Universidad de las Fuerzas Armadas "ESPE"

Departamento de ciencias de la Computación

Ingeniería en Desarrollo de Software

Proyecto Unidad 2: Motor Gráfico 3D

Docente: Darío Javier Morales Caiza

Estudiantes:

Diana Carolina Guerra Coronel

Dylan Alexander Alvarado Palacios

Eduardo Antonio Mortensen Franco

Materia: Computación Gráfica

NRC: 28467

Fecha: 17/12/2025

ÍNDICE

MOTOR GRÁFICO 3D

1. INTRODUCCIÓN

1.1. Contexto del proyecto

Los gráficos tridimensionales son fundamentales en aplicaciones modernas de visualización, desde videojuegos hasta software de diseño arquitectónico. Este proyecto implementa un **motor gráfico 3D completo** desarrollado en C# que permite la creación, transformación, proyección y renderizado de objetos tridimensionales en tiempo real.

1.2. Objetivos del sistema

El motor 3D desarrollado tiene como objetivos principales:

1. **Implementar el pipeline gráfico 3D completo**: Desde la definición de geometría hasta la rasterización en pantalla
2. **Proporcionar transformaciones geométricas**: Traslación, rotación y escalamiento mediante álgebra lineal
3. **Ofrecer sistema de cámara flexible**: Tres modos de operación (orbital, libre y fija) para diferentes casos de uso
4. **Generar figuras primitivas**: Fábrica de geometría predefinida (cubo, esfera, cilindro, cono, pirámide, toroide)
5. **Calcular iluminación**: Modelo Phong simplificado para renderizado realista

1.3. Alcance y características

El sistema implementa:

- **Estructuras matemáticas fundamentales**: Vector3D y Matrix4x4 con operaciones de álgebra lineal
- **Transformaciones 3D**: Matrices de traslación, rotación (X, Y, Z) y escalamiento
- **Proyección perspectiva**: Simulación de profundidad y convergencia
- **Sistema de cámara**: Posicionamiento mediante matriz LookAt con tres modos de control
- **Iluminación básica**: Componentes ambiental y difusa (modelo Phong simplificado)
- **Generación de geometría**: 6 primitivas 3D paramétricas con control de detalle
- **Renderizado en tiempo real**: Compatible con GDI+ de C#

1.4. Aplicaciones prácticas

Este motor gráfico puede utilizarse para:

- **Visualizadores 3D**: Inspección de modelos y productos
- **Herramientas CAD básicas**: Diseño asistido por computadora
- **Simuladores educativos**: Enseñanza de geometría y álgebra lineal
- **Prototipos de juegos**: Desarrollo de mecánicas 3D básicas
- **Renders arquitectónicos**: Visualización de espacios y estructuras

2. MARCO TEÓRICO

2.1. Pipeline de Gráficos 3D

El pipeline gráfico 3D es el proceso que transforma objetos tridimensionales en imágenes bidimensionales en pantalla. Consta de las siguientes etapas secuenciales:

Espacio objeto (local): La geometría se define en su sistema de coordenadas local, donde el objeto está centrado en su propio origen.

Espacio mundo (global): Se aplican transformaciones de modelo (escala, rotación, traslación) para posicionar el objeto en la escena global.

Espacio vista (cámara): Se transforma la escena según la posición y orientación de la cámara.

Espacio clip: Se aplica la proyección perspectiva, convirtiendo coordenadas 3D en coordenadas homogéneas.

Espacio pantalla: Se convierten las coordenadas normalizadas a píxeles de la ventana de visualización.

2.2. Álgebra Lineal para Gráficos 3D

2.2.1. Vectores 3D

Un vector 3D representa una dirección y magnitud en el espacio tridimensional. Se denota como **v** = (x, y, z).

Operaciones fundamentales:

Suma de vectores:

$$v_1 + v_2 = (x_1 + x_2, y_1 + y_2, z_1 + z_2)$$

Resta de vectores:

$$v_1 - v_2 = (x_1 - x_2, y_1 - y_2, z_1 - z_2)$$

Multiplicación por escalar:

$$k \cdot v = (k \cdot x, k \cdot y, k \cdot z)$$

Magnitud (longitud del vector):

$$|v| = \sqrt{(x^2 + y^2 + z^2)}$$

Normalización (vector unitario):

$$\hat{v} = v / |v| = (x/|v|, y/|v|, z/|v|)$$

Producto punto (dot product):

$$v_1 \cdot v_2 = x_1x_2 + y_1y_2 + z_1z_2$$

$$v_1 \cdot v_2 = |v_1| \, |v_2| \cos(\theta)$$

Donde θ es el ángulo entre los vectores. Se utiliza para calcular ángulos, proyecciones e iluminación.

Producto cruz (cross product):

$$v_1 \times v_2 = (y_1z_2 - z_1y_2, z_1x_2 - x_1z_2, x_1y_2 - y_1x_2)$$

El resultado es un vector perpendicular a ambos vectores de entrada. Se utiliza para calcular normales de superficies.

2.2.2. Matrices 4×4 y Coordenadas Homogéneas

Las transformaciones 3D se representan mediante matrices 4×4 usando coordenadas homogéneas (x, y, z, w). Este sistema permite representar traslaciones mediante multiplicación matricial, lo cual es imposible con matrices 3×3.

Matriz identidad:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Multiplicación de matrices: La composición de transformaciones se realiza multiplicando matrices. La aplicación es de derecha a izquierda:

M_final = M_traslación × M_rotación × M_escala

Transformación de punto:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{bmatrix} w' \end{bmatrix} \begin{bmatrix} m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} w \end{bmatrix}$$

Después de la multiplicación, se normaliza dividiendo por w':

$$x\_final = x' / w'$$

$$y\_final = y' / w'$$

$$z\_final = z' / w'$$

2.3. Transformaciones Geométricas

2.3.1. Traslación

Desplaza un objeto en el espacio sin cambiar su orientación ni tamaño.

Matriz de traslación:

$$\begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde (tx, ty, tz) es el vector de desplazamiento.

2.3.2. Escalamiento

Modifica el tamaño del objeto en cada eje de forma independiente.

Matriz de escalamiento:

$$\begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Si sx, sy, sz > 1: Aumenta el tamaño
- Si sx, sy, sz < 1: Reduce el tamaño
- Si sx = sy = sz: Escalamiento uniforme (mantiene proporciones)

2.3.3. Rotación en eje X

Rota el objeto alrededor del eje X (pitch o cabeceo).

Matriz de rotación X:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde α es el ángulo de rotación.

2.3.4. Rotación en eje Y

Rota el objeto alrededor del eje Y (yaw o guiñada).

Matriz de rotación Y:

$$\begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde β es el ángulo de rotación.

2.3.5. Rotación en eje Z

Rota el objeto alrededor del eje Z (roll o balanceo).

Matriz de rotación Z:

$$\begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde γ es el ángulo de rotación.

2.3.6. Ángulos de Euler

Para orientar un objeto en el espacio se utilizan tres rotaciones consecutivas (Ángulos de Euler). El orden de aplicación es crítico ya que las rotaciones no son conmutativas. En este sistema se usa el orden YXZ (Yaw-Pitch-Roll).

2.4. Proyección Perspectiva

La proyección perspectiva simula la forma en que el ojo humano percibe la profundidad: los objetos lejanos aparecen más pequeños y las líneas paralelas convergen en un punto de fuga.

Parámetros de proyección:

- **FOV** (Field of View): Ángulo de campo visual en grados (típicamente 45-90°)
- **Aspect ratio**: Relación ancho/alto de la pantalla (w/h)
- **Near plane** (n): Distancia al plano de recorte cercano
- **Far plane** (f): Distancia al plano de recorte lejano

Matriz de proyección perspectiva:

$$\begin{bmatrix} 1/(a \cdot \tan(FOV/2)) & 0 & 0 & 0 \\ 0 & 1/\tan(FOV/2) & 0 & 0 \\ 0 & 0 & -(f+n)/(f-n) & -2fn/(f-n) \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Donde:

- a = aspect ratio
- FOV debe convertirse a radianes: $FOV\_rad = FOV \times \pi/180$

Propiedades de la proyección perspectiva:

- Objetos lejanos se reducen proporcionalmente a su distancia
- Líneas paralelas en 3D convergen en puntos de fuga en 2D
- Preserva la sensación de profundidad y realismo

2.5. Sistema de Cámara

2.5.1. Matriz LookAt

La matriz LookAt posiciona y orienta la cámara en el espacio especificando tres vectores:

- **Eye** (E): Posición de la cámara
- **Target** (T): Punto hacia donde mira la cámara
- **Up** (U): Vector que indica qué dirección es "arriba"

Construcción de los ejes de la cámara:

Eje Z (dirección de vista):

$$Z = (E - T) / |E - T|$$

Este eje apunta desde el objetivo hacia la cámara (vista hacia atrás).

Eje X (dirección derecha):

$$X = (U \times Z) / |U \times Z|$$

Eje Y (dirección arriba real):

$$Y = Z \times X$$

Matriz LookAt resultante:

$$\begin{bmatrix} X.x & X.y & X.z & -(X \cdot E) \\ Y.x & Y.y & Y.z & -(Y \cdot E) \\ Z.x & Z.y & Z.z & -(Z \cdot E) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde X·E, Y·E, Z·E son productos punto.

2.5.2. Coordenadas Esféricas (Sistema Orbital)

Para implementar una cámara orbital que gira alrededor de un punto objetivo, se utilizan coordenadas esféricas:

Parámetros:

- **r**: Distancia radial (radio de órbita)
- **θ** (theta): Ángulo azimutal, rotación horizontal (0° a 360°)
- **φ** (phi): Ángulo polar, rotación vertical (1° a 179°)

Conversión de coordenadas esféricas a cartesianas:

$$x = r \cdot \sin(\varphi) \cdot \cos(\theta)$$

$$y = r \cdot \cos(\varphi)$$

$$z = r \cdot \sin(\varphi) \cdot \sin(\theta)$$

La posición final de la cámara es:

$$P\_cámara = P\_objetivo + (x, y, z)$$

Gimbal Lock: Es una limitación que ocurre cuando φ = 0° o φ = 180° (en los polos). Se previene restringiendo φ al rango (1°, 179°).

2.6. Modelo de Iluminación

2.6.1. Modelo Phong Simplificado

El modelo de iluminación Phong calcula la intensidad de luz en una superficie considerando múltiples componentes. En su versión simplificada utiliza:

Componente ambiental (Ia):

$$Ia = ka \cdot Iamb$$

Donde:

- ka: Coeficiente de reflexión ambiental (típicamente 0.2)
- Iamb: Intensidad de luz ambiental

Representa la luz que ilumina uniformemente todas las superficies, simulando luz reflejada indirectamente.

Componente difusa (Id) - Ley de Lambert:

$$Id = kd \cdot Il \cdot \max(0, N \cdot L)$$

Donde:

- kd: Coeficiente de reflexión difusa
- Il: Intensidad de la fuente de luz
- N: Vector normal de la superficie (unitario)
- L: Vector dirección hacia la fuente de luz (unitario)
- N · L: Producto punto entre la normal y la dirección de luz

Intensidad total:

$$I = \min(1.0, Ia + Id)$$

Color final de la superficie:

$$Color\_final = Color\_base \times I$$

Interpretación física del producto punto N·L:

- N · L = 1: Superficie perpendicular a la luz (máxima iluminación)
- N · L = 0: Superficie paralela a la luz (solo luz ambiental)
- N · L < 0: Superficie opuesta a la luz (sombra, se clampea a 0)

2.7. Geometría Paramétrica

2.7.1. Esfera mediante Coordenadas Esféricas

Una esfera de radio r se parametriza usando dos ángulos:

Parametrización:

$\theta \in [0, \pi]$ Ángulo polar (latitud, de polo norte a sur)

$\varphi \in [0, 2\pi]$ Ángulo azimutal (longitud, meridiano completo)

$$x = r \cdot \sin(\theta) \cdot \cos(\varphi)$$

$$y = r \cdot \cos(\theta)$$

$$z = r \cdot \sin(\theta) \cdot \sin(\varphi)$$

La generación de la malla se realiza discretizando estos ángulos en "stacks" (anillos horizontales de latitud) y "slices" (meridianos verticales de longitud).

2.7.2. Cilindro

Un cilindro de radio r y altura h se genera mediante:

Círculos base (superior e inferior):

$$x = r \cdot \cos(\varphi)$$

$$z = r \cdot \sin(\varphi)$$

$$y = \pm h/2$$

Donde $\varphi \in [0, 2\pi]$

Las caras laterales conectan los puntos correspondientes de ambos círculos formando cuadriláteros.

2.7.3. Cono

Un cono de radio r (base) y altura h se define por:

Ápice: (0, h/2, 0)

Base circular:

$$x = r \cdot \cos(\varphi)$$

$$z = r \cdot \sin(\varphi)$$

$$y = -h/2$$

$$\text{Donde } \varphi \in [0, 2\pi]$$

Las caras laterales son triángulos que conectan el ápice con cada par de puntos consecutivos de la base.

2.7.4. Toroide (Dona)

Un toroide se define por dos radios y se parametriza con dos ángulos:

Parámetros:

- R: Radio mayor (distancia del centro del toro al centro del tubo)
- r: Radio menor (radio del tubo)

Parametrización:

$$\theta \in [0, 2\pi] \quad \text{Ángulo alrededor del círculo mayor}$$

$$\varphi \in [0, 2\pi] \quad \text{Ángulo alrededor del tubo}$$

$$x = (R + r \cdot \cos(\varphi)) \cdot \cos(\theta)$$

$$y = r \cdot \sin(\varphi)$$

$$z = (R + r \cdot \cos(\varphi)) \cdot \sin(\theta)$$

Topología: El toroide es una superficie cerrada de género 1 (tiene un agujero), a diferencia de la esfera que es de género 0.

2.8. Cálculo de Normales

2.8.1. Normal de una Superficie Plana

Para una cara definida por tres vértices P_0, P_1, P_2, la normal se calcula mediante el producto cruz:

Vectores en la superficie:

$$v_1 = P_1 - P_0$$

$$v_2 = P_2 - P_0$$

Normal (no normalizada):

$$N = v_1 \times v_2$$

Normal unitaria:

$$\hat{N} = N / |N|$$

2.8.2. Regla de la Mano Derecha

El orden de los vértices determina la dirección de la normal:

- Orden antihorario (visto desde fuera): Normal apunta hacia el exterior
- Orden horario: Normal apunta hacia el interior

Este orden es crítico para:

- Iluminación correcta
- Back-face culling (eliminación de caras traseras)
- Consistencia visual

2.9. Back-Face Culling

Es una técnica de optimización que elimina caras no visibles para el observador.

Criterio de visibilidad:

$$\text{Visible si: } N \cdot V > 0$$

Donde:

- N: Normal de la cara (unitaria)

- V: Vector desde la cara hacia la cámara (unitario)

Si $N \cdot V \leq 0$, la cara está orientada hacia el lado opuesto y no es visible.

Ventaja: Reduce aproximadamente el 50% de las caras a renderizar en objetos cerrados convexos.

2.10. Sistemas de Coordenadas

2.10.1. Sistema Right-Handed (Mano Derecha)

Este sistema es el estándar en matemáticas y se usa en este motor:

- **Eje X**: Apunta hacia la derecha
- **Eje Y**: Apunta hacia arriba
- **Eje Z**: Apunta hacia el observador (fuera de la pantalla)

Relación entre ejes:

$$X \times Y = Z$$

$$Y \times Z = X$$

$$Z \times X = Y$$

2.10.2. Conversión a Coordenadas de Pantalla

Las coordenadas normalizadas (NDC, Normalized Device Coordinates) en el rango [-1, 1] se convierten a píxeles:

Transformación viewport:

$$x_{pantalla} = (x_{ndc} + 1) \cdot ancho / 2$$

$$y_{pantalla} = (1 - y_{ndc}) \cdot alto / 2$$

La inversión de y es necesaria porque en sistemas gráficos 2D el origen está en la esquina superior izquierda con Y creciendo hacia abajo.

2.11. Fundamentos Matemáticos Adicionales

2.11.1. Propiedades del Producto Punto

- **Conmutativo**: $a \cdot b = b \cdot a$
- **Distributivo**: $a \cdot (b + c) = a \cdot b + a \cdot c$
- **Relación con ángulo**: $a \cdot b = |a| |b| \cos(\theta)$
- **Proyección**: La proyección de a sobre b es: $proj_b(a) = (a \cdot \hat{b}) \hat{b}$

2.11.2. Propiedades del Producto Cruz

- **Anticonmutativo**: $a \times b = -(b \times a)$
- **No asociativo**: $a \times (b \times c) \neq (a \times b) \times c$
- **Perpendicularidad**: $(a \times b) \cdot a = 0$ y $(a \times b) \cdot b = 0$
- **Magnitud**: $|a \times b| = |a| |b| \sin(\theta)$
- **Área de paralelogramo**: $|a \times b|$ es el área del paralelogramo formado por a y b

2.11.3. Transformaciones Lineales

Una transformación lineal T cumple:

$$T(u + v) = T(u) + T(v)$$

$$T(k \cdot v) = k \cdot T(v)$$

3. IMPLEMENTACIÓN:

3.1. Figuras3DFactory

3.1.1. Descripción técnica general

Figuras3DFactory es una clase estática que genera figuras 3D predefinidas. Actúa como fábrica de primitivas geométricas, retornando objetos Figura3D completamente configurados con vértices, caras y propiedades visuales.

Propósito: Simplificar la creación de geometría 3D común sin calcular manualmente coordenadas y conectividad.

3.1.2. Figuras implementadas

3.1.2.1. CrearCubo

public static Figura3D CrearCubo(float tamaño = 1.0f)

Descripción: Genera un cubo regular centrado en el origen.

Parámetros:

- tamaño: Longitud de arista (default: 1.0)

Geometría:

- **8 vértices**: Esquinas del cubo en ±(t, t, t) donde t = tamaño/2
- **6 caras**: Cuadriláteros, una por cada lado

Caras:

[0,1,2,3] Frente | [5,4,7,6] Atrás | [4,0,3,7] Izquierda

[1,5,6,2] Derecha | [3,2,6,7] Arriba | [4,5,1,0] Abajo

Propiedades: Nombre="Cubo", Color=Azul

Uso típico: Cajas, edificios, volúmenes de colisión

3.1.2.2. CrearEsfera

public static Figura3D CrearEsfera(float radio = 1.0f, int subdivisiones = 2)

Descripción: Genera una esfera por anillos de latitud y meridianos.

Parámetros:

- radio: Radio de la esfera (default: 1.0)
- subdivisiones: Nivel de detalle (default: 2)
    - Mayor valor = más polígonos = más suave

Algoritmo:

stacks = max(6, subdivisiones × 8) // Anillos horizontales

slices = stacks × 2 // Meridianos verticales

1. Polo norte: (0, radio, 0)
2. Anillos intermedios:

   theta = π × i / stacks

   y = radio × cos(theta)

   r = radio × sin(theta)

   Para cada meridiano j:

   phi = 2π × j / slices

   x = r × cos(phi)

   z = r × sin(phi)
3. Polo sur: (0, -radio, 0)

Caras:

- Triángulos en polos conectando con primer/último anillo

- Cuadriláteros en cuerpo conectando anillos consecutivos

Propiedades:

Nombre="Esfera", Color=Rojo

Ejemplos de vértices:

- subdivisiones=1: ~144 vértices
- subdivisiones=2: ~496 vértices
- subdivisiones=3: ~1104 vértices

3.1.2.3. CrearCilindro

public static Figura3D CrearCilindro(float radio = 0.5f, float altura = 2.0f, int segmentos = 20)

Descripción: Genera cilindro recto con tapas, eje vertical Y.

Parámetros:

- radio: Radio de bases (default: 0.5)
- altura: Altura total (default: 2.0)
- segmentos: Subdivisiones del círculo (default: 20)

Vértices:

Círculo superior (y = altura/2):

Para i = 0 a segmentos-1:

angulo = 2π × i / segmentos

(radio × cos(angulo), altura/2, radio × sin(angulo))

Círculo inferior (y = -altura/2): Igual pero con y negativa

Centros: (0, ±altura/2, 0)

Caras:

- **Laterales**: Cuadriláteros conectando círculos superior/inferior
- **Tapas**: Triángulos desde centros hacia bordes

Propiedades: Nombre="Cilindro", Color=Verde

3.1.2.4. CrearCono

public static Figura3D CrearCono(float radio = 0.5f, float altura = 2.0f, int segmentos = 20)

Descripción: Genera cono circular con base y ápice.

Parámetros:

- radio: Radio de base (default: 0.5)
- altura: Altura total (default: 2.0)
- segmentos: Subdivisiones (mínimo: 6, default: 20)

Vértices:

Ápice: (0, altura/2, 0)

Círculo base (y = -altura/2):

Para i = 0 a segmentos-1:

angulo = 2π × i / segmentos

(radio × cos(angulo), -altura/2, radio × sin(angulo))

Centro base: (0, -altura/2, 0)

Caras:

- **Laterales**: Triángulos [ápice, actual, siguiente] - orden crítico para normales
- **Base**: Triángulos [centro, siguiente, actual] - orden invertido

Propiedades:

Nombre="Cono", Color=Amarillo

Nota:

El orden de vértices determina dirección de la normal (regla mano derecha)

3.1.2.5. CrearPiramide

public static Figura3D CrearPiramide(float base_tamaño = 1.0f, float altura = 1.5f)

Descripción: Genera pirámide cuadrangular.

Parámetros:

- base_tamaño: Lado de base cuadrada (default: 1.0)
- altura: Altura total (default: 1.5)

Vértices:

b = base_tamaño / 2

h = altura / 2

0: Ápice (0, h, 0)

1: Base TI (-b, -h, -b) // Trasera-Izquierda

2: Base TD (b, -h, -b) // Trasera-Derecha

3: Base FD (b, -h, b) // Frontal-Derecha

4: Base FI (-b, -h, b) // Frontal-Izquierda

Caras:

[0,2,1] Frente | [0,3,2] Derecha

[0,4,3] Atrás | [0,1,4] Izquierda

[1,2,3,4] Base (cuadrado)

Propiedades:

Nombre="Pirámide", Color=Naranja

Geometría:

vértices, 5 caras (4 triángulos + 1 cuadrado), 8 aristas

3.1.2.6. CrearToroide

public static Figura3D CrearToroide(float radioMayor = 2.0f, float radioMenor = 0.6f, int segMayor = 24, int segMenor = 12)

Descripción: Genera toro (dona) con dos radios.

Parámetros:

- radioMayor: Radio del círculo central (default: 2.0)
- radioMenor: Radio del tubo (default: 0.6)
- segMayor: Subdivisiones del círculo mayor (mínimo: 8, default: 24)
- segMenor: Subdivisiones del tubo (mínimo: 6, default: 12)

Algoritmo paramétrico:

θ (theta): Ángulo alrededor del círculo mayor (0 a 2π)

φ (phi): Ángulo alrededor del tubo (0 a 2π)

Para i = 0 a segMayor-1:

theta = 2π × i / segMayor

Para j = 0 a segMenor-1:

phi = 2π × j / segMenor

x = (radioMayor + radioMenor × cos(phi)) × cos(theta)

y = radioMenor × sin(phi)

z = (radioMayor + radioMenor × cos(phi)) × sin(theta)

Caras: Cuadriláteros con wrap-around en ambas direcciones (topología cerrada)

Propiedades:

Nombre="Toroide", Color=Azul cielo

Total: segMayor × segMenor vértices y caras

Restricción:

radioMenor < radioMayor (evita autointersección)

3.1.3. Tabla comparativa

Figura	Vértices	Caras	Complejidad	Topología
Cubo	8	6	Baja	Convexa
Esfera	Variable	Variable	Media-Alta	Convexa
Cilindro	2×seg + 2	3×seg	Media	Convexa
Cono	seg + 2	2×seg	Media	Convexa
Pirámide	5	5	Baja	Convexa
Toroide	segM×segm	segM×segm	Alta	Género 1

3.1.4. Convenciones técnicas

3.1.4.1. Sistema de coordenadas

- **Y**: Vertical (+ arriba)
- **X**: Horizontal derecha
- **Z**: Profundidad (+ hacia observador)
- **Origen**: Centro geométrico

3.1.4.2. Orden de vértices

Orden antihorario visto desde fuera (regla mano derecha):

Normal = (v2 - v1) × (v3 - v1) // Producto cruz apunta hacia afuera

3.1.4.3. Wrap-around

En figuras circulares:

int siguiente = (i + 1) % segmentos; // Cierra geometría

3.1.5. Parámetros recomendados

Tiempo real (rendimiento)

```
CrearEsfera(1.0f, 1);          // ~144 vértices
CrearCilindro(0.5f, 2.0f, 12);
CrearToroide(2.0f, 0.6f, 16, 8);
```

Calidad visual

```
CrearEsfera(1.0f, 4);          // ~1936 vértices
CrearCilindro(0.5f, 2.0f, 48);
CrearToroide(2.0f, 0.6f, 48, 24);
```

Debug/prototipo

```
CrearCubo();       // Valores por defecto
CrearEsfera();
CrearCilindro();
```

3.1.6. Ejemplo de uso completo

```
using Motor3D;
// Crear escena con múltiples figuras
var cubo = Figuras3DFactory.CrearCubo(1.5f);
cubo.Posicion = new Vector3D(-3, 0, 0);
cubo.Rotacion = new Vector3D(45, 45, 0);
var esfera = Figuras3DFactory.CrearEsfera(1.0f, 3);
esfera.Posicion = new Vector3D(0, 0, 0);
esfera.Color = Color.Red;
var cilindro = Figuras3DFactory.CrearCilindro(0.5f, 2.0f, 24);
cilindro.Posicion = new Vector3D(3, 0, 0);
cilindro.Rotacion = new Vector3D(0, 0, 90); // Renderizar con motor 3D...
```

3.1.7. Notas importantes

- **Validación**: Algunos métodos validan parámetros mínimos (ej: segmentos < 6)
- **Memoria**: Mayor subdivisión = más vértices = más memoria
- **Rendimiento**: Para tiempo real, usar subdivisiones bajas
- **Normales**: Calculadas por Figura3D.CalcularNormal() usando producto cruz
- **Iluminación**: Requiere normales correctas (orden de vértices crítico)

3.2. Motor3DCore

3.2.1. Descripción técnica general

Motor3DCore implementa el pipeline completo de renderizado 3D: **transformaciones → proyección → rasterización**. Proporciona estructuras matemáticas, transformaciones geométricas, proyección perspectiva e iluminación para aplicaciones 3D con C# y GDI+.

Pipeline: Vértices 3D → Transformaciones → Proyección → Coordenadas 2D → Rasterización

3.2.2. Estructuras básicas

3.2.2.1. Vector3D

Descripción: Vector tridimensional con operaciones matemáticas.

Propiedades: float X, Y, Z

Operaciones:

```
Vector3D suma = a + b;                    // Suma vectorial
Vector3D resta = a - b;                   // Resta vectorial
Vector3D escala = v * 2.5f;               // Multiplicación escalar
float longitud = v.Magnitud();            // √(X² + Y² + Z²)
Vector3D unitario = v.Normalizar();       // Vector longitud 1
float dot = Vector3D.ProductoPunto(a, b);     // a·b (ángulos, proyecciones)
Vector3D cross = Vector3D.ProductoCruz(a, b);  // a×b (perpendicular)
```

Uso: Cálculo de normales, direcciones de luz, operaciones de cámara

3.2.2.2. Matrix4x4

Descripción:

Matriz 4×4 para transformaciones homogéneas 3D.

Estructura:

```
float[,] m = new float[4, 4]
```

Métodos clave:

```
matriz.Identidad();                       // Matriz identidad (diagonal 1s)
Matrix4x4 combinada = matrizA * matrizB;  // Multiplicación (B primero, luego A)
Vector3D transformado = matriz.TransformarPunto(punto);  // Aplica transformación
```

Transformación homogénea: Usa coordenadas (x, y, z, w) donde w permite traslaciones mediante multiplicación matricial.

3.2.3. Transformaciones geométricas

3.2.3.1. Traslación

```
Matrix4x4 t = Motor3DCore.Traslacion(x, y, z);
```

Desplaza objeto en el espacio 3D.

3.2.3.2. Escalamiento

```
Matrix4x4 s = Motor3DCore.Escalamiento(sx, sy, sz);
```

Modifica tamaño. Valores típicos: 1.0 = sin cambio, 2.0 = doble tamaño

3.2.3.3. Rotaciones

Eje X (pitch/cabeceo):

```
Matrix4x4 rx = Motor3DCore.RotacionX(angulo);  // angulo en grados
```

Matriz: [1, 0, 0] [0, cos, -sin] [0, sin, cos]

Eje Y (yaw/guiñada):

```
Matrix4x4 ry = Motor3DCore.RotacionY(angulo);
```

Matriz: [cos, 0, sin] [0, 1, 0] [-sin, 0, cos]

Eje Z (roll/balanceo):

```
Matrix4x4 rz = Motor3DCore.RotacionZ(angulo);
```

Matriz: [cos, -sin, 0] [sin, cos, 0] [0, 0, 1]

Dirección: Rotaciones siguen regla mano derecha

3.2.4. Sistema de cámara y proyección

3.2.4.1. Proyección perspectiva

```
Matrix4x4 p = Motor3DCore.ProyeccionPerspectiva(fov, aspecto, cerca, lejos);
```

Parámetros:

- fov: Campo visión vertical (45-90° típico)
- aspecto: ancho/alto (ej: 16/9 = 1.778)
- cerca: Plano cercano (típico: 0.1)
- lejos: Plano lejano (típico: 100)

Efecto: Objetos lejanos más pequeños, líneas paralelas convergen

Configuración típica: ProyeccionPerspectiva(60f, 16f/9f, 0.1f, 100f)

4.2. Matriz LookAt (cámara)

Matrix4x4 vista = Motor3DCore.LookAt(posicion, objetivo, arriba);

Parámetros:

- posicion: Ubicación cámara
- objetivo: Punto hacia donde mira
- arriba: Vector "arriba" (típico: (0, 1, 0))

Algoritmo:

Z = normalizar(posicion - objetivo) // Eje vista

X = normalizar(arriba × Z) // Eje derecha

Y = Z × X // Eje arriba real

Ejemplo: Cámara en (0,5,10) mirando origen

```
var cam = Motor3DCore.LookAt(
    new Vector3D(0, 5, 10),
    new Vector3D(0, 0, 0),
    new Vector3D(0, 1, 0)
);
```

3.2.5. Iluminación

3.2.5.1. Iluminación Phong simplificada

Color resultado = Motor3DCore.CalcularIluminacion(normal, luz, colorBase, intensidad);

Modelo:

ambiente = 0.2 (fijo)

difuso = max(0, normal · luz)

total = min(1.0, ambiente + difuso × intensidad)

colorFinal = colorBase × total

Componentes:

- **Ambiental**: 20% iluminación uniforme
- **Difuso**: Ley Lambert (normal vs luz)

Ejemplo:

Vector3D luz = new Vector3D(1, 1, 1).Normalizar();

Vector3D normal = figura.CalcularNormal(cara);

Color iluminado = Motor3DCore.CalcularIluminacion(normal, luz, Color.Blue, 0.8f);

3.2.6. Conversión de coordenadas

3.2.6.1. A2D (3D a pantalla)

PointF punto2D = Motor3DCore.A2D(vectorProyectado, ancho, alto);

Transformación:

X_pantalla = (v.X + 1) × ancho / 2

Y_pantalla = (1 - v.Y) × alto / 2

Mapeo: (-1,-1) a (1,1) → (0,0) a (ancho,alto)

7. Clase Figura3D

7.1. Propiedades

```
public class Figura3D
{
   string Nombre;                    // Identificador
   List<Vector3D> Vertices;          // Puntos geometría
   List<int[]> Caras;                // Índices conectividad
   Color Color;                      // Color base

   Vector3D Posicion;                // Traslación
   Vector3D Rotacion;                // Ángulos (grados)
   Vector3D Escala;                  // Factores escala
}
```

Valores por defecto: Origen (0,0,0), sin rotación, escala 1

3.2.7.2. ObtenerMatrizTransformacion

public Matrix4x4 ObtenerMatrizTransformacion()

Retorna: Matriz combinada de todas las transformaciones

Orden (derecha a izquierda):

traslacion × rotY × rotX × rotZ × escala

1. Escala en espacio local
2. Rotaciones (orden Euler YXZ)
3. Traslación a posición final

Uso:

```
Matrix4x4 transform = figura.ObtenerMatrizTransformacion();
Vector3D transformado = transform.TransformarPunto(vertice);
```

3.2.7.3. CalcularNormal

public Vector3D CalcularNormal(int[] cara)

Algoritmo:

v1 = Vertices[cara[1]] - Vertices[cara[0]]

v2 = Vertices[cara[2]] - Vertices[cara[0]]

normal = normalizar(v1 × v2)

Retorna: Vector perpendicular a la cara (longitud 1)

Uso: Cálculo de iluminación por cara

3.2.8. Pipeline completo (ejemplo)

```
// 1. Crear figura
```

```
Figura3D cubo = Figuras3DFactory.CrearCubo(2.0f);
cubo.Posicion = new Vector3D(0, 0, -5);
cubo.Rotacion = new Vector3D(30, 45, 0);
// 2. Configurar cámara y proyección
var vista = Motor3DCore.LookAt(
    new Vector3D(0, 2, 5),
    new Vector3D(0, 0, 0),
    new Vector3D(0, 1, 0)
);
var proyeccion = Motor3DCore.ProyeccionPerspectiva(60f, 16f/9f, 0.1f, 100f);
// 3. Renderizar
var matrizModelo = cubo.ObtenerMatrizTransformacion();
Vector3D luz = new Vector3D(1, 1, 1).Normalizar();
foreach (var cara in cubo.Caras)
{
    // Calcular iluminación
    Vector3D normal = cubo.CalcularNormal(cara);
    Color color = Motor3DCore.CalcularIluminacion(normal, luz, cubo.Color, 0.8f);

    // Proyectar vértices
    List<PointF> puntos2D = new List<PointF>();
    foreach (int idx in cara)
    {
        Vector3D v = cubo.Vertices[idx];
        v = matrizModelo.TransformarPunto(v);      // Transformación modelo
        v = vista.TransformarPunto(v);             // Transformación vista
        v = proyeccion.TransformarPunto(v);        // Proyección
        puntos2D.Add(Motor3DCore.A2D(v, 800, 600)); // A pantalla
    }

    // Dibujar
    g.FillPolygon(new SolidBrush(color), puntos2D.ToArray());
}
```

3.2.9. Configuraciones típicas

Proyección

Escenario	FOV	Cerca	Lejos
Normal	60°	0.1	100
Gran angular	90°	0.1	100

Interior	50°	0.01	50
Exterior	70°	1.0	1000

Cámara

Isométrica:

LookAt(new Vector3D(10,10,10), new Vector3D(0,0,0), new Vector3D(0,1,0))

Orbital:

float ang = tiempo × 0.1f;

LookAt(new Vector3D(cos(ang)×15, 5, sin(ang)×15), Vector3D(0,0,0), Vector3D(0,1,0))

Iluminación

- Luz solar: intensidad = 1.0
- Luz interior: 0.6 - 0.8
- Luz tenue: 0.3 - 0.5
- Ambiental: 0.2 (fijo)

3.2.10. Notas importantes

- **Coordenadas homogéneas**: Permiten traslaciones con multiplicación matricial
- **Orden transformaciones**: Crítico (no conmutativo)
- **Normales**: Deben apuntar hacia afuera para iluminación correcta
- **División por w**: En TransformarPunto para proyección perspectiva
- **Ángulos**: Todas las rotaciones en grados (convertidos a radianes internamente)
- **Sistema coordenadas**: Y arriba, X derecha, Z hacia observador

3.3. Camara3D

3.3.1. Descripción técnica general

Camara3D es un sistema de cámara 3D con tres modos de operación: **Orbital** (gira alrededor de objetivo), **Libre** (movimiento FPS) y **Fija** (estática). Maneja posicionamiento, rotación, zoom y genera matrices de vista/proyección para el motor 3D.

Propósito: Simplificar control de cámara en aplicaciones 3D interactivas.

3.3.2. Modos de cámara

3.3.2.1. ModoCamara (enum)

```
public enum ModoCamara
{
   Orbital,    // Gira alrededor de punto objetivo
   Libre,      // Movimiento libre (estilo FPS)
   Fija        // Cámara estática sin movimiento
}
```

3.3.3. Propiedades

3.3.3.1. Propiedades básicas

```
Vector3D Posicion;              // Ubicación cámara en espacio 3D
Vector3D Objetivo;              // Punto hacia donde mira
Vector3D Arriba;                // Vector "arriba" (típico: (0,1,0))
```

ModoCamara Modo; // Modo actual

3.3.3.2. Parámetros cámara orbital

float Distancia; // Distancia desde objetivo (zoom)

float AnguloAzimutal; // Rotación horizontal (0-360°)

float AnguloPolar; // Rotación vertical (1-179°)

3.3.3.3. Parámetros proyección

float CampoVision; // FOV en grados (típico: 60)

float PlanoConjunto; // Plano cercano (típico: 0.1)

float PlanoLejano; // Plano lejano (típico: 100)

3.3.4. Constructor

public Camara3D()

Valores por defecto:

Posición: (0, 2, 5)

Objetivo: (0, 0, 0)

Arriba: (0, 1, 0)

Modo: Orbital

Distancia: 5.0

AnguloAzimutal: 0°

AnguloPolar: 30°

CampoVisión: 60°

PlanoConjunto: 0.1

PlanoLejano: 100.0

Comportamiento: Inicializa en modo orbital mirando al origen desde posición elevada.

3.3.5. Métodos principales

3.3.5.1. ActualizarPosicionOrbital

public void ActualizarPosicionOrbital()

Descripción: Calcula posición cámara en coordenadas esféricas (modo orbital).

Algoritmo:

// Limitar ángulo polar [1°, 179°] (evita gimbal lock)

AnguloPolar = clamp(AnguloPolar, 1, 179)

// Convertir a radianes

radAz = AnguloAzimutal × π/180

radPol = AnguloPolar × π/180

// Coordenadas esféricas → cartesianas

x = Distancia × sin(radPol) × cos(radAz)

y = Distancia × cos(radPol)

z = Distancia × sin(radPol) × sin(radAz)

Posicion = Objetivo + (x, y, z)

Comportamiento: Solo actúa si Modo == Orbital

3.3.5.2. RotarOrbital

public void RotarOrbital(float deltaAzimutal, float deltaPolar)

Parámetros:

- deltaAzimutal: Incremento rotación horizontal (grados)
- deltaPolar: Incremento rotación vertical (grados)

Comportamiento:

AnguloAzimutal += deltaAzimutal

AnguloPolar += deltaPolar

// Normalizar azimutal [0°, 360°]

AnguloAzimutal = wrap(AnguloAzimutal, 0, 360)

ActualizarPosicionOrbital()

Uso típico: Arrastrar mouse para orbitar

camara.RotarOrbital(mouseDeltaX * 0.5f, mouseDeltaY * 0.5f);

3.3.5.3. Zoom

public void Zoom(float delta)

Descripción: Ajusta distancia al objetivo (acerca/aleja cámara).

Parámetros:

- delta: Cambio en distancia (positivo = alejar, negativo = acercar)

Límites: Distancia restringida a [1.0, 50.0]

Uso típico: Rueda del mouse

camara.Zoom(mouseWheelDelta * -0.1f); // Invertir para comportamiento natural

3.3.5.4. MoverLibre

public void MoverLibre(float adelante, float derecha, float arriba)

Descripción: Mueve cámara en modo libre (estilo FPS).

Parámetros:

- adelante: Movimiento hacia adelante/atrás
- derecha: Movimiento lateral izquierda/derecha
- arriba: Movimiento vertical arriba/abajo

Algoritmo:

direccion = normalizar(Objetivo - Posicion) // Eje Z

lateral = normalizar(direccion × Arriba)   // Eje X

vertical = normalizar(lateral × direccion)   // Eje Y

// Aplicar movimiento en todos los ejes

Posicion += direccion×adelante + lateral×derecha + vertical×arriba

Objetivo += direccion×adelante + lateral×derecha + vertical×arriba

Uso típico: Teclas WASD

if (tecla_W) camara.MoverLibre(0.1f, 0, 0);   // Adelante

if (tecla_S) camara.MoverLibre(-0.1f, 0, 0);   // Atrás

if (tecla_A) camara.MoverLibre(0, -0.1f, 0);   // Izquierda

if (tecla_D) camara.MoverLibre(0, 0.1f, 0);   // Derecha

if (tecla_Space) camara.MoverLibre(0, 0, 0.1f); // Arriba

if (tecla_Ctrl) camara.MoverLibre(0, 0, -0.1f); // Abajo

3.3.5.5. RotarLibre

public void RotarLibre(float horizontal, float vertical)

Descripción: Rota cámara en modo libre (look around).

Parámetros:

- horizontal: Rotación izquierda/derecha (grados)
- vertical: Rotación arriba/abajo (grados)

Algoritmo:

direccion = normalizar(Objetivo - Posicion)

distancia = magnitud(Objetivo - Posicion)

// Rotación horizontal (alrededor eje Y)

direccion = RotacionY(horizontal) × direccion

// Rotación vertical (alrededor eje lateral)

lateral = normalizar(direccion × Arriba)

direccion = direccion×cos(vertical) + Arriba×sin(vertical)

direccion = normalizar(direccion)

Objetivo = Posicion + direccion×distancia

Uso típico: Movimiento del mouse

camara.RotarLibre(mouseDeltaX * 0.1f, mouseDeltaY * 0.1f);

3.3.5.6. ObtenerMatrizVista

public Matrix4x4 ObtenerMatrizVista()

Retorna: Matriz LookAt de la cámara actual

Equivalente a:

return Motor3DCore.LookAt(Posicion, Objetivo, Arriba);

Uso: Pipeline de renderizado

Matrix4x4 vista = camara.ObtenerMatrizVista();

Vector3D transformado = vista.TransformarPunto(vertice);

3.3.5.7. ObtenerMatrizProyeccion

public Matrix4x4 ObtenerMatrizProyeccion(float aspectRatio)

Parámetros:

- aspectRatio: Relación aspecto (ancho/alto)

Retorna: Matriz proyección perspectiva

Equivalente a:

return Motor3DCore.ProyeccionPerspectiva(CampoVision, aspectRatio, PlanoConjunto, PlanoLejano);

Uso:

float aspecto = (float)ancho / alto;

Matrix4x4 proyeccion = camara.ObtenerMatrizProyeccion(aspecto);

3.3.5.8. Reiniciar

public void Reiniciar()

Descripción: Restaura cámara a valores por defecto según modo actual.

Comportamiento por modo:

Orbital:

Objetivo: (0, 0, 0)

Distancia: 5.0

AnguloAzimutal: 0°

AnguloPolar: 30°

Libre:

Posicion: (0, 2, 5)

Objetivo: (0, 0, 0)

Fija:

Posicion: (0, 3, 8)

Objetivo: (0, 0, 0)

3.3.5.9. CambiarModo

public void CambiarModo(ModoCamara nuevoModo)

Descripción: Cambia modo de cámara y reinicia a valores por defecto.

Uso:

```
camara.CambiarModo(ModoCamara.Libre);     // Cambiar a modo libre
camara.CambiarModo(ModoCamara.Orbital);   // Volver a orbital
```

3.3.5.10. EnfocarObjetivo

public void EnfocarObjetivo(Vector3D nuevoObjetivo)

Descripción: Establece nuevo punto objetivo (útil para centrar en objetos).

Parámetros:

- nuevoObjetivo: Nuevo punto a mirar

Comportamiento: Si Modo == Orbital, recalcula posición automáticamente.

Uso:

```
// Enfocar en objeto seleccionado
camara.EnfocarObjetivo(figura.Posicion);
```

3.3.5.11. ObtenerDireccion

public Vector3D ObtenerDireccion()

Retorna: Vector dirección normalizado (hacia donde mira la cámara)

Cálculo: normalizar(Objetivo - Posicion)

Uso: Cálculo de rayos, picking, física

3.3.5.12. ObtenerInfo

public string ObtenerInfo()

Retorna: String formateado con información de cámara para UI/debug

Formato:

```
Modo: Orbital
Posición: (1.23, 4.56, 7.89)
Objetivo: (0.00, 0.00, 0.00)
Distancia: 5.00
```

Ángulos: Az=45.0° Pol=30.0°

Uso: Mostrar en label o consola de debug

3.3.6. Casos de uso por modo

3.3.6.1. Modo Orbital (inspección de objetos)

```
Camara3D cam = new Camara3D();
cam.Modo = ModoCamara.Orbital;
cam.EnfocarObjetivo(new Vector3D(0, 0, 0));
// Interacción
void OnMouseDrag(float dx, float dy)
{
   cam.RotarOrbital(dx * 0.5f, dy * 0.5f);
}
void OnMouseWheel(float delta)
{
   cam.Zoom(delta * -0.1f);
}
```

Aplicaciones: Visualizadores 3D, editores de modelos, inspección de productos

3.3.6.2. Modo Libre (navegación FPS

```
Camara3D cam = new Camara3D();
cam.CambiarModo(ModoCamara.Libre);
// Bucle de actualización
void Update()
{
   // Movimiento WASD
   float velocidad = 0.1f;
   if (tecla_W) cam.MoverLibre(velocidad, 0, 0);
   if (tecla_S) cam.MoverLibre(-velocidad, 0, 0);
   if (tecla_A) cam.MoverLibre(0, -velocidad, 0);
   if (tecla_D) cam.MoverLibre(0, velocidad, 0);
   // Look con mouse
   cam.RotarLibre(mouseDeltaX * 0.1f, mouseDeltaY * 0.1f);
}
```

Aplicaciones: Juegos FPS, recorridos virtuales, exploradores de escenas

3.3.6.3. Modo Fija (presentaciones)

```
Camara3D cam = new Camara3D();
cam.CambiarModo(ModoCamara.Fija);
cam.Posicion = new Vector3D(10, 5, 10);
cam.EnfocarObjetivo(new Vector3D(0, 0, 0));
// No requiere actualización (estática)
```

Aplicaciones: Renders estáticos, screenshots, animaciones con keyframes

3.3.7. Integración con motor 3D

```
// Setup
Camara3D camara = new Camara3D();
camara.Modo = ModoCamara.Orbital;
float aspectRatio = (float)ancho / alto;
// Bucle de renderizado
void Render()
{
   Matrix4x4 vista = camara.ObtenerMatrizVista();
   Matrix4x4 proyeccion = camara.ObtenerMatrizProyeccion(aspectRatio);

   foreach (var figura in escena)
   {
      Matrix4x4 modelo = figura.ObtenerMatrizTransformacion();
      foreach (var cara in figura.Caras)
      {
         List<PointF> puntos2D = new List<PointF>();
         foreach (int idx in cara)
         {
            Vector3D v = figura.Vertices[idx];
            v = modelo.TransformarPunto(v);        // Modelo
            v = vista.TransformarPunto(v);         // Vista
            v = proyeccion.TransformarPunto(v);    // Proyección
            puntos2D.Add(Motor3DCore.A2D(v, ancho, alto));
         }
         g.FillPolygon(brush, puntos2D.ToArray());
      }
   }
}
```

3.3.8. Límites y restricciones

3.3.8.1. Ángulo polar

- **Rango**: [1°, 179°]
- **Motivo**: Evita gimbal lock en polos (0° y 180°)
- **Comportamiento**: Clampea automáticamente

3.3.8.2. Distancia orbital

- **Rango**: [1.0, 50.0]
- **Motivo**: Evita acercamiento excesivo o alejamiento infinito
- **Ajustable**: Modificar límites en método Zoom()

3.3.8.3. Normalización azimutal

- **Rango**: [0°, 360°]

- **Comportamiento**: Wrap automático (360° → 0°)

3.3.9. Configuraciones recomendadas

Visualización arquitectónica

```
camara.Modo = ModoCamara.Orbital;
camara.CampoVision = 50f;          // FOV reducido
camara.Distancia = 10f;
camara.AnguloPolar = 45f;
```

Juego/navegación

```
camara.Modo = ModoCamara.Libre;
camara.CampoVision = 75f;          // FOV amplio
camara.PlanoConjunto = 0.01f;      // Más cercano
```

Presentación/render

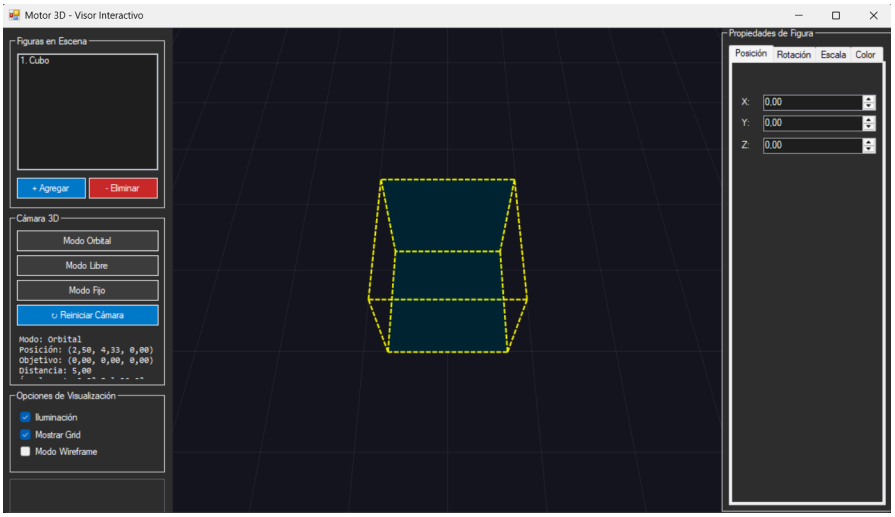
```
camara.Modo = ModoCamara.Fija;
camara.CampoVision = 60f;
camara.Posicion = new Vector3D(8, 6, 8);
camara.EnfocarObjetivo(new Vector3D(0, 0, 0));
```

3.3.10. Notas importantes

- **Thread-safety**: No es thread-safe, usar desde un solo thread
- **Coordenadas esféricas**: Orbital usa (distancia, azimutal, polar)
- **Gimbal lock**: Evitado limitando ángulo polar a (1°, 179°)
- **Persistencia**: Estado cámara puede serializarse guardando propiedades públicas
- **Performance**: Operaciones ligeras, aptas para tiempo real (60+ FPS)
- **Y-up**: Sistema asume Y como eje vertical (arriba)

3.4. Justificación del diseño gráfico

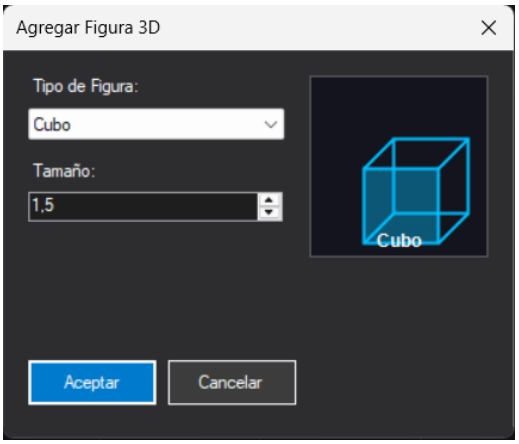
3.4.1. Form3DViewer



El diseño del formulario adopta una estructura basada en tres áreas principales: un panel izquierdo de control, un área central de visualización y un panel derecho de propiedades.

- El panel izquierdo agrupa los controles relacionados con la gestión de la escena y la cámara. La sección Figuras en Escena permite una administración clara de los objetos 3D mediante una lista y botones diferenciados por color para acciones de agregar y eliminar. El grupo Cámara 3D organiza los distintos modos de cámara en botones alineados verticalmente. Se incluyó un área informativa de la cámara para brindar retroalimentación al usuario sobre el estado actual de la visualización. Finalmente, tenemos el grupo Opciones de Visualización que concentra configuraciones globales como iluminación, grilla y modo wireframe, permitiendo activar o desactivar características visuales sin saturar la interfaz.
- El área central de visualización utiliza un esquema de colores oscuros para reducir la fatiga visual y mejorar el contraste de las figuras renderizadas.
- El panel derecho contiene las propiedades de la figura seleccionada, esto se realiza con el fin de separar las opciones, entre acciones globales y configuraciones específicas.

3.4.2. FormAgregarFiguras



El formulario FormAgregarFigura fue diseñado como una ventana modal de apoyo al formulario principal, con el objetivo de permitir la creación de nuevas figuras tridimensionales.

La organización de los controles sigue una disposición vertical y secuencial, comenzando con la selección del tipo de figura mediante un ComboBox de lista cerrada. Se utilizaron etiquetas descriptivas ubicadas sobre cada control.

El control NumericUpDown para el tamaño de la figura permite definir valores dentro de un rango controlado. Se incluye un panel de previsualización panelPreview, el cual permite al usuario observar una representación básica de la figura seleccionada antes de confirmarla.

3.5. Reflexión sobre las decisiones de diseño

Las decisiones de diseño adoptadas fueron tomadas para con el fin de ofrecer un equilibrio entre funcionalidad, usabilidad y claridad visual. Desde una perspectiva reflexiva, se priorizó que la interfaz no solo cumpliera con los requerimientos técnicos del sistema, sino que también facilitara la comprensión y el control del entorno tridimensional por parte del usuario.

La separación del formulario principal en áreas bien definidas permite que cada grupo de controles tenga un propósito específico, evitando la saturación visual y favoreciendo una navegación intuitiva.

La previsualización de figuras y la información de la cámara ayudan en la interacción del usuario con el sistema con el propósito de que comprenda el impacto de sus acciones.

Referencias Teóricas

- Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F. (1995). *Computer Graphics: Principles and Practice*. Addison-Wesley.
- Akenine-Möller, T., Haines, E., & Hoffman, N. (2018). *Real-Time Rendering* (4th ed.). CRC Press.

- Dunn, F., & Parberry, I. (2011). *3D Math Primer for Graphics and Game Development* (2nd ed.). CRC Press.
- Shirley, P., & Marschner, S. (2015). *Fundamentals of Computer Graphics* (4th ed.). CRC Press.
- Hughes, J. F., van Dam, A., McGuire, M., Sklar, D. F., Foley, J. D., Feiner, S. K., & Akeley, K. (2013). *Computer Graphics: Principles and Practice* (3rd ed.). Addison-Wesley.