BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

Práctica 5

Implementación de una escena 3D realista con OpenGL

Graficación | M.C. Luis Yael Méndez Sánchez

- Gerardo Cabrera Morales - 202322210
- Astrid García Ramírez - 202330744
- Odalys Daniela Hernández Méndez - 202331904
- Alan Ortiz Pérez - 202349577

Primavera 2025

ÍNDICE

INTRODUCCIÓN

Para el siguiente trabajo, el equipo trabajó para cumplir con éxito todas las instrucciones proporcionadas e integrar con éxito las bibliotecas PyGame, OpenGL y NumPy para desarrollar aplicaciones 3D interactivas con un nivel de realismo y funcionalidad. Se ha implementado un menú principal que le permite elegir al usuario entre cinco formas geométricas (cubo, pirámide, esfera, cilindro y elipsoide) mediante las teclas numéricas, ocultándose automáticamente al seleccionar una opción para dar paso a una escena dedicada con controles interactivos.

Durante la visualización, se habilitan transformaciones completas, así como cambios entre perspectiva y proyección paralela (tecla P), encendido/apagado de iluminación (tecla I) y texturas (tecla T), mediante el uso del búfer Z y sombras. Cada gráfico está encapsulado en un módulo separado, con un método draw() y la capacidad de aplicar texturas. Además, se implementa la rotación en modo local/global (tecla M) alrededor de cualquier eje, resaltando la diferencia entre los dos sistemas de coordenadas.

La carga y aplicación de texturas se realiza mediante PyGame, lo que garantiza una visualización coherente, mientras que la estructura del código facilita la colaboración en equipo al dividir las responsabilidades.

CLASE CUBO:

Esta clase permite crear un cubo en un entorno 3D con la ayuda de OpenGL

El código consta de tres funciones principales:

1. Constructor __init__()
2. Cargador de textura load_texture()
3. Dibujador del cilindro draw()

A continuación se describe el funcionamiento detallado de cada función.

Función __init__(self, size=1.0)

define el tamaño del cubo y su textura

- Size: define el tamaño que tendrán las caras del cubo
- Texture: define la textura que tendrá el cubo, su valor cambia al cargar una nueva textura

```python
class Cubo:
    def __init__(self, size=1.0):
        self.size = size
        self.texture = None
```

Función load_texture(self, surface)

Este método permite cargar de una imagen la textura que se aplica a el objeto 3D (cubo)
A continuación se describe el método:

- Convierte la imagen a bytes en formato RGB. (pygame.image.tostring()
- Obtiene las dimensiones de la imagen. (surface.get_size()
- Genera un nuevo ID de textura y lo activa para configuraciones posteriores. (glGlenTextures())
- Define los filtros de minimización y magnificación como lineales (suavizado). (glTexParameteri())

- Carga los datos de la textura en la memoria de video de la GPU. (glTextImage2D()

```
def load_texture(self, surface):
    tex_data = pygame.image.tostring(surface, "RGB", True)
    w, h = surface.get_size()
    self.texture = glGenTextures(1)
    glBindTexture(GL_TEXTURE_2D, self.texture)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0, GL_RGB, GL_UNSIGNED_BYTE, tex_data)
```

Función draw(self)

Esta función permite renderizar el cubo con su textura asignada

- Para centrar los vértices con respecto al origen usamos la variable s = self.size/2
- Se activa la textura para cada una de las caras del cubo
- se crea una lista para tener pares (vector, vértices), el vector sirve para la iluminación de la cara, los vértices definen el límite de cada cara

```
def draw(self):
    s = self.size / 2
    glBindTexture(GL_TEXTURE_2D, self.texture)
    faces = [
        ((0,0,1),  [(-s,-s, s),( s,-s, s),( s, s, s),(-s, s, s)]),
        ((0,0,-1), [( s,-s,-s),(-s,-s,-s),(-s, s,-s),( s, s,-s)]),
        ((-1,0,0), [(-s,-s,-s),(-s,-s, s),(-s, s, s),(-s, s,-s)]),
        ((1,0,0),  [( s,-s, s),( s,-s,-s),( s, s,-s),( s, s, s)]),
        ((0,1,0),  [(-s, s, s),( s, s, s),( s, s,-s),(-s, s,-s)]),
        ((0,-1,0), [(-s,-s,-s),( s,-s,-s),( s,-s, s),(-s,-s, s)])
    ]
```

- se recorre la lista
- se dibuja la cara (con los vértices)
- se le asigna el vector de iluminación para la cara

```
        for normal, verts in faces:
            glBegin(GL_QUADS)
            glNormal3f(*normal)
```

- se asignan las coordenadas para la textura

- se dibujan los vértices en las coordenadas 3D

```
for i, v in enumerate(verts):
    glTexCoord2f(i%2, i//2)
    glVertex3f(*v)
glEnd()
```

CLASE PIRÁMIDE:

Se define una clase llamada `Piramide` que permite renderizar una pirámide de base cuadrada en un entorno 3D con **OpenGL**
La clase incluye:s

1. Un constructor que inicializa parámetros geométricos.
2. Una función `load_texture()` que carga texturas desde superficies Pygame.
3. Una función `draw()` que renderiza la pirámide con textura.

Función __init__(self, size=1.0)

Inicializa la pirámide con un tamaño definido y sin textura asociada por defecto.

- `size`: representa la longitud total de un lado de la base de la pirámide.
- Se divide entre dos más adelante para definir el semilado (`s`), útil para el centrado.
- `texture`: se inicializa como `None`, y se asignará al cargar una imagen como textura.

```
5   class Piramide:
6       def __init__(self, size=1.0):
7           self.size = size
8           self.texture = None
```

Función load_texture(self, surface)

Carga una imagen como textura OpenGL desde una superficie de Pygame.

- Convierte la imagen a bytes en formato RGB.
- Obtiene las dimensiones de la imagen.
- Genera un nuevo ID de textura y lo activa para configuraciones posteriores.
- Define los filtros de minimización y magnificación como lineales (suavizado).
- Carga los datos de la textura en la memoria de video de la GPU.

```
10      def load_texture(self, surface):
11          tex_data = pygame.image.tostring(surface, "RGB", True)
12          w, h = surface.get_size()
13          self.texture = glGenTextures(1)
14          glBindTexture(GL_TEXTURE_2D, self.texture)
15          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
16          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
17          glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0, GL_RGB, GL_UNSIGNED_BYTE, tex_data)
```

Función draw(self)

Renderiza la pirámide 3D, aplicando la textura previamente cargada a sus caras laterales y base.

- Calcula `s`, la mitad del tamaño, para posicionar vértices centrados en el origen.
- Asocia la textura cargada al contexto actual.
- Cada entrada representa una cara triangular lateral de la pirámide:
- Una normal predefinida (orientación de la superficie).
- Tres vértices en sentido antihorario para formar el triángulo:
    - El vértice superior común `(0, s, 0)`.
    - Dos vértices en la base.

```
19      def draw(self):
20          s = self.size / 2
21          glBindTexture(GL_TEXTURE_2D, self.texture)
22          faces = [
23              ((0,math.sqrt(2)/2, math.sqrt(2)/2), [(0,s,0),(-s,-s,s),(s,-s,s)]),
24              ((math.sqrt(2)/2,math.sqrt(2)/2,0), [(0,s,0),(s,-s,s),(s,-s,-s)]),
25              ((0,math.sqrt(2)/2,-math.sqrt(2)/2),[(0,s,0),(s,-s,-s),(-s,-s,-s)]),
26              ((-math.sqrt(2)/2,math.sqrt(2)/2,0),[(0,s,0),(-s,-s,-s),(-s,-s,s)])
27          ]
```

Para cada cara:

- Se inicia un `GL_TRIANGLES`.
- Se define su normal con `glNormal3f`.
- Para cada vértice:
    - Se calcula una coordenada de textura `u` y `vtex` de forma manual.
        - El vértice superior central (índice 0) tiene coordenadas `(0.5, 1)`.
        - Los vértices de base tienen `(0, 0)` o `(1, 0)` según su posición.
    - Se especifican coordenadas de textura (`glTexCoord2f`) y posición (`glVertex3f`).
- Se cierra el triángulo con `glEnd()`.

```
28              for normal, verts in faces:
29                  glBegin(GL_TRIANGLES)
30                  glNormal3f(*normal)
31                  for i, v in enumerate(verts):
32                      u = 0.5 if i==0 else (0 if v[0]<0 else 1)
33                      vtex = 1 if i==0 else 0
34                      glTexCoord2f(u, vtex)
35                      glVertex3f(*v)
36                  glEnd()
```

- Inicia el dibujo de un cuadrado con `GL_QUADS`.

- Define la normal hacia abajo (`(0, -1, 0)`).
- Define los cuatro vértices de la base (en sentido antihorario).
- Calcula coordenadas de textura de forma sistemática para los cuatro vértices:
  - `(0,0)`, `(1,0)`, `(1,1)`, `(0,1)`
- Dibuja el cuadrado.

```
37          glBegin(GL_QUADS)
38          glNormal3f(0,-1,0)
39          base = [(-s,-s,s),(s,-s,s),(s,-s,-s),(-s,-s,-s)]
40          for i, v in enumerate(base):
41              glTexCoord2f(i%2, i//2)
42              glVertex3f(*v)
43          glEnd()
```

CLASE ESFERA:

Esta clase permite modelar, texturizar y renderizar una esfera tridimensional usando OpenGL
Los métodos de la clase esfera se definen a continuación

- Constructor __init__()
- Cargador de textura load_texture()
- Dibujador del cilindro draw()

Constructor__init__(self, radius=1.0)

El constructor permite definir los atributos que tendrá la esfera, como el radio, la textura y la superficie

- self.radius: se asigna el radio de la esfera
- self.texture: se asigna la textura inicial
- self.quad: crea el objeto cuadrático para la textura
- gluQuadricTexture: permite aplicar texturas al objeto cuadrático creado previamente

```
def __init__(self, radius=1.0):
    self.radius = radius
    self.texture = None
    self.quad = gluNewQuadric()
    gluQuadricTexture(self.quad, GL_TRUE)
```

Método load_texture(self, surface)

Este método permite cargar de una imagen la textura que se aplica a el objeto 3D (cubo)
A continuación se describe el método:

- Convierte la imagen a bytes en formato RGB. (pygame.image.tostring()
- Obtiene las dimensiones de la imagen. (surface.get_size()
- Genera un nuevo ID de textura y lo activa para configuraciones posteriores. (glGlenTextures())
- Define los filtros de minimización y magnificación como lineales (suavizado). (glTexParameteri())
- Carga los datos de la textura en la memoria de video de la GPU. (glTextImage2D()

```
def load_texture(self, surface):
    tex_data = pygame.image.tostring(surface, "RGB", True)
    w, h = surface.get_size()
    self.texture = glGenTextures(1)
    glBindTexture(GL_TEXTURE_2D, self.texture)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0, GL_RGB, GL_UNSIGNED_BYTE, tex_data)
```

Método draw(self)

Este método se encarga de renderizar la esfera, usando los atributos del objeto y la textura cargada previamente

- glBlindTexture: activa la textura cargada del objeto
- gluSphere: dibuja la esfera usando el objeto cuadrático, el radio y el nivel de detalle

```
def draw(self):
    glBindTexture(GL_TEXTURE_2D, self.texture)
    gluSphere(self.quad, self.radius, 32, 32)
```

CLASE CILINDRO:

Implementa una clase denominada Cilindro, la cual permite modelar, texturizar y renderizar un cilindro tridimensional mediante OpenGL.

El código consta de tres funciones principales:

4. Constructor __init__()
5. Cargador de textura load_texture()
6. Dibujador del cilindro draw()

A continuación se describe el funcionamiento detallado de cada función.

Función __init__(self, base=1.0, top=1.0, height=2.0)

Este método es el constructor de la clase `Cilindro`. Se encarga de inicializar las propiedades geométricas del cilindro (radio inferior, radio superior y altura),

- Define tres parámetros opcionales:
- `base`: radio del círculo de la base inferior.
- `top`: radio del círculo superior.
- `height`: altura del cilindro desde la base hasta la parte superior.
- Asigna los valores de los parámetros a atributos del objeto.
- Inicializa el atributo `texture` como `None`, para luego contener un identificador de textura OpenGL.
- Crea un nuevo objeto cuadrático (tipo de objeto GLU) que se utilizará para renderizar primitivas como cilindros y discos.
- Este objeto permite simplificar el uso de funciones como `gluCylinder()` y `gluDisk()`.
- Habilita la generación automática de coordenadas de textura para el objeto cuadrático. Esto es necesario para poder aplicar imágenes como textura en el cilindro.

```
5   class Cilindro:
6       def __init__(self, base=1.0, top=1.0, height=2.0):
7           self.base = base
8           self.top = top
9           self.height = height
10          self.texture = None
11          self.quad = gluNewQuadric()
12          gluQuadricTexture(self.quad, GL_TRUE)
```

Función load_texture(self, surface)

Esta función carga una imagen como textura y la prepara para ser utilizada por OpenGL. La textura proviene de una superficie (`surface`) generada con Pygame.

- Recibe como parámetro una superficie Pygame, que puede haber sido creada a partir de una imagen mediante `pygame.image.load()`.
- Convierte la superficie a una cadena de bytes con formato RGB, adecuada para ser interpretada por OpenGL.
- El tercer parámetro `True` invierte la imagen verticalmente (como requiere OpenGL).
- Obtiene el ancho (`w`) y la altura (`h`) de la superficie de la textura.
- Genera un nuevo ID de textura (entero único) y lo vincula al tipo `GL_TEXTURE_2D`, indicando que se va a trabajar con una textura bidimensional.
- Define los parámetros de filtrado:
    - `GL_TEXTURE_MIN_FILTER`: especifica el tipo de filtrado cuando la textura se reduce. Se usa interpolación lineal (`GL_LINEAR`).
    - `GL_TEXTURE_MAG_FILTER`: especifica el tipo de filtrado cuando la textura se amplía.
- Envía la textura a la memoria de la GPU.

```
14      def load_texture(self, surface):
15          tex_data = pygame.image.tostring(surface, "RGB", True)
16          w, h = surface.get_size()
17          self.texture = glGenTextures(1)
18          glBindTexture(GL_TEXTURE_2D, self.texture)
19          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
20          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
21          glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0, GL_RGB, GL_UNSIGNED_BYTE, tex_data)
```

Función draw(self)

Renderiza visualmente el cilindro en una escena 3D, aplicando la textura cargada y dibujando la superficie lateral y las tapas (inferior y superior) del cilindro.

- Método sin parámetros externos. Dibuja el cilindro completo según la configuración actual.
- Asocia la textura previamente cargada con la superficie que se va a renderizar.
- Guarda el estado actual de la matriz de transformación. Esto es importante para que cualquier transformación aplicada al cilindro no afecte a otros objetos de la escena.

- Rota el sistema de coordenadas local -90 grados alrededor del eje `x`. Esto cambia la orientación del cilindro de modo que su eje longitudinal (por defecto alineado con `z`) ahora apunte hacia arriba (`y`).
- Dibuja la superficie lateral del cilindro.
- Parámetros:
    - `self.quad`: objeto cuadrático.
    - `self.base`, `self.top`: radios de los extremos.
    - `self.height`: altura.
    - `32`: número de sectores en el círculo (más alto = mayor resolución).
    - `16`: número de divisiones a lo largo del eje de altura.
- Dibuja la tapa inferior del cilindro como un disco plano con el mismo radio que la base.
- Traslada el sistema de coordenadas local en el eje `z` (que tras la rotación es `y`) para posicionarse en la parte superior del cilindro.
- Dibuja la tapa superior del cilindro, igual que la inferior pero en la parte opuesta.
- Restaura la matriz de transformación anterior, garantizando que cualquier transformación usada para dibujar el cilindro no afecte el resto de la escena 3D.:

```
23      def draw(self):
24          glBindTexture(GL_TEXTURE_2D, self.texture)
25          glPushMatrix()
26          glRotatef(-90,1,0,0)
27          gluCylinder(self.quad, self.base, self.top, self.height, 32, 16)
28          gluDisk(self.quad, 0, self.base, 32, 1)
29          glTranslatef(0,0,self.height)
30          gluDisk(self.quad, 0, self.top, 32, 1)
31          glPopMatrix()
```

CLASE SUPERELIPSOIDE:

El código tiene 3 funciones principales:

1. Constructor __init__() que inicializa los parámetros de forma y curvatura del sólido.
2. Una función load_texture() que carga texturas desde superficies Pygame.
3. Dibujador del cilindro draw() que dibuja el superelipsoide con textura en la escena 3D.

Se describe el funcionamiento detallado de cada función incluyendo las tres funciones principales.

Función init

- a, b, c: Es el escalado en los ejes X, Y y Z que determinan el tamaño del sólido en cada dirección.
- n1, n2: Son exponentes que controlan la forma redondeada o cúbica del sólido (valores cercanos a 1 lo hacen redondo, valores mayores lo hacen de forma más cúbica).
- texture: Se inicializa como None para luego almacenar el identificador de textura OpenGL.

```
def __init__(self, a=1, b=1, c=1, n1=0.5, n2=1.0):
    self.a, self.b, self.c = a, b, c
    self.n1, self.n2 = n1, n2
    self.texture = None
```

Función sign

La función sign sirve para obtener el signo de un número y se usa para preservar la dirección del número al aplicar potencias fraccionarias, lo cual es importante al trabajar con superelipsoides.

- Devuelve el signo de un valor (1 si es positivo, -1 si es negativo).
- Se usa para mantener la dirección al elevar a potencias fraccionarias.

```
def sign(self, v): return math.copysign(1, v)
```

Función fexp

La función fexp es una función para calcular potencias con signo de una manera controlada, especialmente útil para generar formas suaves.

- Calcula una potencia preservando el signo.
- Permite modelar formas suaves o afiladas según el exponente.

```
def fexp(self, base, exp): return self.sign(base) * (abs(base)**exp)
```

Función load_texture

- Convierte una superficie Pygame (imagen cargada) en una textura de OpenGL.
- Genera un identificador de textura con glGenTextures y carga los píxeles con glTexImage2D.
- Se configuran los filtros GL_LINEAR para suavizar la textura.

```
def load_texture(self, surface):
    tex_data = pygame.image.tostring(surface, "RGB", True)
    w, h = surface.get_size()
    self.texture = glGenTextures(1)
    glBindTexture(GL_TEXTURE_2D, self.texture)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0, GL_RGB, GL_UNSIGNED_BYTE, tex_data)
```

Función draw(self)

Renderiza visualmente un superelipsoide en una escena 3D, aplicando la textura cargada y dibujando la superficie completa mediante la parametrización matemática de una superelipse generalizada.

- **Método sin parámetros externos.** Dibuja el superelipsoide completo según la configuración actual.
- **Asocia la textura previamente cargada con la superficie que se va a renderizar.**

Define la resolución del modelo:

`stacks`: número de divisiones verticales (latitudes) del superelipsoide

`slices`: número de divisiones horizontales (longitudes) alrededor del superelipsoide

Calcula dos ángulos verticales consecutivos (`phi0` y `phi1`)

Calcula el ángulo horizontal (`theta`) representando la longitud alrededor del superelipsoide.

Utiliza la función exponencial de signo (`self.fexp`) para calcular las coordenadas X, Y, Z según la ecuación paramétrica del superelipsoide, donde:

- `self.a`, `self.b`, `self.c`: factores de escala en los tres ejes
- `self.n1`: exponente que controla la forma en sección vertical
- `self.n2`: exponente que controla la forma en sección horizontal

Asigna coordenadas de textura donde:

- `u`: posición horizontal normalizada (0 a 1) alrededor del superelipsoide
- `v`: posición vertical normalizada (0 a 1) de abajo hacia arriba

Define dónde se aplicará la textura en cada vértice y luego especifica la posición 3D del vértice.

```
def draw(self):
    glBindTexture(GL_TEXTURE_2D, self.texture)
    stacks, slices = 32, 32
    for i in range(stacks):
        phi0 = math.pi * (i / stacks - 0.5)
        phi1 = math.pi * ((i + 1) / stacks - 0.5)
        glBegin(GL_TRIANGLE_STRIP)
        for j in range(slices + 1):
            theta = 2 * math.pi * j / slices
            for phi, t in zip((phi0, phi1), (i / stacks, (i + 1) / stacks)):
                x = self.a * self.fexp(math.cos(phi), self.n1) * self.fexp(math.cos(theta), self.n2)
                y = self.b * self.fexp(math.cos(phi), self.n1) * self.fexp(math.sin(theta), self.n2)
                z = self.c * self.fexp(math.sin(phi), self.n1)
                u = j / slices
                v = t
                glTexCoord2f(u, v)
                glVertex3f(x, y, z)
        glEnd()
```

CLASE UTILIDADES:

Esta clase lo que va a hacer es gestionar la iluminación y las sombras 3D de forma modular, para cada figura.

1. **Función *init_lighting()*:** Va a configurar la luz principal (GL_LIGHT0).
    - GL_SMOOTH: Suaviza el sombreado en las superficies.
    - Posiciona la luz en coordenadas (x=4, y=4, z=6).
    - Componentes ambiental (bajo) y difusa (alto) para contraste.

```
3  def init_lighting():
4      glShadeModel(GL_SMOOTH)
5      glEnable(GL_LIGHTING)
6      glEnable(GL_LIGHT0)
7      glLightfv(GL_LIGHT0, GL_POSITION, (4.0,4.0,6.0,1.0))
8      glLightfv(GL_LIGHT0, GL_AMBIENT,  (0.2,0.2,0.2,1))
9      glLightfv(GL_LIGHT0, GL_DIFFUSE,  (0.7,0.7,0.7,1))
```

2. **Propiedades de Materiales:** Va a definir cómo los objetos reflejan la luz.
    - GL_SPECULAR: Reflejos blancos intensos.
    - GL_SHININESS=50: Controla el brillo.

```
10     glMaterialfv(GL_FRONT,GL_SPECULAR,(1,1,1,1))
11     glMaterialfv(GL_FRONT,GL_SHININESS,50)
```

3. **Segunda fuente de luz:** Esta va a añadir una luz secundaria azulada.
    - Tiene su posición simétrica a GL_LIGHT0 en el eje X.
    - Da un tono azul (0.5,0.5,0.8) para efectos de contraste.

```
12     glEnable(GL_LIGHT1)
13     glLightfv(GL_LIGHT1, GL_POSITION, (-4.0, 4.0, 6.0, 1.0))
14     glLightfv(GL_LIGHT1, GL_DIFFUSE,  (0.5, 0.5, 0.8, 1))
15     glLightfv(GL_LIGHT1, GL_AMBIENT,  (0.1, 0.1, 0.1, 1))
```

4. **Función rotate_arbitrary_axis(angle, axis_x, axis_y, axis_z):** Se realiza una rotación alrededor de un eje arbitrario.
    - Parámetros:
        - angle: Ángulo de rotación.
        - (axis_x, axis_y, axis_z): Vector que define el eje.
    - Internamente usa *glRotatef*.

```
17    def rotate_arbitrary_axis(angle, axis_x, axis_y, axis_z):
18        glRotatef(angle, axis_x, axis_y, axis_z)
```

5. **Función draw_shadow(shape, light_pos, ground_level=-3.0):** Dibuja la sombra de un objeto en un plano (por defecto en y=-3.0).
    - Preparación:
        - Desactiva la iluminación (GL_LIGHTING).
        - Desactiva texturas (GL_TEXTURE_2D).
        - Establece el color de la sombra (gris oscuro semitransparente).

```
20    def draw_shadow(shape, light_pos, ground_level=-3.0):
21        glDisable(GL_LIGHTING)
22        glDisable(GL_TEXTURE_2D)
23        glColor4f(0.1, 0.1, 0.1, 0.5)
```

6. **Matriz de Proyección de Sombras:**
    - Crea una matriz de transformación 4x4 para proyectar la sombra.
    - La sombra se proyecta según la posición de la luz (light_pos).
    - Usa proyección plana (shadow projection matrix).

```
25        shadow_mat = [0]*16
26        dot = light_pos[1]  # y * 1
27
28        for i in range(4):
29            for j in range(4):
30                if i == j:
31                    shadow_mat[i*4+j] = dot - light_pos[j] if i == 1 else dot
32                else:
33                    shadow_mat[i*4+j] = -light_pos[j] if i == 1 else 0
```

7. **Dibujado de la Sombra:**
    - Guarda la matriz actual (glPushMatrix).
    - Posiciona la sombra en el nivel del suelo (ground_level).
    - Aplica la matriz de sombra.
    - Dibuja la forma usando el método draw() del objeto pasado.
    - Restaura la matriz (glPopMatrix).

```
35      glPushMatrix()
36      glTranslatef(0, ground_level, 0)
37      glMultMatrixf(shadow_mat)
38
39      shape.draw()
40
41      glPopMatrix()
```

8. **Restauración:**
    - Reactiva las texturas y la iluminación.
    - Restablece el color blanco.

```
42      glEnable(GL_TEXTURE_2D)
43      glEnable(GL_LIGHTING)
44      glColor4f(1, 1, 1, 1)
```

CLASE MAIN:

Esta clase gestiona toda la lógica de visualización, interacción y control de la escena.

1. **Configuración Inicial:**
    - ***Inicialización:***
        - Crea una ventana de 800x600 píxeles.
        - Se establece el título de la ventana.
        - Se configura el color de fondo (gris oscuro, para que se note la iluminación).
        - Habilita las características para la profundidad, texturas 2D y blending.

```
16  def main():
17      pygame.init()
18      display = (800, 600)
19      pygame.display.set_mode(display, DOUBLEBUF | OPENGL)
20      pygame.display.set_caption("Práctica 5 - Escena 3D Realista")
21
22      glClearColor(0.1, 0.1, 0.1, 1)
23      glEnable(GL_DEPTH_TEST)
24      glEnable(GL_TEXTURE_2D)
25      glEnable(GL_BLEND)
```

    - ***Iluminación:***
        - Llama a init_lighting() para configurar el sistema de luces.

```
28      init_lighting()
```

    - ***Carga de Texturas:***
        - Busca y carga la textura 'textura.jpg' desde el directorio del script.
        - Si no encuentra la textura, muestra error y sale del programa.

```
30      script_dir = os.path.dirname(os.path.abspath(__file__))
31      tex_path = os.path.join(script_dir, 'textura.jpg')
32      if not os.path.exists(tex_path):
33          print(f"Error: 'textura.jpg' no encontrado en {script_dir}")
34          sys.exit(1)
35      tex_surf = pygame.image.load(tex_path)
```

2. **Carga de los recursos:**
    - ***Lista de las formas:***
        - Crea instancias de todas las figuras geométricas.
        - Carga la textura en cada figura que tenga el método load_texture().

```
37      shapes = [Cubo(), Piramide(), Esfera(), Cilindro(), Superelipsoide()]
38      for s in shapes:
39          if hasattr(s, 'load_texture'):
40              s.load_texture(tex_surf)
```

- ***Sistema de Transformaciones:*** Va a almacenar todas las transformaciones aplicables.
    - global: Rotaciones respecto al mundo.
    - local: Rotaciones respecto al objeto.

```
42      projection = 'perspective'
43      selected = None
44      transforms = {
45          'rot': [0, 0],
46          'trans': [0, 0, 0],
47          'scale': 1.0,
48          'arb_rot': {'angle': 0, 'axis': [1, 1, 1]},
49          'rotation_mode': 'global'
50      }
```

3. **Función set_projection():** Configura la matriz de proyección según el modo actual.
    - Perspectiva: gluPerspective(45°, relación aspecto, near=0.1, far=50)
    - Ortográfica: glOrtho(-5,5, -5,5, -50,50)

```
62      def set_projection():
63          glMatrixMode(GL_PROJECTION)
64          glLoadIdentity()
65          if projection == 'perspective':
66              gluPerspective(45, display[0]/display[1], 0.1, 50)
67          else:
68              glOrtho(-5, 5, -5, 5, -50, 50)
69          glMatrixMode(GL_MODELVIEW)
```

4. **Bucle principal:** Gestiona los eventos, actualiza transformaciones y renderiza la escena.

   Manejo de Eventos:

   - Selección de figuras: Teclas 1-5
   - Resetear transformaciones: Tecla R
   - Cambiar proyección: Tecla P
   - Alternar iluminación: Tecla I
   - Alternar texturas: Tecla T
   - Mostrar sombras: Tecla H
   - Cambiar modo rotación: Tecla M (global/local)
   - Escala: Teclas +/-
   - Rotación eje específico: Teclas 7 (X), 8 (Y), 9 (Z)
   - Movimiento continuo: Flechas (rotación), WASDQE (traslación XYZ)

```
74      while True:
75          for event in pygame.event.get():
76              if event.type == pygame.QUIT:
77                  pygame.quit()
78                  sys.exit()
79              elif event.type == KEYDOWN:
80                  if event.key == K_1: selected = 0
81                  elif event.key == K_2: selected = 1
82                  elif event.key == K_3: selected = 2
83                  elif event.key == K_4: selected = 3
84                  elif event.key == K_5: selected = 4
85                  elif event.key == K_6:
86                      pygame.quit()
87                      sys.exit()
88                  elif event.key == K_r:
89                      transforms = {
90                          'rot': [0, 0],
91                          'trans': [0, 0, 0],
92                          'scale': 1.0,
93                          'arb_rot': {'angle': 0, 'axis': [1, 1, 1]},
94                          'rotation_mode': transforms['rotation_mode']
95                      }
```

```
96                 elif event.key == K_p:
97                     projection = 'perspective' if projection == 'ortho' else 'ortho'
98                     set_projection()
99                 elif event.key == K_i:
100                    lighting = not lighting
101                    if lighting:
102                        glEnable(GL_LIGHTING)
103                        glEnable(GL_LIGHT0)
104                        glEnable(GL_LIGHT1)
105                    else:
106                        glDisable(GL_LIGHTING)
107                elif event.key == K_t:
108                    textured = not textured
109                    if textured:
110                        glEnable(GL_TEXTURE_2D)
111                    else:
112                        glDisable(GL_TEXTURE_2D)
113                elif event.key == K_h:
114                    show_shadow = not show_shadow
115                elif event.key == K_m:
116                    transforms['rotation_mode'] = 'local' if transforms['rotation_mode'] == 'global' else 'global'
117                elif event.unicode == '+':
118                    transforms['scale'] *= 1.05
119                elif event.unicode == '-':
120                    transforms['scale'] /= 1.05
121                elif event.key == K_7:
122                    transforms['arb_rot']['axis'] = [1, 0, 0]
123                    transforms['arb_rot']['angle'] += 5
124                elif event.key == K_8:
125                    transforms['arb_rot']['axis'] = [0, 1, 0]
126                    transforms['arb_rot']['angle'] += 5
127                elif event.key == K_9:
128                    transforms['arb_rot']['axis'] = [0, 0, 1]
129                    transforms['arb_rot']['angle'] += 5
130                elif event.key == K_ESCAPE:
131                    selected = None
132                    transforms = {
133                        'rot': [0, 0],
134                        'trans': [0, 0, 0],
135                        'scale': 1.0,
136                        'arb_rot': {'angle': 0, 'axis': [1, 1, 1]},
137                        'rotation_mode': 'global'
138                    }
```

```
140                keys = pygame.key.get_pressed()
141                if keys[K_LEFT]: transforms['rot'][1] -= 1
142                if keys[K_RIGHT]: transforms['rot'][1] += 1
143                if keys[K_UP]: transforms['rot'][0] -= 1
144                if keys[K_DOWN]: transforms['rot'][0] += 1
145                if keys[K_w]: transforms['trans'][1] += 0.1
146                if keys[K_s]: transforms['trans'][1] -= 0.1
147                if keys[K_a]: transforms['trans'][0] -= 0.1
148                if keys[K_d]: transforms['trans'][0] += 0.1
149                if keys[K_q]: transforms['trans'][2] -= 0.1
150                if keys[K_e]: transforms['trans'][2] += 0.1
```

Renderizado:

- Modo Menú (cuando selected is None):
    - Dibuja el menú de selección.
    - Configura una proyección ortográfica 2D temporal.

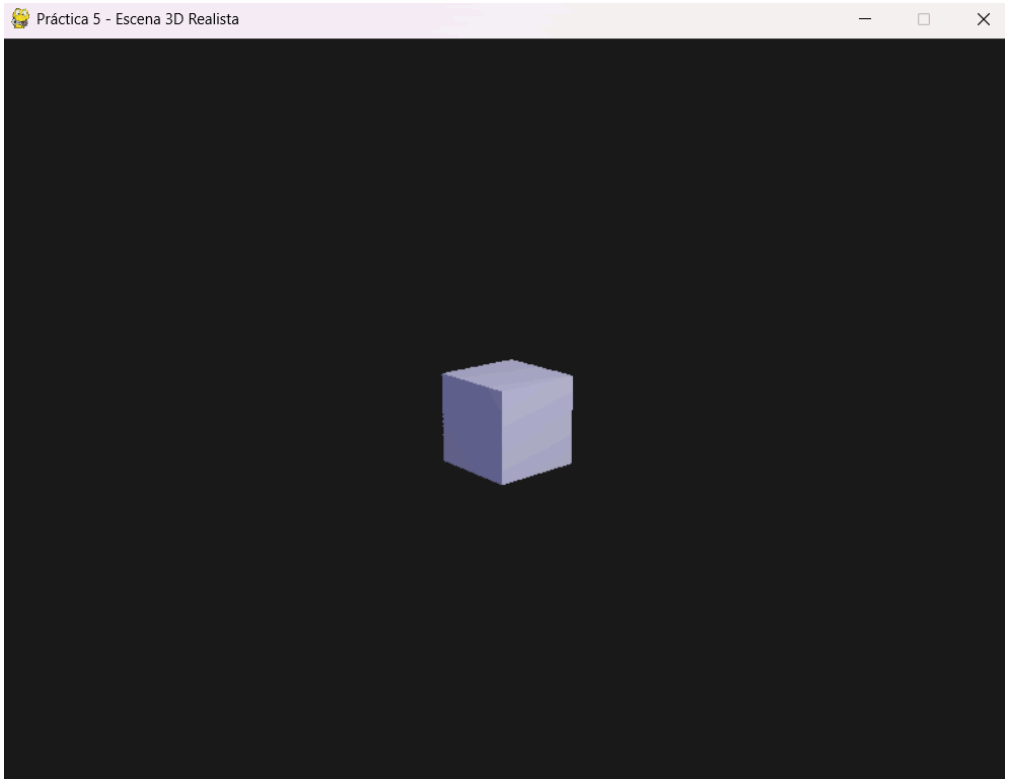
```
157         if selected is None:
158             glMatrixMode(GL_PROJECTION); glPushMatrix(); glLoadIdentity()
159             glOrtho(0, display[0], 0, display[1], -1, 1)
160             glMatrixMode(GL_MODELVIEW); glPushMatrix(); glLoadIdentity()
161             glDisable(GL_TEXTURE_2D)
162             for i, line in enumerate(menu):
163                 if not line: continue
164                 surf = font.render(line, True, (255, 255, 255)).convert_alpha()
165                 x = (display[0] - surf.get_width()) // 2
166                 y = display[1] - 110 - i * 35
167                 glRasterPos2f(x, y)
168                 data = pygame.image.tostring(surf, "RGBA", True)
169                 glDrawPixels(surf.get_width(), surf.get_height(), GL_RGBA, GL_UNSIGNED_BYTE, data)
```

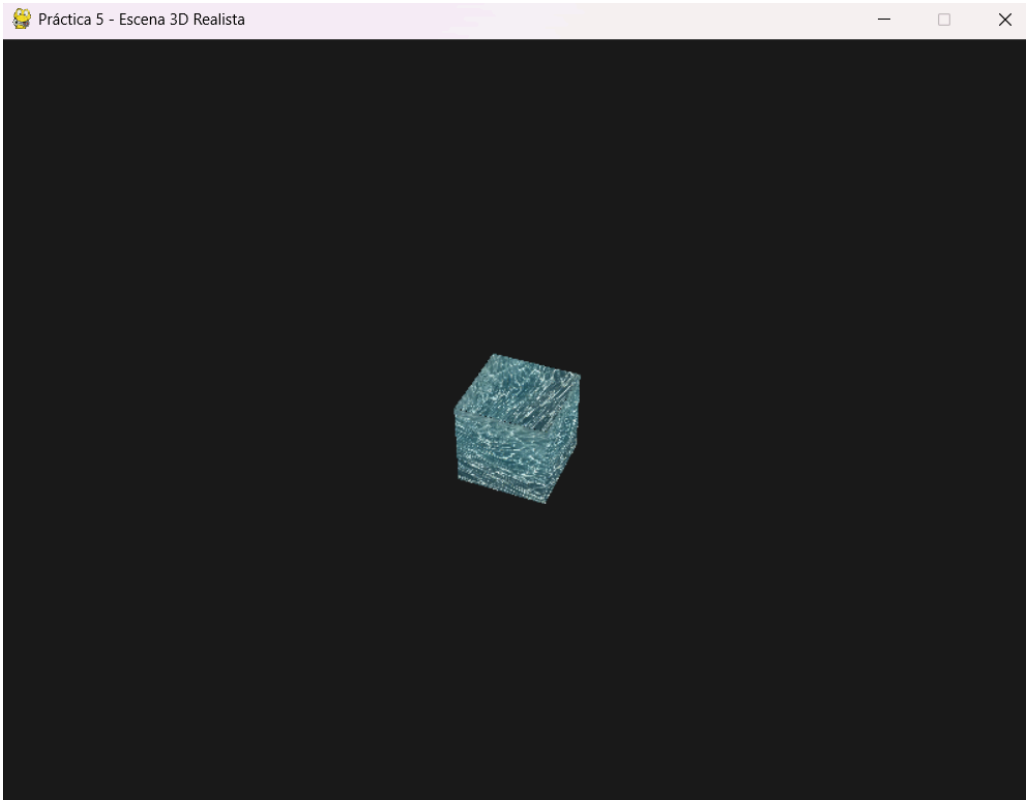
- **Modo Visualización 3D:**
    - Aplica todas las transformaciones acumuladas.
    - Gestiona el orden de rotaciones según el modo (global/local).
    - Dibuja la sombra si está activado (show_shadow).
    - Renderiza la figura seleccionada llamando a su método draw().

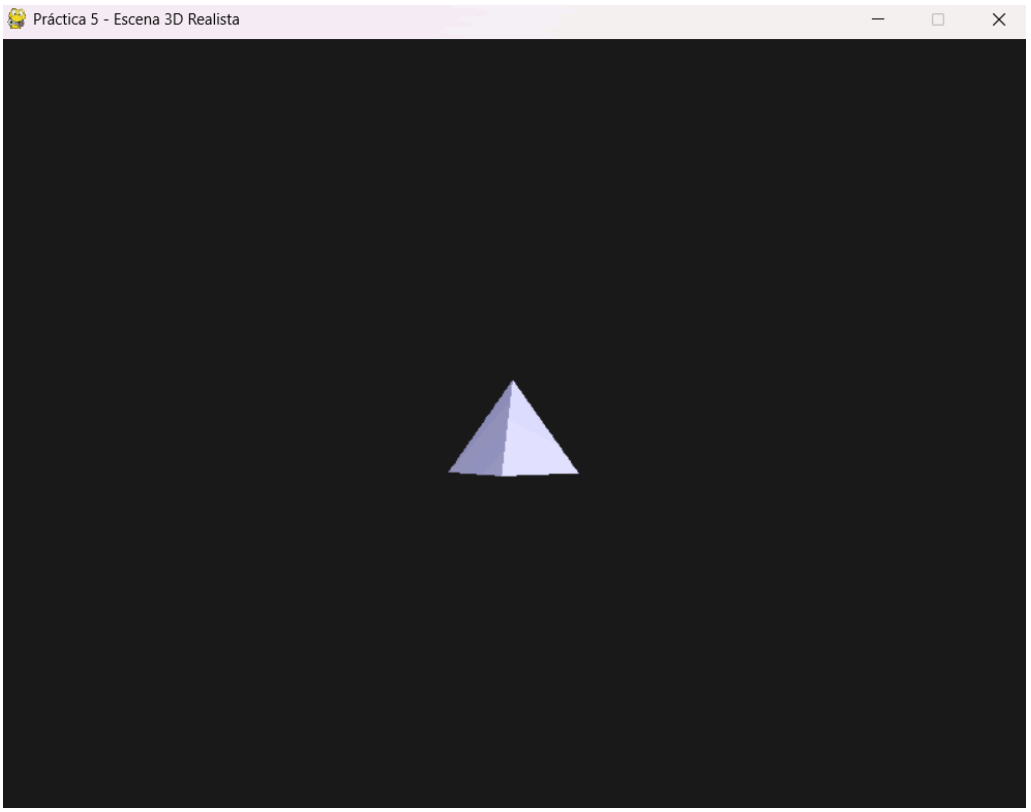
```
170             glPopMatrix()
171             glMatrixMode(GL_PROJECTION); glPopMatrix()
172             glMatrixMode(GL_MODELVIEW)
173             glEnable(GL_TEXTURE_2D)
174         else:
175             glTranslatef(*transforms['trans'])
176             glScalef(transforms['scale'], transforms['scale'], transforms['scale'])
177
178             if transforms['rotation_mode'] == 'global':
179                 glRotatef(transforms['rot'][0], 1, 0, 0)
180                 glRotatef(transforms['rot'][1], 0, 1, 0)
181                 rotate_arbitrary_axis(
182                     transforms['arb_rot']['angle'],
183                     *transforms['arb_rot']['axis']
184                 )
185             else:
186                 rotate_arbitrary_axis(
187                     transforms['arb_rot']['angle'],
188                     *transforms['arb_rot']['axis']
189                 )
190                 glRotatef(transforms['rot'][1], 0, 1, 0)
191                 glRotatef(transforms['rot'][0], 1, 0, 0)
192
193             if show_shadow:
194                 draw_shadow(shapes[selected], light_pos)
195
196             shapes[selected].draw()
197
198         pygame.display.flip()
199         clock.tick(60)
200
201 if __name__ == "__main__":
202     main()
```

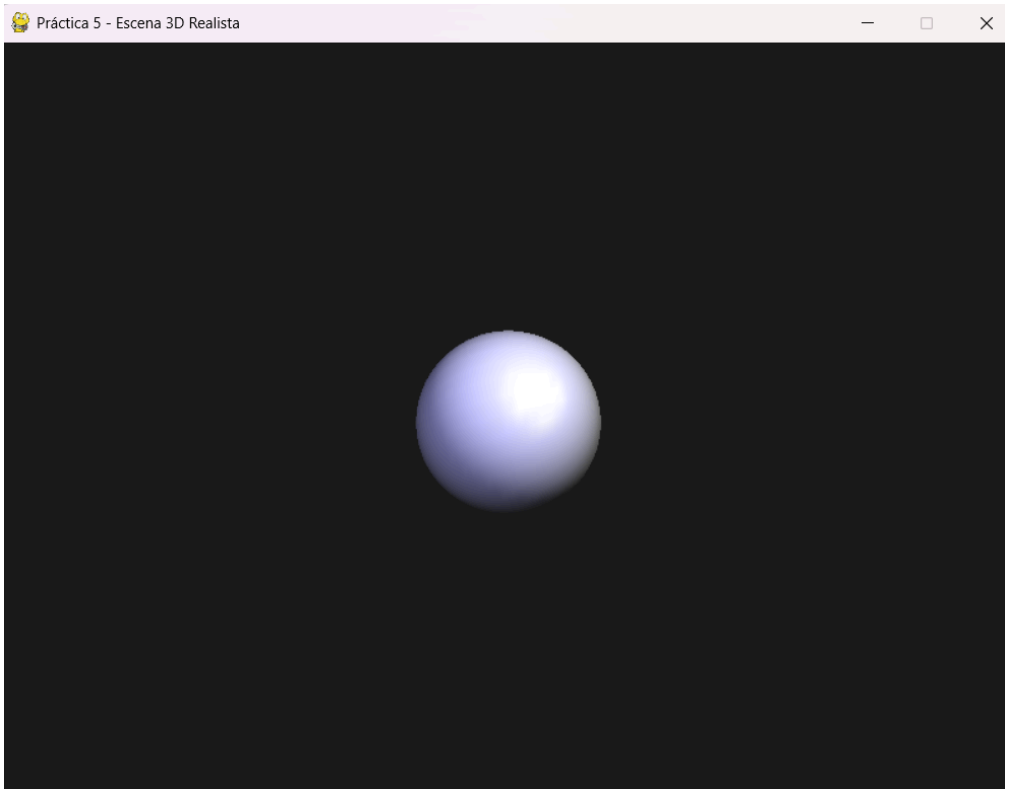
RESULTADO:

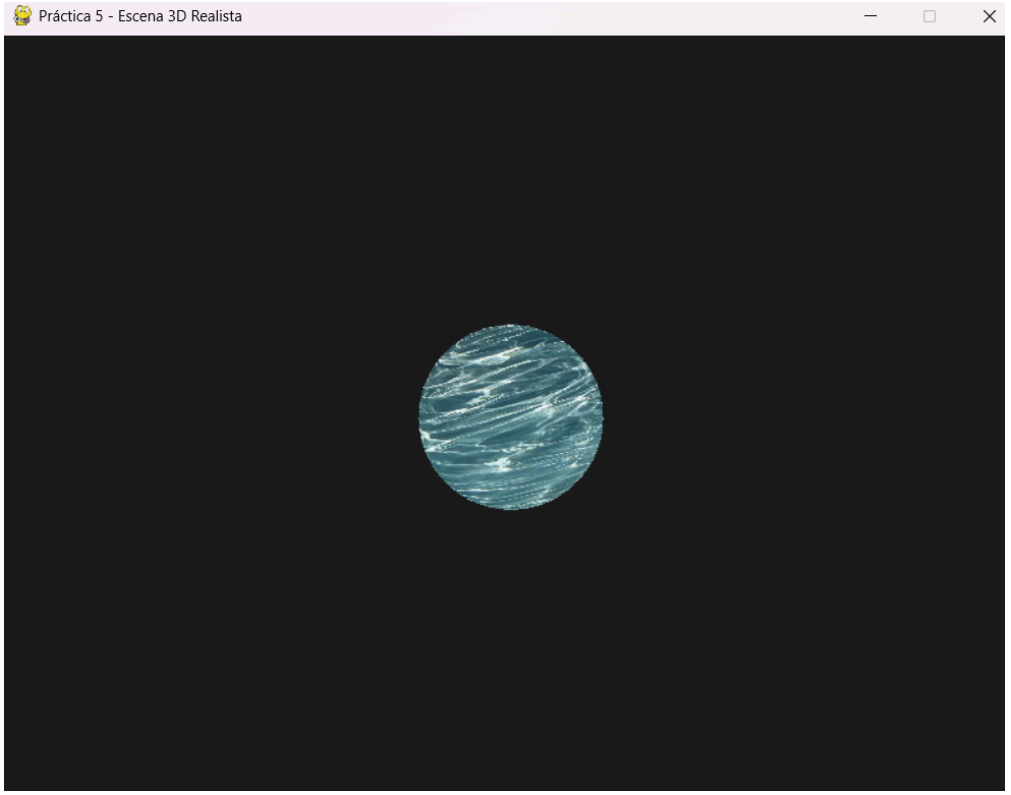

Práctica 5 - Escena 3D Realista
MENÚ
1) Mostrar Cubo
2) Mostrar Pirámide
3) Mostrar Esfera
4) Mostrar Cilindro
5) Mostrar Superelipsoide
6) Salir

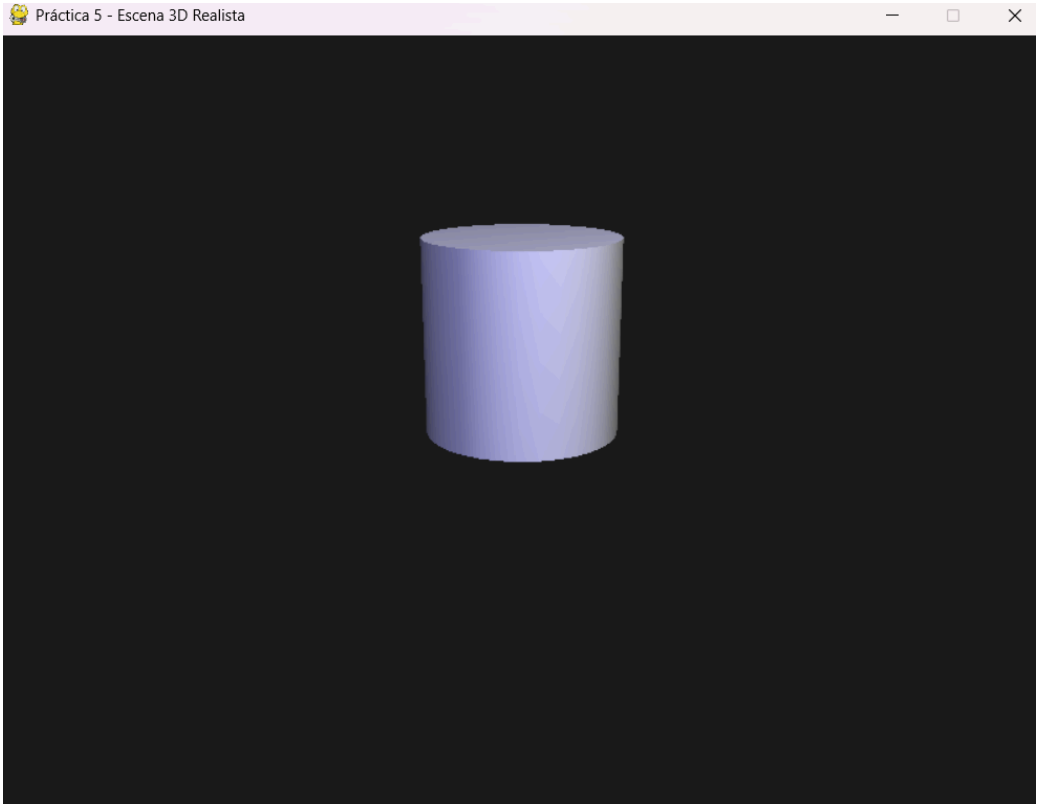

Práctica 5 - Escena 3D Realista

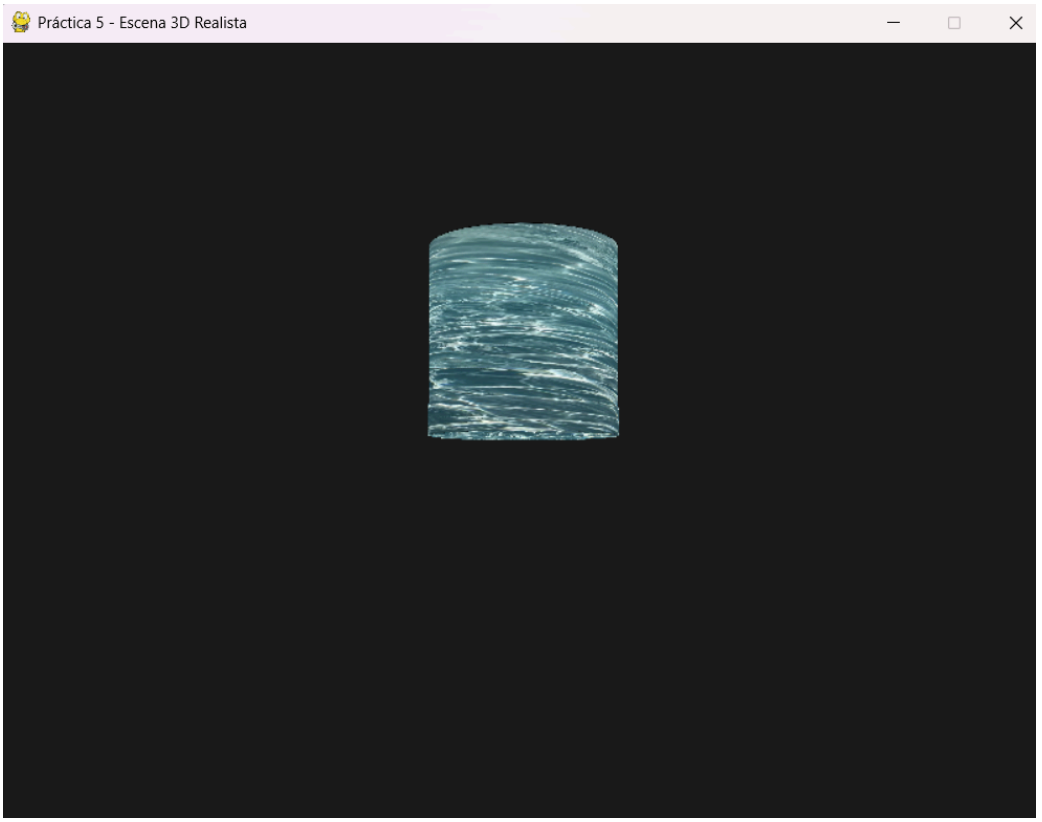

Práctica 5 - Escena 3D Realista

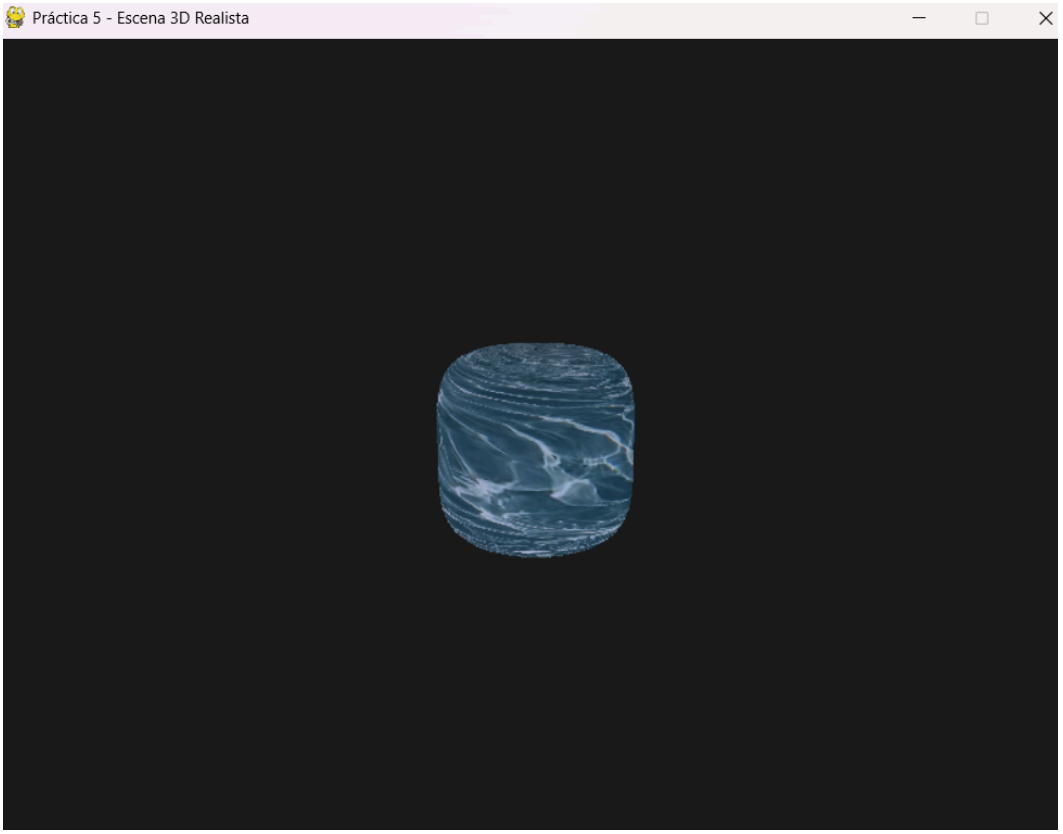

Práctica 5 - Escena 3D Realista


Práctica 5 - Escena 3D Realista


Práctica 5 - Escena 3D Realista


Práctica 5 - Escena 3D Realista


Práctica 5 - Escena 3D Realista



ENLACE DEL VÍDEO:

https://drive.google.com/file/d/1M6Y3p2bvbTnE44YLif-pUsEfLHGru78e/view?usp=sharing

CONCLUSIÓN:

La implementación del proyecto fue un desafío importante, ya que permitió la integración de múltiples componentes de gráficos 3D en un sistema interactivo. La experiencia se destacó en la necesidad de coordinar transformaciones geométricas, gestión de texturas, iluminación dinámica y proyección visual manteniendo un rendimiento fluido. El uso de OpenGL con PyGame ha demostrado ser una combinación poderosa para el renderizado, aunque requiere una sincronización cuidadosa entre el bucle de eventos de PyGame y las operaciones de renderizado de OpenGL. Modularizar el código en clases separadas para cada forma (cubo, pirámide, esfera, etc.) facilita el trabajo en equipo, permitiendo que cada componente se desarrolle y pruebe de forma independiente antes de integrarse en el sistema principal.

Los efectos visuales se logran mediante el uso de múltiples fuentes de luz (GL_LIGHT0 y GL_LIGHT1) para lograr una iluminación dinámica, incluida la luz ambiental, la reflexión difusa y los componentes de reflexión especular para simular materiales reflectantes. Habilitar GL_SMOOTH garantiza un sombreado suave en las superficies, mientras que el mapeo de texturas (cargando desde una imagen JPEG y aplicando coordenadas UV) agrega detalles realistas a los objetos. El sistema de sombra proyectada, mejora la percepción de la profundidad al simular la interacción entre la luz y los objetos en el plano del suelo. Incluir el búfer Z (GL_DEPTH_TEST) es fundamental para manejar correctamente la oclusión entre objetos y evitar artefactos visuales.

Finalmente, la implementación traslación, escala, rotación y efectos adicionales, mejora la experiencia del usuario, mostrando un equilibrio entre funcionalidad y complejidad. El proyecto no solo logró sus objetivos originales, sino que también sirvió como un excelente ejercicio para comprender los fundamentos de los gráficos 3D, desde la teoría matemática hasta la implementación práctica, enfatizando la importancia de la optimización, el trabajo en equipo y la atención al detalle para lograr resultados visualmente atractivos y sólidos.