ que tiene un período de 2 unidades.
- Para adaptar esta función a un período arbitrario T, se escala el tiempo: $\theta(t) = \theta_{max} \cdot f(\frac{2t}{T})$.

Sustituyendo la función base, obtenemos la fórmula final para la implementación:

$$\theta(t) = \theta_{max} \cdot \sin\left(\pi \cdot \frac{2t}{T}\right) = \theta_{max} \cdot \sin\left(\frac{2\pi t}{T}\right) \tag{1.15}$$

Donde:

- θ_{max} es la amplitud máxima (en radianes).
- T es el período de oscilación (en segundos).
- t es el tiempo acumulado.

2. Estructura del Grafo de Escena

Para simular correctamente el péndulo, es fundamental establecer la jerarquía de nodos adecuada, ya que la rotación debe ocurrir en el punto de anclaje (extremo superior) y no en el centro de masa del péndulo.

1) **Nodo Raíz (Soporte):** Punto fijo en el espacio.
2) **Pivote (Node3D):** Hijo del soporte. Este nodo se ubica en $(0, 0, 0)$ relativo al soporte y es el que recibirá la rotación $\theta(t)$.
3) **Varilla (MeshInstance3D):** Hija del Pivote. Se desplaza verticalmente hacia abajo (eje Y negativo) una distancia $L/2$ y se escala para tener longitud L.
4) **Masa/Bob (MeshInstance3D):** Hija del Pivote (o de la varilla). Se desplaza verticalmente hacia abajo una distancia L.

3. Implementación en GDScript

El siguiente script se debe adjuntar al nodo **Pivote**. Este script genera la geometría visual procedimentalmente para facilitar la prueba y aplica la fórmula de oscilación.

Código 1.7: *Script del Péndulo Oscilante*

```
extends Node3D

# Parametros fisicos configurables
export var theta_max_degrees: float = 45.0 # Amplitud maxima en
    grados
export var periodo: float = 2.0            # Periodo T en
    segundos
export var longitud_cuerda: float = 3.0    # Longitud L

```

```
# Variables internas
var tiempo_acumulado: float = 0.0
var theta_max_rad: float = 0.0

# Referencias a los nodos visuales (se crearan por codigo si no existen)
var varilla: CSGBox3D
var masa: CSGSphere3D

func _ready():
    # Convertir grados a radianes para las funciones trigonometricas
    theta_max_rad = deg_to_rad(theta_max_degrees)

    # Construccion procedimental del pendulo visual
    construir_geometria()

func construir_geometria():
    # 1. Crear la varilla (Cuerda)
    varilla = CSGBox3D.new()
    varilla.size = Vector3(0.1, longitud_cuerda, 0.1) # Grosor y largo

    # IMPORTANTE: Desplazar la varilla hacia abajo la mitad de su longitud.
    # Asi, el extremo superior coincide con el origen del Pivote (0,0,0).
    varilla.position = Vector3(0, -longitud_cuerda / 2.0, 0)

    # Material visual para la varilla
    var mat_varilla = StandardMaterial3D.new()
    mat_varilla.albedo_color = Color.gray
    varilla.material = mat_varilla

    add_child(varilla)

    # 2. Crear la masa (Esfera en el extremo)
    masa = CSGSphere3D.new()
    masa.radius = 0.4

    # La masa se coloca al final de la cuerda
    masa.position = Vector3(0, -longitud_cuerda, 0)

    # Material visual para la masa
    var mat_masa = StandardMaterial3D.new()
    mat_masa.albedo_color = Color.red
    masa.material = mat_masa

```

```
    add_child(masa)

func _process(delta):
    # 1. Acumular el tiempo
    tiempo_acumulado += delta

    # Opcional: Evitar desbordamiento de float reseteando cada
    periodo
    if tiempo_acumulado > periodo:
        tiempo_acumulado -= periodo

    # 2. Calcular el angulo actual usando la formula armonica
    # theta(t) = theta_max * sin(2 * PI * t / T)
    var theta = theta_max_rad * sin((2.0 * PI * tiempo_acumulado
    ) / periodo)

    # 3. Aplicar la rotacion al Pivote
    # Se rota en el eje Z para oscilar izquierda-derecha
    rotation.z = theta
```

4. Explicación del Código

- **Setup (`_ready`):** Se convierten los grados a radianes, ya que las funciones matemáticas de Godot y la propiedad `rotation` trabajan en radianes. Se invoca la construcción de la malla.
- **Geometría (`construir_geometria`):** Se crean primitivas CSG. El punto clave es `varilla.position.y = -longitud_cuerda / 2.0`. Esto asegura que, aunque el centro geométrico del cubo está en su mitad, visualmente la varilla "cuelga" del nodo padre (el Pivote en 0,0,0). La masa se coloca en `-longitud_cuerda`.
- **Animación (`_process`):**
  1) Se actualiza el tiempo t.
  2) Se calcula el valor de la función seno, que oscilará suavemente entre -1 y $+1$.
  3) Se multiplica por θ_{max} para escalar la oscilación a la amplitud deseada.
  4) Se asigna directamente a `rotation.z`. Al ser este nodo el padre de la varilla y la masa, ambos rotarán rígidamente alrededor del punto de anclaje, simulando la física del péndulo.

Ejercicio 1.10.5. Desarrolla un proyecto Godot para el ejemplo de animación de una bala de cañón. Las condiciones y supuestos teóricos son:

1) La bola sale del cañón en una posición inicial $p(0)$ y con una velocidad inicial conocida v_0.
2) La bola está sujeta a la gravedad ($g = 9{,}8$ m/s^2).
3) No se consideran efectos de fricción con el aire.
4) La animación simula la trayectoria hasta que la bola vuelve a la altura inicial.
5) Se utiliza la ecuación de la curva paramétrica: $p(t) = p(0) + v_0 t + \frac{1}{2}at^2$, donde $a = (0, -g, 0)$.

Solución 1.10.5. Se presenta la implementación de la trayectoria parabólica de un proyectil. A diferencia de las simulaciones físicas que integran la velocidad frame a frame (Euler), este ejercicio

pide implementar la solución analítica exacta (curva paramétrica) dependiente del tiempo acumulado t.

1. Modelo Físico-Matemático

La posición $p(t)$ en el instante t se calcula mediante la fórmula vectorial del movimiento uniformemente acelerado:

$$p(t) = p_0 + v_0 \cdot t + \frac{1}{2} \cdot \vec{a} \cdot t^2 \tag{1.16}$$

Desglosando los componentes:

- **Vector aceleración ($\vec{a}$):** Si la gravedad actúa hacia abajo en el eje Y, entonces $\vec{a} = (0, -9{,}8, 0)$.
- **Vector velocidad inicial (v_0):** (v_x, v_y, v_z). Es crucial que $v_y > 0$ para que haya un arco parabólico.
- **Duración del vuelo:** El proyectil vuelve a la altura $y = 0$ (suponiendo $p_0 = 0$) en el instante $t_{fin} = \frac{2v_{0y}}{g}$.

2. Configuración de la Escena

1) **Nodo Raíz (Node3D):** Controlador de la escena.
2) **Suelo (CSGBox3D):** Referencia visual estática.
3) **Bala (MeshInstance3D o CSGSphere3D):** El objeto móvil. Inicialmente en $(0, 0, 0)$.

3. Implementación en GDScript

El siguiente script controla la posición absoluta de la bala basándose en el tiempo transcurrido desde el disparo.

Código 1.8: *Script de Trayectoria Balística Paramétrica*

```
extends Node3D

# Parametros de lanzamiento (Vector3)
# v_y debe ser positiva para que suba.
# v_z o v_x dan el desplazamiento horizontal.
export var velocidad_inicial: Vector3 = Vector3(0, 15, 10)
export var gravedad: float = 9.8

# Variables de estado
var tiempo_vuelo: float = 0.0
var posicion_inicial: Vector3
var vector_gravedad: Vector3

# Referencia al objeto visual
onready var bala = $Bala

func _ready():
```

```
    # Guardamos la posicion original para reiniciar el ciclo
    if bala:
        posicion_inicial = bala.global_position
    else:
        posicion_inicial = Vector3.ZERO

    # Pre-calculamos el vector de aceleracion
    vector_gravedad = Vector3(0, -gravedad, 0)

    # Configuracion visual opcional (crear bala si no existe)
    if not bala:
        crear_bala_procedimental()

func crear_bala_procedimental():
    var mesh = CSGSphere3D.new()
    mesh.radius = 0.5
    mesh.name = ''Bala''
    add_child(mesh)
    bala = mesh
    mesh.global_position = posicion_inicial

    # Material rojo para visibilidad
    var mat = StandardMaterial3D.new()
    mat.albedo_color = Color(1, 0, 0)
    mesh.material = mat

func _process(delta):
    # 1. Acumular el tiempo real transcurrido
    tiempo_vuelo += delta

    # 2. Calcular la posicion usando la formula parametrica
    exacta:
    # p(t) = p0 + v0*t + 0.5 * a * t^2
    var desplazamiento_vel = velocidad_inicial * tiempo_vuelo
    var desplazamiento_acel = 0.5 * vector_gravedad * pow(
    tiempo_vuelo, 2)

    var nueva_posicion = posicion_inicial + desplazamiento_vel +
     desplazamiento_acel

    # 3. Aplicar al objeto
    if bala:
        bala.global_position = nueva_posicion

    # 4. Logica de reinicio (Loop)
    # Si la bala cae por debajo de la altura inicial y ha pasado
     algo de tiempo
    if nueva_posicion.y < posicion_inicial.y and tiempo_vuelo >
```

```
    0.1:
62          reiniciar_animacion()
63
64  func reiniciar_animacion():
65      tiempo_vuelo = 0.0
66      if bala:
67          bala.global_position = posicion_inicial
68
69      # Opcional: Imprimir duracion teorica vs real
70      # T_teorico = 2 * Vy / g
71      # var t_teorico = (2.0 * velocidad_inicial.y) / gravedad
72      # print(''Ciclo completado. T esperado: '', t_teorico)
```

4. Análisis del Código

- **Cálculo Vectorial:** Se aprovecha la capacidad de Godot para operar con vectores completos (`Vector3`). La línea `velocidad_inicial * tiempo_vuelo` escala todas las componentes simultáneamente.
- **Gravedad:** Se aplica como un vector constante hacia abajo $(0, -9{,}8, 0)$. El término cuadrático (t^2) es lo que genera la forma parabólica característica: el movimiento horizontal es lineal (velocidad constante), mientras que el vertical se frena y luego acelera hacia abajo.
- **Reinicio:** La condición `nueva_posicion.y < posicion_inicial.y` detecta cuándo el proyectil ha completado el arco y cruza el plano del suelo, momento en el que se resetea el tiempo $t = 0$ para repetir la animación en bucle.