Capítulo 9

Trabajar en 3D con Pygame

1. Introducción

Los juegos desarrollados durante este libro son exclusivamente en dos dimensiones (2D). En la práctica, Pygame está más asociado con el desarrollo de juegos 2D, aunque permite la visualización tridimensional.

Hay varios videojuegos que son tridimensionales o utilizan gráficos que hacen que se acerquen a las tres dimensiones. En particular, podemos mencionar los llamados juegos FPS (*First Person Shooter*, videojuego de disparos con vista en primera persona, en español). Este tipo de juego suele consistir en la encarnación de un personaje con un arma de fuego dentro de un 3D inmersivo. El objetivo es derribar a los enemigos que se va encontrando. En general, el personaje principal se mueve a través de una red de pasillos dibujados en tres dimensiones.

Debido a su base SDL, Pygame es una herramienta muy orientada a las 2D. Todos los juegos que se han puesto como ejemplos en este libro, son juegos bidimensionales. Entonces, ¿qué necesitarían para ser juegos en 3D? La respuesta puede parecer obvia: una decoración en 3D y una visualización en 3D. En otras palabras, su algoritmo sigue siendo el mismo: la lógica y el bucle del juego en sí, se supone que no necesitan modificaciones excesivas. Pero queda por ver cómo hacer 3D con Pygame. La respuesta se puede resumir en una palabra: OpenGL.

2. La librería 3D OpenGL

OpenGL (*Open Graphics Library*) es una librería que permite hacer 3D. Se utiliza en muchos dominios más allá de los videojuegos: simulación científica, CAO, realidad aumentada, etc. Su objetivo es permitir crear objetos 3D a los que se puedan asociar texturas. OpenGL se encarga de la visualización de estos objetos 3D teniendo en cuenta la orientación, la distancia y los aspectos relativos a la luz, como la sombra proyectada, la transparencia, etc.

Desde luego, el objetivo aquí no es enseñarle OpenGL. De hecho, OpenGL es un mundo amplio y complejo y se necesitaría mucho más que estas pocas páginas para aprender a programar de forma autónoma con esta herramienta. El objetivo es acompañarle en sus primeros pasos en 3D en un entorno Pygame.

3. OpenGL en Python/Pygame

3.1 PyOpenGL

Hay un módulo de Python que permite trabajar con OpenGL: `PyOpenGL`. Si no está instalado en su máquina, lanzar el comando `pip` en el terminal le permite hacerlo.

```
pip install PyOpenGL
```

Para usarlo en código Python, simplemente importe el módulo como se hace normalmente (para Pygame, por ejemplo). En general, procedemos a importar escribiendo estas dos líneas:

```
from OpenGL.GL import *
from OpenGL.GLU import *
```

3.2 Las nociones fundamentales: vértice y arista

Para definir objetos 3D en OpenGL, utilizaremos dos conceptos fundamentales: vértice (*vertex* en inglés) y arista (*edge* en inglés).

Esquemáticamente, podemos considerar que los vértices son puntos en el espacio y, por lo tanto, tienen sus coordenadas en tres dimensiones. Por ejemplo: (1, 2, -4). Las aristas son los segmentos que conectan los vértices.

De esta manera, si definimos dos vértices:

- vértice 1 situado en las coordenadas (1, 0, 0),
- vértice 2 situado en las coordenadas (0, -1, 0),
- entonces podemos definir la arista 1 que conecta el vértice 1 con el vértice 2.

Esto es equivalente a pensar que definir los objetos 3D en OpenGL es, ante todo, definir una colección de puntos tridimensionales (vértices) y luego definir las aristas que conectan estos vértices entre sí.

3.3 PyOpenGL y Pygame

Cuando creamos la ventana de juego Pygame, especificamos un modo de funcionamiento relativo a OpenGL. El valor utilizado es: `pygame.DOUBLEBUF | pygame.OPENGL`.

Por ejemplo, tenemos este tipo de código:

```
import pygame
pygame.init()
ZONA = (800,800)
pygame.display.set_mode(ZONA, pygame.DOUBLEBUF|pygame.OPENGL)
```

4. PyOpenGL/Pygame: el ejemplo del cubo

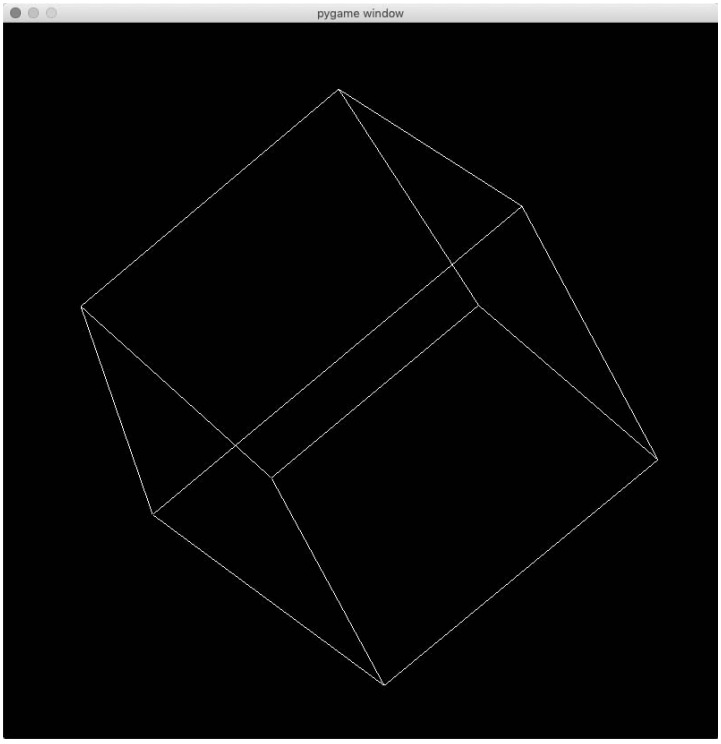
Tomemos el ejemplo clásico de un cubo representado en tres dimensiones, que sometemos a una rotación continua. La idea es mostrar solo las aristas del cubo; no definimos ninguna textura o coloración de las caras.

4.1 El código global

Comencemos presentando el código global del ejemplo, antes de proceder a la explicación detallada del programa.

```
import pygame
from OpenGL.GL import *
from OpenGL.GLU import *

VERTICES = (
    (1, -1, -1),
    (1, 1, -1),
    (-1, 1, -1),
    (-1, -1, -1),
    (1, -1, 1),
    (1, 1, 1),
    (-1, -1, 1),
    (-1, 1, 1)
```

```
    )

ARISTAS = (
    (0,1),
    (0,3),
    (0,4),
    (2,1),
    (2,3),
    (2,7),
    (6,3),
    (6,4),
    (6,7),
    (5,1),
    (5,4),
    (5,7)
    )


def Cubo():
    glBegin(GL_LINES)
    for arista in ARISTAS:
        for vertice in arista:
            glVertex3fv(VERTICES[vertice])
    glEnd()

pygame.init()
ZONA = (800,800)
pygame.display.set_mode(ZONA, pygame.DOUBLEBUF|pygame.OPENGL)

gluPerspective(45, (ZONA[0]/ZONA[1]), 0.1, 50.0)
glTranslatef(0.0,0.0, -5)

while True:
 for event in pygame.event.get():
   if event.type == pygame.QUIT:
     pygame.quit()
     quit()

 glRotatef(1, 3, 1, 3)
 glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
 Cubo()
 pygame.display.flip()
 pygame.time.wait(10)
```



Captura de pantalla de la ventana del cubo 3D en rotación

4.2 Explicación detallada del código

Empezamos importando Pygame y lo que se necesita para usar OpenGL.

```
import pygame
from OpenGL.GL import *
from OpenGL.GLU import *
```

Defina una lista de vértices.

```
VERTICES = (
 (1, -1, -1),
 (1, 1, -1),
 (-1, 1, -1),
 (-1, -1, -1),
 (1, -1, 1),
 (1, 1, 1),
 (-1, -1, 1),
 (-1, 1, 1)
)
```

Por lo tanto, el cubo tiene dos unidades en cada lado y se define entre -1 y +1 en cada uno de los tres ejes.

Los ocho vértices participan en la definición del cubo:

- Vértice 0: (1, -1, -1)
- Vértice 1: (1, 1, -1)
- Vértice 2: (-1, 1, -1)
- Vértice 3: (-1, -1, -1)
- Vértice 4: (1, -1, 1)
- Vértice 5: (1, 1, 1)
- Vértice 6: (-1, -1, 1)
- Vértice 7: (-1, 1, 1)

A continuación, defina los bordes del cubo.

```
ARISTAS = (
 (0,1),
 (0,3),
 (0,4),
 (2,1),
 (2,3),
 (2,7),
 (6,3),
 (6,4),
 (6,7),
 (5,1),
 (5,4),
 (5,7)
)
```

por ejemplo, `(2,7)` significa "la arista entre los vértices 2 y 7".

A continuación, cree la función que genera el cubo. Para ello, utilice las siguientes funciones OpenGL:

- `glBegin` se utiliza para declarar el inicio del procesamiento. La función recibe como argumento el tipo de elemento geométrico dibujado, aquí `GL_LINES` lo que significa que dibujamos líneas.
- `glEnd` se utiliza para indicar el final de este procesamiento específico.
- `glVertex3fv` permite dibujar aristas.

```
def Cubo():
 glBegin(GL_LINES)
 for arista in ARISTAS:
   for vertice in arista:
     glVertex3fv(VERTICES[vertice])
 glEnd()
```

Además de las líneas (*GL_LINES*), el resto de elementos geométricos que puede dibujar con OpenGL son:

`GL_POINTS`: para dibujar puntos (vértices no conectados).

`GL_LINES`: para dibujar líneas (que es lo que usamos aquí).

`GL_LINE_STRIP`: para dibujar líneas conectadas (de un paso al siguiente).

`GL_LINE_LOOP`: para dibujar líneas conectadas, pero que finalmente forman un bucle.

`GL_TRIANGLES`: para dibujar triángulos.

`GL_QUADS`: para dibujar cuadriláteros.

`GL_POLYGON`: para dibujar polígonos convexos.

Ahora podemos inicializar Pygame y definir una ventana de juego dedicada a OpenGL.

```
pygame.init()
ZONA = (800,800)
pygame.display.set_mode(ZONA, pygame.DOUBLEBUF|pygame.OPENGL)
```

Después, manejamos la perspectiva, es decir, la característica que define cómo el observador verá el sujeto. Usamos la función OpenGL `gluPerspective` que recibe cuatro argumentos:

- El ángulo del campo de visión (en grados).
- El marco de visualización.
- La distancia desde el plano del sujeto más cercano.
- La distancia desde el plano del sujeto más lejano.

```
(gluPerspective(45, (ZONA[0]/ZONA[1]), 0.1, 50.0)
```

Seguidamente hacemos una ligera traslación a lo largo del eje z para obtener una mayor comodidad. De hecho, permite alejar un poco el cubo. De lo contrario estaríamos demasiado cerca de él.

```
GlTranslatef(0.0,0.0, -5)
```

Luego inicie el bucle del juego.

```
while True:
```

Como de costumbre, ocúpese de la acción de abandonar el programa.

```
for event in pygame.event.get():
 if event.type == pygame.QUIT:
   pygame.quit()
   quit()
```