Teoría y Práctica

Informática Gráfica

https://ismael-sallami.github.io

https://elblogdeismael.github.io

Autor: **Ismael Sallami Moreno**

UNIVERSIDAD DE GRANADA

28 de septiembre de 2025

Licencia

Índice general

Índice de figuras

Índice de cuadros

Parte I

Teoría

Capítulo 1

Introducción

Para acceder a los materiales debemos de entrar con la cuenta go.

La asignatura de Informática Gráfica tiene como objetivo principal proporcionar los fundamentos teóricos y prácticos necesarios para el desarrollo de aplicaciones gráficas interactivas. A lo largo del curso, se estudian conceptos clave como la representación y modelado de escenas, técnicas de visualización 2D y 3D, y el uso de APIs gráficas modernas. Además, se exploran algoritmos esenciales como la rasterización y el ray-tracing, así como su aplicación en contextos como videojuegos, simuladores y producción de efectos visuales.

Capítulo 2

Aplicaciones gráficas interactivas y visualización

2.1 Aplicaciones gráficas interactivas y proceso de visualización 2D y 3D

Un programa gráfico se define como un programa que constituye un sistema computacional. Pueden ser interactivos o no interactivos. Los elementos esenciales de una AG son los modelos digitales y las imágenes o vídeos digitales que se usan.
Destacamos los eventos de entrada que son las acciones del usuario mediante las cuales se envía información a la aplicación. Las aplicaciones gráficas siempre se estructuran como un bucle de gestión de eventos, podemos mencionar los siguientes pasos:

1. Esperar al evento y recuperar datos.

2. Procesar el evento actualizando el modelo y los parámetros de visualización.

3. Visualizar el modelo actualizado con los nuevos parámetros.

Las aplicaciones gráficas pueden dividirse en dos tipos:

- 2D: los objetos se definen en planos, pueden incluir algunos 3D (sombras). Ejemplos de ello puede ser un diagramas de barras, videojuegos 2D. En este proceso de visualización se produce una imagen a partir de un modelo y parámetros como entradas.

- 3D: se sitúan en un espacio tridimensional, incluyendo texturas, materiales, fuentes de luz,... A la vez, pueden incluir figuras 2D. Ejemplos: videojuegos, simuladores,... En este proceso de visualización se usa como entrada el modelo de escena y unos parámetros.
  - En el modelo de escena distinguimos dos partes:
    - Modelo geométrico: conjunto de primitivas (polígonos, planos) que definen los objetos a visualizar.

    - Modelo de aspecto: parámetros que definen el aspecto de los objetos.

  - En los parámetros de visualización encontramos:
    - Cámara virtual

    - Viewport

2.2 Rasterización versus ray-tracing

En este apartado vamos a ver algoritmos de rasterización.

```
1: Inicializar el color de todos los pixels al color de fondo.
2: for cada primitiva P del conjunto E do
3:     S ← conjunto de pixels de la imagen I cubiertos por P
4:     for cada pixel q de S do
5:         c ← color de la primitiva P en el pixel q
6:         Asignar el color c al pixel q en I
7:     end
8: end
```

Este pseudocódigo describe el proceso básico de rasterización, que es una técnica utilizada para convertir primitivas geométricas (como polígonos) en una imagen pixelada. El algoritmo recorre cada primitiva del conjunto de entrada, determina qué píxeles de la imagen están cubiertos por ella, calcula el color correspondiente para cada píxel y lo asigna a la imagen final. Es un enfoque eficiente para generar imágenes en aplicaciones gráficas interactivas.

```
1: Inicializar el color de todos los pixels
2: for cada pixel q de la imagen I a producir do
3:     T ← subconjunto de primitivas de E que cubren q
4:     for cada primitiva P del conjunto T do
5:         c ← color de la primitiva P en el pixel q
6:         Asignar color c al pixel q en I
7:     end
8: end
```

Este pseudocódigo describe el proceso básico del algoritmo de Ray-tracing. A diferencia de la rasterización, aquí se invierte el orden de los bucles: se recorre cada píxel de la imagen y se determina qué primitivas geométricas lo afectan. Luego, se calcula el color del píxel en función de las primitivas que lo cubren. Este enfoque permite generar imágenes con mayor realismo, aunque suele ser más costoso computacionalmente.

En el algoritmo de Ray-tracing podemos optimizarlo de manera que la eficiencia sea O(log n) mediante la indexación espacial.

2.2.1 Rasterización

Se lleva a cabo en GPUs. Es preferible para aplicaciones interactivas y para la simulación de videojuegos, realidad virtual.

2.2.2 Ray-tracing

Respecto a la técnica de Ray-tracing:

- Suele ser más lento, pero consigue resultados más realistas.
- Preferibles para elementos no interactivos. En la actualidad se usa para la producción de animaciones y efectos especiales.
- Se ha usado en algunos videojuegos, pero requiere elementos computacionales de alto rendimiento.

2.2.3 El cauce gráfico en rasterización

Cauce gráfico se define como el conjunto de etapas de cálculo para la generación de imágenes. Las entradas se definen como primitivas. Un vértice es un punto 2D o 3D. EL cauce escribe en el framebuffer.

Hay dos pasos importantes:

1. Transformación: partiendo de las coordenadas se calculan las coordenadas de proyección.
2. Sombreado: cálculo del color de un pixel.
3. Hay otras como recortado de polígonos.

Cada primitiva se sitúa en un plano imaginario (plano de visión) situado entre el observador y la escena. La proyección puede ser perspectiva o paralela. En la rasterización, para cada primitiva se calcula que píxeles tienen su centro cubierto por ella. En el sombreado se usan los atributos de la primitiva para asignar color al píxel.

Etapas del cauce gráfico

1. Procesado de vértices
    1. Transformación: los vértices de cada primitiva se transforman para encontrar su proyección en el plano.
    2. Teselación y nivel de detalle: transformaciones avanzadas.
2. Post-procesado de vértices y montaje de primitivas
3. Rasterización

4. Sombreado

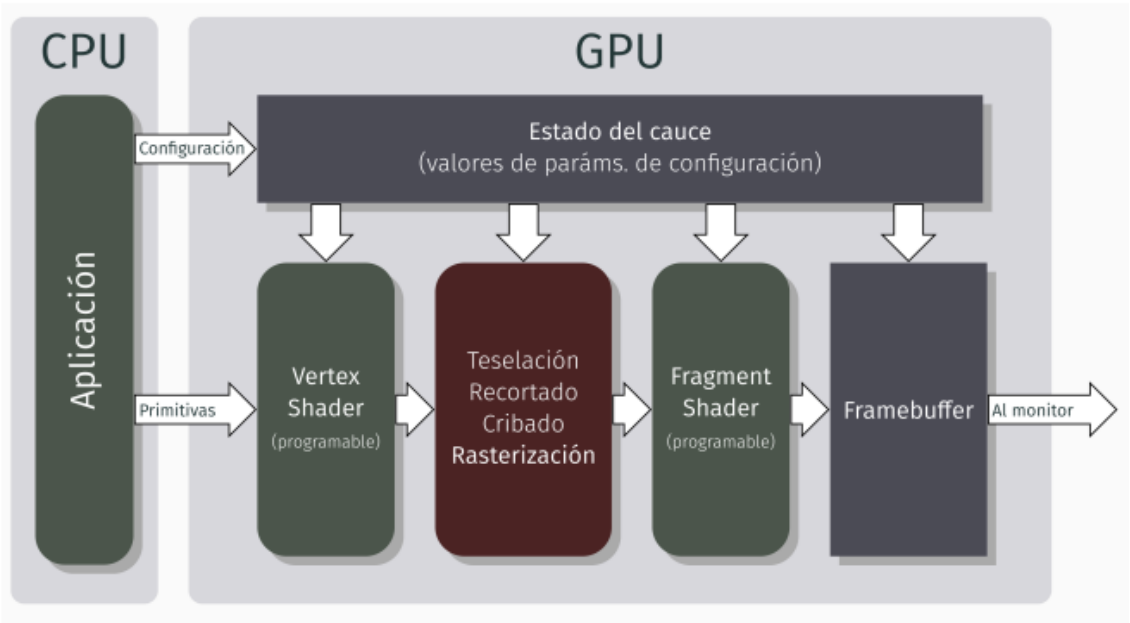

Figura 2.1: Esquema del cauce

2.3 APIs y motores gráficos

2.3.1 APIs para Rasterización, Ray-tracing y GPGPU

APIs de rasterización: conjunto de funciones para visualización 2D/3D, clases, interfaces. Son definidas sin ánimo de lucro, se puede dejar compilado en código intermedio.

APIs gráficas: estas proporcionan portabilidad y acceso simultáneo. La escritura en el framebuffer sigue siendo lenta, esto se soluciona usando GPU y enviando información de alto nivel a través del bus del sistema.

Como ejemplos podemos mencionar OpenGL, Vulkan, Metal, . . .

Las APIs modernas son más eficientes aunque tienen ciertas desventajas como más complejidad y menos portabilidad.

2.3.2 Motores gráficos

Definición 2.1 . Conjunto de herramientas software que facilita la creación de aplicaciones gráficas interactivas, como videojuegos, aunque se usa en la simulación, realidad virtual, ... Incluye motor de renderización y otras muchas herramientas. Permite crear aplicaciones gráficas portables y más. Algunos son de código abierto. Estos motores gráficos usan APIs gráficas como OpenGL, ...

El *grafo de escena* o *jerarquía* es una estructura de datos que modela las relaciones jerárquicas entre

los objetos.

El término *visual scripting* se define como la posibilidad de programar aspectos de la aplicación gráfica creando un grafo que codifica un **diagrama de flujo de datos**.

Los motores gráficos permiten programar partes usando lenguajes tradicionales como c++, entre otros.

Definición 2.2 . Un shader en un motor gráfico es un programa pequeño que se ejecuta en la GPU (tarjeta gráfica) y que define cómo deben procesarse y representarse los píxeles, vértices o fragmentos de una escena 3D en pantalla.

Se pueden programar los shaders mediante visual scripting y lenguajes de shading.

Principales motores gráficos: Unreal Engine, Unity, CryEngine y de fuentes abiertas: Godot, Armory3D.

Capítulo 3

El engine Godot. Mallas.

3.1 Introducción a Godot

Funcionalidad y características de Godot

Godot es un IDE (entorno integrado de desarrollo) que permite desarrollar aplicaciones gráficas interactivas en 2D y 3D, como videojuegos, simulaciones y visualizaciones. Además, facilita la ejecución, depuración y generación de archivos ejecutables independientes para diversas plataformas.

Entre sus características principales destacan:

- **Código abierto**: completamente gratuito y con una comunidad activa.
- **Multiplataforma**: el IDE puede ejecutarse en Linux, Windows, macOS e incluso en navegadores web (con ciertas limitaciones).
- **Generación de aplicaciones multiplataforma**: permite crear aplicaciones nativas para Windows, macOS, Linux, Android, iOS y Web.

Elementos de Godot

- **Editor**: herramienta tipo IDE que organiza recursos, diseña escenas, programa scripts y permite ejecutar y depurar aplicaciones.
- **Proyecto**: conjunto de escenas, nodos, scripts y recursos asociados a una aplicación.
- **Escenas**: estructuras jerárquicas (árboles de nodos) que representan los elementos de la aplicación.
- **Nodos**: componentes de las escenas, cada uno perteneciente a una clase específica de Godot.
- **Scripts**: código que define el comportamiento de nodos y escenas, utilizando GDScript (similar a Python), C#, o el lenguaje visual de Godot.
- **Recursos**: archivos como imágenes, audios, videos y modelos 3D utilizados en el desarrollo de la aplicación.

3.1.1 El lenguaje de programación GDScript

GDScript es un lenguaje de programación interpretado, de alto nivel y orientado a objetos.

- Tiene una sintaxis similar a python. Usando la indentación de manera similar.
- Las variables no tienen porque tener siempre asociado un tipo. [1]
- Es orientado a objetos.

[1] En las transparencias está mal. Página 7/84.

- Gestión automática de memoria mediante conteo de referencias.
- Especiamente diseñado para Godot.

```
extends Node3D # obligatorio: indica la clase base

class_name MiNodo # opcional: si no está es una clase anónima

var velocidad : float = 100.0 # variable de instancia (tipo opc
    .)

func v_cuadrado() -> float : # método que devuelve un float
 return velocidad * velocidad # devuelve la velocidad al
    cuadrado

func _init(): # constructor, puede tener parámetros
 pass # 'pass' indica que está vacío

func _ready(): # método de nodo listo
 print("Nodo listo") # imprime en la consola o terminal

func _process( delta: float ): # método de proceso por frame
 position.x += velocidad * delta # usa 'position' de Node2D
```

Extensión .gd, siempre definen una clase, aunque puede ser anónima, que siempre hereda de otra.

Tipos básicos y contenedores en GDScript

- **bool**: valores lógicos (true o false).
- **int**: enteros de 64 bits.
- **float**: números reales de doble precisión (64 bits).
- **String**: cadenas de caracteres en Unicode.
- **Vector2, Vector3, Vector4**: tuplas con 2, 3 o 4 elementos flotantes (32 bits).
- **Vector2i, Vector3i, Vector4i**: tuplas con 2, 3 o 4 elementos enteros.
- **Transform2D, Transform3D**: matrices de transformación en 2D o 3D.
- **Color**: colores en formato RGBA.
- **Array**: dinámico, puede contener elementos de cualquier tipo (Variant).
- **Array[T]**: homogéneo, todos los elementos son del tipo T.[2]
- **Arrays empaquetados**: homogéneos y contiguos en memoria, ideales para GPU:

[2]Debemos de tener cuidado al eliminar y añadir elementos, para garantizar la continuidad de los elementos se recomienda copiarlos completamente en otra array.

  - **PackedByteArray, PackedInt32Array, PackedInt64Array**: bytes o enteros.
  - **PackedFloat32Array, PackedFloat64Array**: flotantes de simple o doble precisión.
  - **PackedVector2Array, PackedVector3Array, PackedVector4Array**: vectores.
  - **PackedColorArray**: colores.

Permite tipos estáticos y dinámicos.

```
var x : float = 10.0 # tipo 'float' (explícito)
  # tipo explícito, no puede cambiar de tipo a lo largo de su
    vida

var y := 20 # tipo 'int' (es el tipo de la expresión '20').
  # implícito inferido, se infiere a partir de la expresión
    inicial

var z = 30.0 # sin tipo (inicialmente 'float', pero puede
    cambiar)
  # sin tipo, este puede cambiar a lo largo de su vida
```

Ventajas del Tipado Estático

El tipado estático ofrece múltiples beneficios:

- **Detección temprana de errores**: Identifica errores en tiempo de desarrollo, reduciendo el tiempo de depuración y evitando problemas en producción.
- **Mejor rendimiento**: Optimiza la ejecución al eliminar la necesidad de verificar tipos en tiempo de ejecución.
- **Legibilidad y comprensión**: Facilita la lectura del código al hacer explícitos los tipos de las variables.
- **Expresividad**: Los tipos implícitos permiten un código más conciso, aunque pueden afectar la legibilidad en ciertos casos.

3.1.2 Jerarquía de clases de Godot

Las clases de Godot están organizadas en una jerarquía de herencia, donde la clase Object es la raíz. Todas las demás clases heredan directa o indirectamente de ella. Algunas clases derivadas importantes son:

- Node: base para nodos de escenas, como visualización 2D/3D, controles, cámaras, luces, materiales, y más.
- Viewport: define una zona rectangular donde se renderiza una escena. Cada proyecto tiene un Viewport por defecto.
- MainLoop: clase abstracta para implementar el bucle principal de la aplicación. SceneTree es su implementación por defecto.

Otras clases derivadas incluyen:

- RefCounted: gestiona objetos en memoria dinámica con cuenta de referencias.
  - Resource: base para recursos de Godot, como:
    - Mesh: para mallas 2D/3D. Las mallas ocupan demasiado espacio por lo que duplicarlo es demasiado costoso.
    - Material: para definir la apariencia visual de objetos.
    - Image, Texture: para imágenes y texturas.
    - Shader: para programas que se ejecutan en la GPU.

Godot incluye diversos tipos de nodos organizados jerárquicamente. Entre los nodos principales destacan:

- **CanvasItem**: base para elementos visuales en 2D, con subclases como:
  - Control: interfaz de usuario (botones, menús, etc.).
  - Node2D: objetos 2D en escenas 2D.
- **Node3D**: base para objetos 3D, con subclases como:
  - VisualInstance3D: mallas y luces.
  - Camera3D: cámaras en escenas 3D.

En cuanto a mallas, la clase **Mesh** es la base para representar primitivas geométricas en 2D/3D. Sus subclases incluyen:

- ArrayMesh: definida por programador, permanece en GPU.
- ImmediateMesh: enviada a GPU en cada frame.
- PrimitiveMesh: primitivas predefinidas como BoxMesh, SphereMesh, etc.

Nodos específicos para 2D y 3D permiten instanciar mallas, cámaras y luces, como MeshInstance2D, Sprite2D, MeshInstance3D, y Light3D (DirectionalLight3D, OmniLight3D, SpotLight3D). Estos nodos facilitan la creación y manipulación de escenas gráficas.

Las mallas pueden tener referencias a muchas mallas o solo una referencia a un mesh (lo que está en la GPU).

Godot define varias clases para representar tuplas de valores, entre ellas tenemos Vector2, Vector3, Vector 4 (para float), con el sufijo i (para int), color para colores RGBA, Rect2 para rectángulos 2D.

La clase abstracta MainLoop define métodos que permite configurar el comportamiento de cualquier aplicación creada con Godot. Los métodos son **_initialize**, **_process** y **_finalize**. Cuando se ejecuta se dan estos pasos:

1. Se crea una instancia de la clase derivada de MainLoop (SceenTree), creando el árbol de inicio, dándole los valores por defecto que hay en el editor.
2. Se invoca al método **_initialize**
3. Mientras no se termine la aplicación:
   1. Se invoca a **_process** par actualizar el estado de los objetos de la aplicación.
   2. Si es necesario, se hace una espera hasta que sea el momento del siguiente frame.
4. Se invoca al método **_finalize**.

Clase SceneTree

- Implementación concreta de MainLoop.
- Creación del árbol.
- Gestión de nodos.
- Actualización de frames.

- Cálculos asociados.
- Entrada de usuario.
- Ejecución de scripts.
- Terminación y liberación de recursos.

Se puede diseñar el árbol de escena

- Antes de ejecutar. Para ello debe de crear un nodo y asignarle un script que redefine varios métodos de la calse Node. Por otro lado, se puede crear un nodo de clase definido por el usuario, que herede de Node. Esa clase pude tener métodos específicos.
- En tiempo de ejecución. Creando un nodo h con el método new de su clase y añadiéndolo al árbol como un hijo de un nodo p existente.

```
# Añadir nodos hijos a una malla
var h:= ArrayMesh.new()
p.add_child(h)
```

Redefinición de métodos

El comportamiento de un nodo puede adaptarse redefiniendo métodos de Node u Object que se invocan por SceneTree en determinados momentos durante la ejecución:

- **_init**: Constructor del nodo (método de Object), se invoca al crear un nodo para inicializar sus variables de instancia. Puede tener parámetros.
- **_enter_tree**: Se invoca al añadir un nodo al árbol, cuando su padre se ha añadido, pero antes de añadir sus hijos.
- **_ready**: Se invoca al añadir el nodo, después de **_enter_tree**, cuando ya se han añadido los nodos hijos.
- **_process(delta)**: Se invoca antes de cada frame para actualizar el estado del nodo. El parámetro delta indica el tiempo (en segundos) transcurrido desde el último frame.
- **_input(event)**: Invocado cuando se produce un evento de entrada (teclado, ratón, etc.). El parámetro event lleva información del evento.

3.2 Mallas en Godot

Cada primitiva o conjunto de primitivas se especifica mediante una secuencia ordenada de vértices. Un vértice es un espacio afín en 3D, puede tener asociado otros valores como colores, . . .

Existen 3 tipos de primitivas: puntos segmentos y triángulos.

A cada forma de codificar primitivas en una secuencia se le llama un tipo de primitivas, en GDScript se definen diversas constantes para eso, del tipo enumerado PrimitiveType, en la clase Mesh.

Tipo de primitiva	Descripción
Mesh.PRIMITIVE_POINTS	n puntos aislados (n arbitrario)

Tipo de primitiva	Descripción
Mesh.PRIMITIVE_LINES	n/2 segmentos independientes (n debe ser par)
Mesh.PRIMITIVE_LINE_STI	n - 1 segmentos formando una polilínea abierta (n debe ser mayor o igual a 2)
Mesh.PRIMITIVE_TRIANGI	n/3 triángulos (n debe ser múltiplo de 3), siempre debe de estar relleno de algo, una textura, . . .
Mesh.PRIMITIVE_TRIANGI	n - 2 triángulos compartiendo aristas (tira de triángulos), (n debe ser mayor o igual a 3)

Una secuenci a de coordenadas $(v_0, v_1, v_2, \ldots, v_{n-1})$ pueden formar puntos, segmentos o polilíneas abiertas.

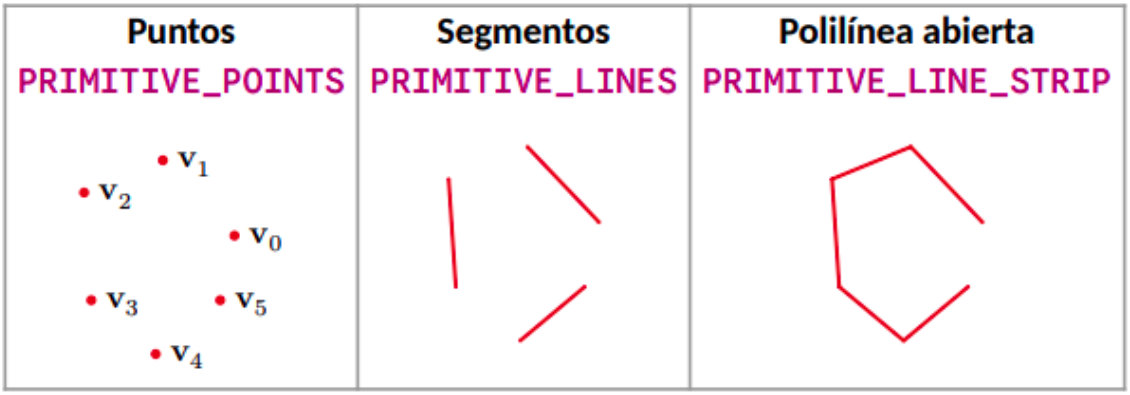

Figura 3.1: Con n=6

Cada primitiva de tipo triángulo tiene dos caras:

- Delantera: si sus vértices se visualizan en pantalla en el sentido contrario de las agujas del reloj.
- Trasera: sentido de las agujas del reloj.

Godot puede ser configurado para visualizar solo las delanteras, las traseras o ambas, esto se conoce como hacer *cribado de caras*. Por defecto, solo visualiza las delanteras.

En la figura podemos ver los puntos en las coordendas y un esquema de las aristas de los triángulos que se formarían.

En la figura es un ejemplo de cuando necesitamos repetir coordenadas, para ver 7 triángulos como es ese caso. Usando *GL_TRIANGLES*, necesitamos esta secuencia de vértices:

$$V_0, V_2, V_1, \quad V_1, V_2, V_3, \quad V_1, V_3, V_4, \quad V_2, V_5, V_3, \quad V_3, V_5, V_6, \quad V_3, V_6, V_7, \quad V_3, V_7, V_4$$

Supone emplear más memoria y tiempo para visualizar el escenario. Tiene 21 coordenadas de vértices, pero solo 8 son distintos.

Secuencias Indexadas

Figura 3.2: Esquema

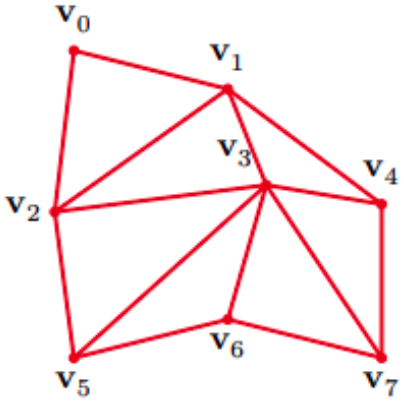

Figura 3.3: Necesidad de repetir coordenadas

Para solucionar el problema anterior relacionado con la memoria, las APIs y engines, permiten especificar una secuencia de vértices a partir de una secuencia de vértices únicos.

- Se parte de una secuencia V_n de n coordenadas arbitrarias de vértices $V_n = \{v_0, v_1, ..., v_{n-1}\}$.
- Se usa una secuencia I_m de m índices $I_m = \{i_0, i_1, ..., i_{m-1}\}$ donde cada valor i_j es un entero entre 0 y $n-1$ (ambos incluidos). Puede haber valores repetidos.
- La secuencia de vértices V_n y la de índices determinan otra secuencia S_m de m vértices:

  $$S_m = \{v_{i_0}, v_{i_1}, ..., v_{i_{m-1}}\}$$

  que tiene las mismas coordenadas de vértices de V_n pero en el orden especificado por los índices en I_m.

En el ejemplo de la figura usaríamos una lista de índices (cada tres forman un triángulo).

$$V_8 = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$

$$I_{21} = \{0, 2, 1, 1, 2, 3, 1, 3, 4, 2, 5, 3, 3, 5, 6, 3, 6, 7, 3, 7, 4\}$$

3.2.1 Atributos de vértices

Las coordenadas de su posición se considera una atributo de vértices, siendo este imprescindible. Además, hay otros como el color, la normal [3] y las coordenadas de textura [4].

Colores de vértices

Podemos añadir en los triángulos tres colores codificados con una terna RGB, de manera que se produciría una interpolación de los colores (para calcular cada pixel), o bien, una cuádrupla añadiendole transparencia.

[3] Vector unitario con tres coordenadas reales que determina la orientación de la superficie de un obeto en el punto donde está el vértice.

[4] Típicamente un par de valores reales, que se usan para determinar que punto de textura se fija al vértice.

Normales

A cada vértice se le puede asociar un vector de 3 componentes (x, y, z), su vector normal, que determina la orientación de la superficie en ese vértice para hacer el sombreado y la iuminación.

Coordenadas de texturas

Podemos asociar a cada vértice una dupla (s, t), que corresponden a sus coordenadas de textura, típicamente en $[0, 1]^2$.

Definición de valores de los atributos

En Godot, cada vértice tiene asociados varios atributos, como posición, color, normal y coordenadas de textura, entre otros como hemos visto. La utilización de estos atributos depende de la configuración del cauce gráfico. Por ejemplo:

- Si un objeto no tiene textura, las coordenadas de textura no se usarán.
- Si la iluminación está desactivada, las normales no serán utilizadas.

Es posible definir un único valor de un atributo para todos los vértices de una primitiva o especificar valores individuales para cada vértice. Durante la rasterización, los valores de los atributos en cada píxel se calculan mediante interpolación, permitiendo una representación visual más precisa y detallada.

3.2.2 Modos de envíos de datos a la GPU

Las GPUs modernas estan diseñadas para visualizar secuencias de vértices y atributos almacenados en la memoria de la GPU. Esto se debe que el acceso a su propia memoria es más rápido que a la del sistema. Pero, el origen de los datos estará en el sistema, es decir, siempre quedan inicialmente en la CPU. Es necesario realizar un envío de datos desde la CPU a la GPU para esto.

Para ello se puede realizar de dos formas:

- Modo inmediato: cada vez que queremos visualizar un frame, se envían atributos e índices a la GPU por el bus del sistema. Este modo es muy ineficiente en tiempo, ya que el ancho de banda del bus del sistema es limitado.
- Modo diferido: los datos de la secuencia de vértices se envían a la GPU una vez, como parte de la inicialización de la aplicación. Emplea menos tiempo por cuadro que el anterior, ya que la transferencia de datos se hace menos veces, suele ser una vez.

Por defecto, se suele usar el envío en modo diferido, debido a su mayor eficiencia. En algunos casos, es conveniente usar el de envío inmediato, para ello es aconsejable que se den estas dos condiciones:

- Secuencia de vértices y atributos es actualizada por la aplicación con mucha frecuencia.
- La secuencia es pequeña.

El envio previo a cada frame de los datos actualizables es realizable ya que no penaliza mucho el tiempo debido a que la secuencia de vértices no ocupa mucho espacio.

En mallas grandes se usa el envío diferido.

Godot usa normalmente el envío diferido, para los nodos que contienen mallas en 2D y 3D. Se puede usar el envío en modo inmediato usando nodos específicos para ello.

3.2.3 Representación de mallas en Godot

Las APIs y engines suelen ofrecer dos formas de almacenar arrays de posiciones de vértices y sus atributos: - Array de estructuras (AOS Array of Structures): usa un array o un vector donde cada entrada tiene las coordenadas de un vértice y todos sus atributos. - Estructura de arrays (SOA Structure of Arrays): usa una estructura con varios punteros a arrays de número de elementos. Uno de ellos contiene las coordenadsa y los otros contienen cada uno una tabla de atributos.

Los índices siempre están contiguos en su propio array. En Godot se usa la opción SOA.

En este lenguaje que venimos viendo se contemplan diversos tipos de datos para guardar tuplas con valores reales:

- Vector2: tupla de dos reales.
- Vector3: tupla de tres reales.
- Color: tupla de 4 reales.

Se suele asociar un valor entero a cada atributo de vértice. Se define en este lenguaje el tipo de enumerado ArrayType en la clase Mesh.

Identificador	**Valor**	**Significado**
Mesh.ARRAY_VERTEX	0	Coordenadas de posición (obligatorias)
Mesh.ARRAY_NORMAL	1	Vector normal
Mesh.ARRAY_TANGENT	2	Vector tangente
Mesh.ARRAY_COLOR	3	Color RGB
Mesh.ARRAY_TEX_UV	4	Coordenadas de textura
Mesh.ARRAY_TEX_UV2	5	Segundas coordenadas de textura
Mesh.ARRAY_CUSTOM0–3	6 – 9	Atributos definidos por el usuario

Cuadro 3.2: **Atributos de vértices en Godot**

Las tablas de atributos de vértices se pueden codificar en arrays empaquetados.

Atributo	**Clase array**	**Tipo elem.**
Posiciones 2D	PackedVector2Array	Vector2
Posiciones 3D	PackedVector3Array	Vector3
Colores	PackedColorArray	Color
Normales	PackedVector3Array	Vector3
Coords. textura	PackedVector2Array	Vector2

Cuadro 3.3: **Atributos de vértices y sus arrays empaquetados en Godot**

Clases de Mallas en Godot

- ArrayMesh: derivada de Mesh, contiene una malla que se visualizará en modo diferido.
- InmediateMesh: derivada de Mesh, contiene una malla que se visualizará en modo inmediato.
- SurfaceTool: derivada de RefCounted.
- MeshInstance2D: derivada de Node2D, permite instanciar una malla en 2D.

– MeshInstance3D: derivada de Node3D, permite instanciar una malla en 3D.

Ahora vamos a ver varios ejemplos con mallas 2D y 3D.

3.2.4 Mallas en 2D

Código 3.1: Malla sin indentación

```
extends MeshInstance2D

func _ready():
  # Declarar y dimensionar un array para las tablas
  var tablas: Array = []
  tablas.resize(Mesh.ARRAY_MAX)

  # Añadir la tabla con las coordenadas de posición de 6 v
   értices

  # Cambiar el color de la malla (atributo 'modulate' del
   nodo)
  modulate = Color(1.0, 0.5, 0.5)

  # Inicializar el atributo 'mesh' de este nodo
  mesh = ArrayMesh.new()  # Crea malla en modo diferido,
   vacía
  mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
   tablas)

  ## añadir la tabla con las coordenadas de posición de 6
   vértices:
  tablas[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([
    Vector2( 0.2, 0.1 ), Vector2( 0.7, 0.1 ), Vector2(
   0.1, 0.9 ),
    Vector2( 0.3, 1.0 ), Vector2( 1.0, 0.2 ), Vector2(
   1.0, 1.1 ),
  ])

  ## Otra posibilidad es haciendo push_back:

  ## añadir la tabla con las coordenadas de posición de 6
   vértices:
```

```
26  var posv := PackedVector2Array([]) # crear array posic.
     verts. vacío
27  posv.push_back(Vector2(0.2,0.1)); posv.push_back(Vector2
     (0.7,0.1))
28  posv.push_back(Vector2(0.1,0.9)); posv.push_back(Vector2
     (0.3,1.0))
29  posv.push_back(Vector2(1.0,0.2)); posv.push_back(Vector2
     (1.0,1.1))
30  tablas[ Mesh.ARRAY_VERTEX ] = posv # asignar la tabla de
      posiciones
```

Código 3.2: Malla con colores de vértices

```
1  extends MeshInstance2D
2  func _ready():
3      ## declarar y dimensionar un array para las tablas
4      var tablas : Array = []
5      tablas.resize( Mesh.ARRAY_MAX )
6
7      ## añadir las tablas con posiciones y colores de 6 vé
     rtices
8      ## inicializar el atributo 'mesh' de este nodo
9      mesh = ArrayMesh.new() ## crea malla en modo diferido,
      vacía
10     mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES
     , tablas )
11
12     ## añadir las tablas con posición y colores de 6 vé
     rtices
13     tablas[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([ ...
      ]) # igual
14     tablas[ Mesh.ARRAY_COLOR ] = PackedColorArray([
15         Color( 1.0,0.0,0.0 ), Color( 0.0,1.0,0.0 ), Color(
     0.0,0.0,1.0 ),
16         Color( 1.0,1.0,0.0 ), Color( 0.0,1.0,1.0 ), Color(
     1.0,0.0,1.0 ),
17     ])
18
19     ## Otra posibilidad es haciendo push_back:
20
21     var posv := PackedVector2Array([]) # crear array posic
```

```
    . verts. vacío
22    var colv := PackedColorArray([]) # crear array colores
      vértices vacío
23    posv.push_back(Vector2(0.2,0.1)); colv.push_back(Color
    (1.0,0.0,0.0))
24    # .... posiciones y colores del resto de vértices ...
25    tablas[ Mesh.ARRAY_COLOR ] = colv # asignar la tabla
    de colores
26    tablas[ Mesh.ARRAY_VERTEX ] = posv # asignar la tabla
    de posiciones
```

Código 3.3: Carga de texturas en mallas 2D

```
1  func CargarTextura( arch : String ) -> ImageTexture :
2      ## crear un objeto 'Image' con la imgen
3      var imagen := Image.new()
4      assert( imagen.load(arch) == OK, "Error cargando '"+
     arch+"'." )
5
6      ## crear un objeto 'ImageTexture' a partir del objeto
     'Image'
7      var textura := ImageTexture.create_from_image( imagen
     )
8      print("Textura cargada desde archivo: '",arch,"'.")
9
10     ## devolver la textura
11     return textura
```

Código 3.4: Envío inmediato de mallas 2D

```
1  extends MeshInstance2D
2
3  var t : float = 0 # tiempo total desde inicio en segundos.
4
5  func _ready():
6      mesh = ImmediateMesh.new() # crear 'Mesh' vacío al
     inicio.
7
8  func _process(delta: float):
```

```
    # actualiza tiempo transcurrido
    t += delta

    # calcular un vector 'd' que va cambiando con el tiempo
    var d := 0.2 * Vector2(cos(4.0 * t), sin(4.0 * t))

    # volver a definir las tablas
    mesh.clear_surfaces() # limpia el mesh
    mesh.surface_begin(Mesh.PRIMITIVE_TRIANGLES) # dar tipo de primitiva

    # definir vértice 0
    mesh.surface_set_color(Color(1.0, 0.0, 0.0))
    mesh.surface_add_vertex_2d(Vector2(0.2, 0.2) + d) # usa 'd' animado

    # definir vértice 1
    mesh.surface_set_color(Color(0.0, 1.0, 0.0)) # opcional: si no conserva
    mesh.surface_add_vertex_2d(Vector2(1.0, 0.5))

    # definir vértice 2
    mesh.surface_set_color(Color(0.0, 0.0, 1.0)) # opcional: si no conserva
    mesh.surface_add_vertex_2d(Vector2(0.5, 1.0))

    mesh.surface_end() # fin de los vértices
```

Código 3.5: Malla 2D con texturas

```
extends MeshInstance2D
func _ready():
    ## definir tablas para 4 vértices formando 2 triángulos:
    var tablas : Array = [] ; tablas.resize( Mesh.ARRAY_MAX )
    tablas[ Mesh.ARRAY_VERTEX ] = PackedVector2Array([ ... ]) # igual
    tablas[ Mesh.ARRAY_TEX_UV ] = PackedVector2Array([
    Vector2( 0.0,0.0 ), Vector2( 1.0,0.0 ),
```

```
    Vector2( 0.0,1.0 ), Vector2( 1.0,1.0 )
    ])
    tablas[ Mesh.ARRAY_INDEX ] = PackedInt32Array([ 0,1,2,
    2,1,3 ])
    ## crear el objeto 'mesh'
    mesh = ArrayMesh.new()
    mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES
    , tablas )
    ## cargar la textura y asignarla a este objeto
    texture = CargarTextura( "imgs/madera2.jpg" )
```

3.2.5 Mallas en 3D

Código 3.6: Malla indexada en 3D

```
extends MeshInstance2D
func _ready():
    ## declarar y dimensionar un array para las tablas (
    igual que en 2D)
    var tablas : Array = []
    tablas.resize( Mesh.ARRAY_MAX )

    ## añadir las tablas posiciones, colores e índices de
    un triángulo
    ## .... (ver siguiente transparencia) ....
    ## inicializar el atributo 'mesh' de este nodo (igual
    que en 2D)
    mesh = ArrayMesh.new() ## crea malla en modo diferido,
    vacía
    mesh.add_surface_from_arrays( Mesh.PRIMITIVE_TRIANGLES
    , tablas )
    ## asignarle un material sin iluminación ni sombreado
    ## ...(ver siguientes transparencias) ....
```

Código 3.7: Material con colores de vértices en 3D

```
# asignarle un material sin iluminación ni sombreado
var mat := StandardMaterial3D.new() # crear objeto '
```

```
    StandardMaterial3D'
## configurar el objeto con el material ('mat')
mat.vertex_color_use_as_albedo = true # usar colores de vé
    rtices
mat.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADED #
    sin sombr.
mat.cull_mode = BaseMaterial3D.CULL_DISABLED # sin cribado
mat.lighting = false # no usar la luces
## redefinir atributo 'material_override' del nodo '
    MeshInstance3D'
material_override = mat
```

Código 3.8: Material con color de plano en 3D

```
# asignarle un material sin iluminación ni sombreado
var mat := StandardMaterial3D.new() # crear objeto '
    StandardMaterial3D'
## configurar el objeto con el material ('mat')
mat.vertex_color_use_as_albedo = false # no usar colores
    de vértices
mat.albedo_color = Color( 1.0, 0.4, 0.4 ) # color plano de
     la malla
mat.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADED #
    sin sombr.
mat.cull_mode = BaseMaterial3D.CULL_DISABLED # sin cribado
mat.lighting = false # no usar la luces
## redefinir atributo 'material_override' del nodo '
    MeshInstance3D'
material_override = mat
```

Código 3.9: Tablas en mallas 3D

```
## añadir las tablas posiciones, colores e índices de un
    triángulo
tablas[ Mesh.ARRAY_VERTEX ] = PackedVector3Array([
     Vector3(0.6,0.6,0.1), Vector3(0.1,0.8,0.1 ), Vector3
    (0.2,0.1,0.1)
])
tablas[ Mesh.ARRAY_COLOR ] = PackedColorArray([
```

```
    Color( 1, 0, 0 ), Color( 0, 1, 0 ), Color( 0, 0, 1 )
])
tablas[ Mesh.ARRAY_INDEX] = PackedInt32Array([
    0, 1, 2
])

## también se puede hacer con push back
```

Parte II

Prácticas

Capítulo 4

Introducción

Las prácticas de la asignatura se llevaran acabo con *Godot*.

Un motor de juego es una herramienta compleja y difícil de presentar en pocas palabras. Aquí hay una rápida sinopsis, que eres libre de reutilizar si necesitas una breve reseña sobre Godot Engine:

Godot Engine es un motor de videojuegos repleto de características, multiplataforma para crear juegos 2D y 3D por medio de una interfaz unificada. Provee un conjunto exhaustivo de herramientas comunes, para que los usuarios puedan enfocarse en crear juegos sin tener que reinventar la rueda. Los juegos pueden exportarse en un sólo clic a numerosas plataformas, incluyendo las principales plataformas de escritorio (Linux, macOS, Windows), plataformas móviles (Android, iOS), así como plataformas y consolas basadas en la web.

Godot es completamente gratis y de código abierto bajo la permisiva licencia MIT (Licencia del Instituto Tecnológico de Massachusetts). Sin condiciones, sin regalías, nada. Los juegos de los usuarios son suyos, hasta la última línea de código del motor. El desarrollo de Godot es totalmente independiente y dirigido por la comunidad, lo que permite a los usuarios ayudar a dar forma a su motor para que coincida con sus expectativas. Está respaldado por la Godot Foundation (Fundación Godot) sin fines de lucro.[1]

Para la *descarga* de Godot debemos de hacerlo mediante el enlace de la web oficial [2].

Nota

El equipo de Godot no puede proporcionar una exportación de consola de código abierto debido a los términos de licencia impuestos por los fabricantes de consolas. Independientemente del motor que use, lanzar juegos en consolas siempre es mucho trabajo. Puedes leer más sobre eso en el Soporte de consolas en Godot.

Se recomienda encarecidamente leer el *started* de Godot [3].

4.1 Aprendiendo a programar con Godot

En Godot, es posible escribir código utilizando los lenguajes de programación GDScript y C#. Para aprender los lenguajes relacionados, podemos realizar el curso de manera interactiva que nos ofrecen.[4]

[1] https://docs.godotengine.org/es/4.x/
[2] https://godotengine.org/es/
[3] https://docs.godotengine.org/es/4.x/getting_started
[4] https://gdquest.github.io/learn-gdscript

Ejemplo de GDScript comparando con otros lenguajes que ya conocemos

```
# GDScript
func take_damage(amount):
    health -= amount
    if health < 0:
        die()
```

```
# Python
def take_damage(amount):
    health -= amount
    if health < 0:
        die()
```

```
# Javascript
function takeDamage(amount) {
    health -= amount;
    if (health < 0) {
        die();
    }
}
```

Además, en el curso se nos ofrece la posibilidad de ir practicando como vemos en la figura 4.1.

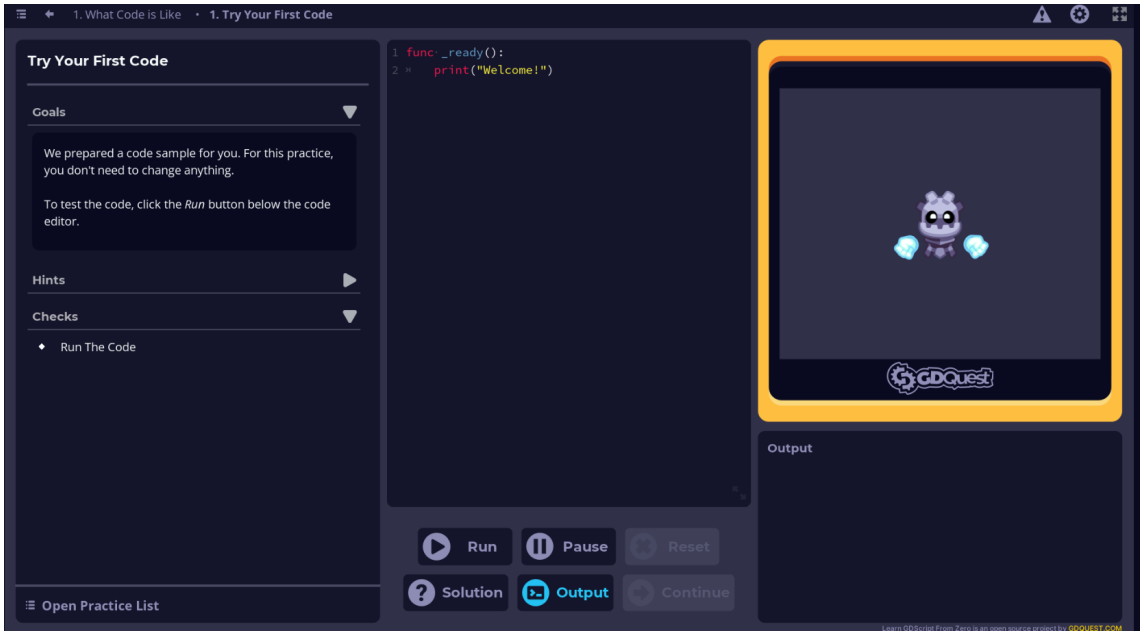

Figura 4.1: Imagen de práctica

4.2 Conceptos clave de Godot

Todos los motores de videojuegos giran alrededor de abstracciones que usas para crear tus aplicaciones. En Godot, un juego es un árbol de nodos agrupados en escenas. Los nodos pueden comunicarse entre sí mediante señales.

- **Escenas:** En Godot, puedes dividir tu juego en escenas reutilizables, como personajes, niveles o menús. Estas escenas pueden combinarse y anidarse para formar estructuras más complejas.
- **Nodos:** Las escenas están compuestas por nodos, que son los bloques de construcción básicos de tu juego. Godot ofrece una amplia biblioteca de nodos base que puedes combinar y extender.
- **El árbol de escenas:** El árbol de escenas organiza todas las escenas y nodos de tu juego. Representa la jerarquía y estructura general de tu proyecto.
- **Señales:** Los nodos emiten señales para comunicar eventos, como colisiones o interacciones. Esto permite una comunicación flexible entre nodos sin acoplamiento directo.
- **Sumario:** Los nodos, escenas, el árbol de escenas y las señales son los conceptos fundamentales de Godot. Estos elementos te permitirán estructurar y desarrollar tus juegos de manera eficiente.

Capítulo 5

Práctica 1

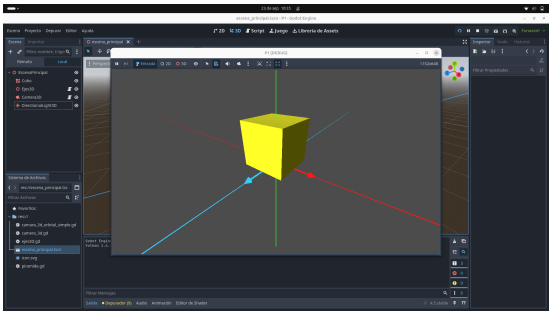

Figura 5.1: Diferentes tonalidades

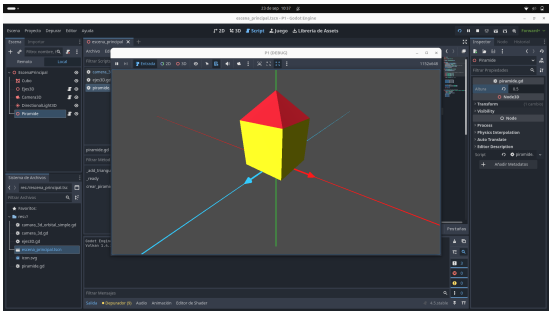

Figura 5.2: Pirámide

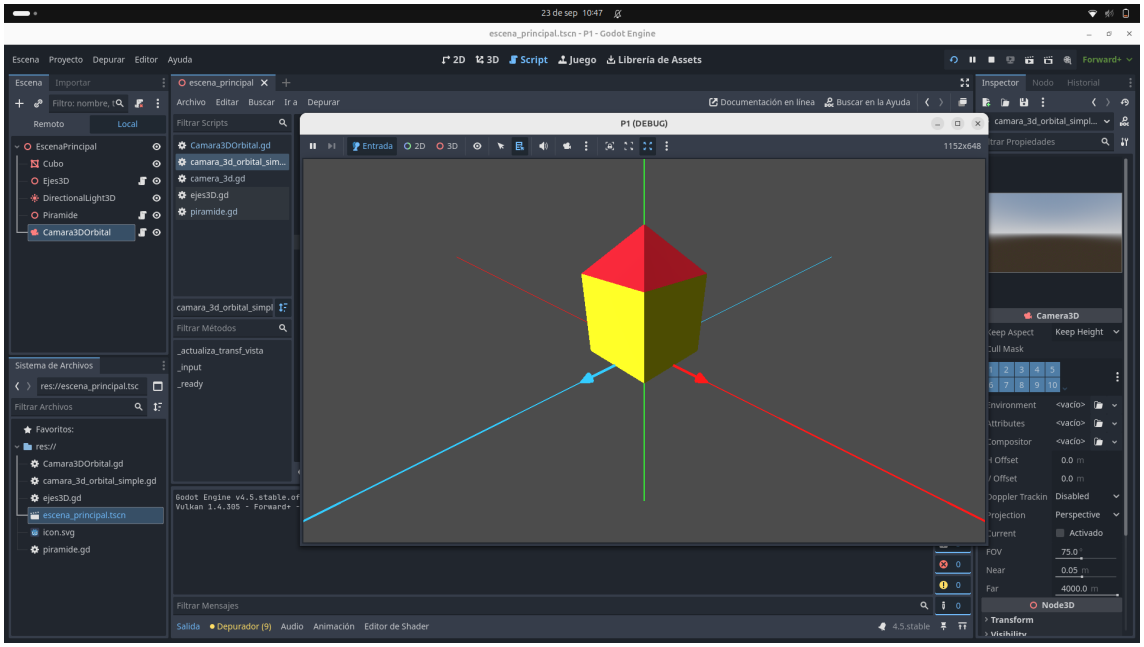

Figura 5.3: Resultado final

Bibliografía

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

[2] Universidad de Granada, *Diapositivas de la asignatura*, Curso 2025/2026.