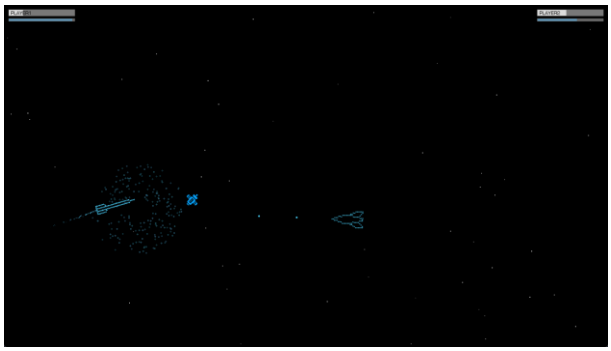
SPYCEWAR!

Daniel Marín Buj, 17 de julio de 2024

https://github.com/dmar1n/spycewar



Descripción

Spycewar es un juego inspirado en el clásico *Spacewar!*, probablemente el primer videojuego de la historia. Fue diseñado en 1961 por Martin Graetz, Stephen Russell, y Wayne Wiitanen, e implementado para la computadora PDP-1 en 1962.

En el juego clásico, dos jugadores combaten entre sí; cada uno maneja una nave que ha de navegar en torno a una estrella central con atracción gravitatoria. Cada jugador tiene la posibilidad de disparar misiles, rotar y propulsar la nave, así como usar un botón especial de salto al hiperespacio.

En **Spycewar**, se ha implementado una mecánica similar, con un sistema de propulsión, disparo de proyectiles y salto al hiperespacio (la nave cambia instantáneamente de posición a otro punto aleatorio de la pantalla). Las principales diferencias son la ausencia de estrella central, la adición de un escudo temporal, barras de vida y escudo, y la aparición aleatoria de una nave de suministro que restaura una parte de vida de la nave.

El salto al hiperespacio permite, como en el juego original, cambiar la posición instantáneamente. Otra estrategia defensiva es el escudo que, cuando se activa, hace que la nave sea inmune a los misiles; sin embargo, el escudo se consume y no se puede recuperar (segunda barra de vida).

Por último, cada nave tiene características diferentes (velocidad, rotación, vida, cadencia de disparo, etc.), facilitando así diferentes estrategias en función de la nave elegida

Implementación

La implementación está basada en polimorfismo de clases, herencia y el patrón de estado, así como siguiendo el principio de composición. Adicionalmente, se ha utilizado el sistema de eventos propio de Pygame para mantener el código desacoplado en la medida de lo posible. En el anexo final, se muestran los esquemas para jugadores y naves, donde se puede ver que se sigue tanto el principio de herencia como el de composición.

Optimizaciones

Para mejorar la eficiencia y el rendimiento del juego, se han implementado las siguientes optimizaciones.

Imágenes

Las imágenes de las naves, los proyectiles y la nave de suministros se cargan una sola vez en memoria. Adicionalmente, se cachean las imágenes rotadas de las naves, de manera que, si la nave se mueve o permanece estática, pero no rota, se reutiliza la imagen ya rotada.

```
if self.__last_angle != self.__ship_state.angle:
    self.__rotated_image = pygame.transform.rotate(self.image, self.__ship_state.angle)
    self.__last_angle = self.__ship_state.angle
```

Igualmente, los proyectiles que alcanzan el límite de la pantalla se borran de la memoria.

Colisiones

Para las colisiones, primero se detecta la colisión entre rectángulos (más eficiente). Si se da la colisión entre rectángulos, entonces se detecta la colisión entre máscaras (menos eficiente, pero más precisa). Este principio se sigue para todas las colisiones del juego (naves vs naves, naves vs proyectiles, y naves vs suministro). Por ejemplo:

```
for player in groupcollide(self.__players, self.__projectiles, False, False, collide_rect):
    if results := groupcollide(self.__players, self.__projectiles, False, True, collide_mask):
```

Dificultades técnicas

Rotación y aceleración de las naves

La rotación, aceleración y momento de cada nave supone la implementación de físicas más complejas que las de un juego tipo *Space Invaders*, en el que los objetos de juego se mueven principalmente en dos ejes. En este caso, para la rotación, es necesario calcular el ángulo en el que se ejerce la fuerza de empuje, así como el vector de movimiento, que depende de si se acelera o no. Al no tener ninguna fuerza de rozamiento, las naves siempre están a la derivan hasta que colisionan o se aplica una fuerza de propulsión (las naves tienen una velocidad máxima que no les permite acelerar de manera indefinida). Asimismo, se ha de tener en cuenta que el movimiento de los proyectiles ha de sumarse al movimiento de la nave.

Las naves y los proyectiles tienen comportamientos diferentes en los límites de la pantalla: Los proyectiles se eliminan al alcanzar el límite, mientras que las naves aparecen en el lado contrario.

Función para calcular el movimiento de naves y proyectiles:

```
def __compute_trajectory(self, speed: float, backwards: bool = False, offset: int = 10) -> tuple[Vector2, Vector2]:
    """Computes the trajectory of the object based on the player's position and angle.

    Args:
        backwards: a boolean indicating whether the object should go backwards.
        offset: the offset from the player's position to spawn the object.

    Returns:
        The velocity and position of the object.
    """
    angle_radians = math.radians(self.__ship_state.angle)
    direction_vector = -Vector2(math.sin(angle_radians), math.cos(angle_radians))
    direction_vector = -direction_vector if backwards else direction_vector
    velocity = self.__ship_state.velocity + direction_vector * speed
    position = direction_vector * (self.image.get_height() // 2 + offset) + self._position
    return velocity, position
```

Sistema de partículas

Para ilustrar tanto las colisiones como la propulsión de las naves se ha implementado un sistema de partículas. Cada partícula es un objeto de juego que admite diferentes parámetros en función de la necesidad (explosiones o propulsión):

```python
class Particle(Sprite):
    """Particle class to represent an explosion particle in the game."""

    __alpha = 255

    def __init__(self, groups: Group, position: Vector2, direction: Vector2, radius: int, fade: float = 0.1) -> None:
        super().__init__(groups)
        self.__position = position.xy
        self.__direction = direction
        self.__radius = radius
        self.__fade_rate = fade
        self.__create_surface()
```

Encapsulación y polimorfismo de las naves

Para hacer el código modular, se han separado las especificaciones de cada nave (con características como la aceleración, la velocidad de rotación, el tiempo de *cooldown*...) en su propia clase. Se ha procedido igualmente para el estado de la nave, incluyendo la vida restante y el comportamiento relacionado con quitar daño o restaurar puntos de vida. Esto permite instanciar a un jugador modelo que acepta un estado y unas especificaciones (relación *has a*). De esta manera, se pueden instanciar tantos jugadores diferentes como se desee si tener que implementar cada jugador por separado.

```python
class ShipSpecs:
    """Represents the specifications of a ship.

    Attributes:
        player: the player id of the ship.
        image: the image of the ship.
        max_speed: the maximum speed of the ship.
        acceleration: the acceleration of the ship.
        rotation_speed: the rotation speed of the ship.
        projectile_speed: the speed of the projectile fired by the ship.
        projectile_cooldown: the cooldown between shots.
        hyperspace_cooldown: the cooldown between teleports.
    """
```

Uso compartido de datos entre estados

Una de las dificultades encontradas ha sido que el patrón de estado no permite (teóricamente) el intercambio de información entre los diferentes estados. Sin embargo, en un juego parece pertinente que cierta información pueda ser compartida (como que el estado final pueda saber qué jugador ha resultado vencedor en el estado de juego). Para ello se ha implementado una clase de contexto que encapsula un diccionario con cualquier dato que se quiera pasar de un estado al otro. En la entrada de cada estado se pasa este contexto, y se devuelve a la salida, con o sin información añadida:

```python
def __init__(self) -> None:
    self.__data: dict = {}

def set_data(self, **kwargs: Any) -> None:
    """Sets the context data."""
    self.__data.update(**kwargs)

@property
def data(self) -> dict:
    """Returns the context data."""
    return self.__data
```

Posibles mejoras

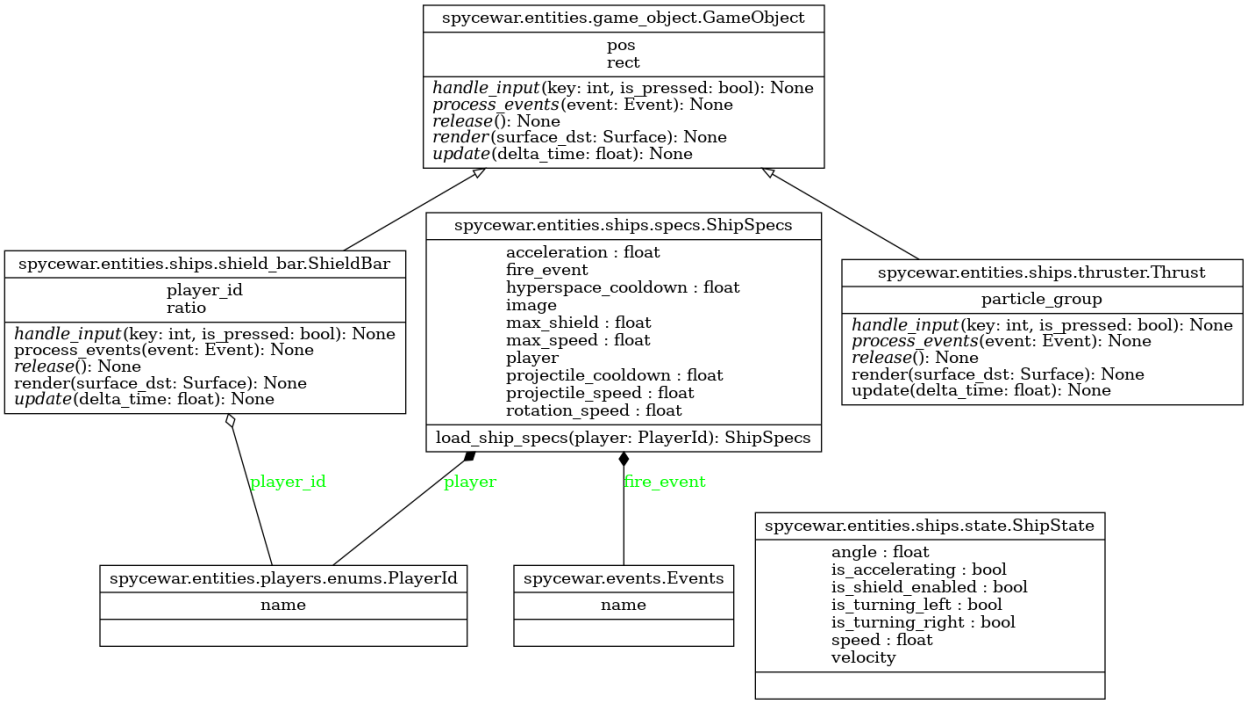
Spycewar! podría extenderse y mejorarse con las siguientes características:

- Implementación de IA para uno de los jugadores.
- Barras de combustible y munición, junto con los correspondientes *powerups*.
- Una estrella central con fuerza gravitatoria, como en el videojuego clásico.
- Sistema de aciertos críticos, al estilo de juegos RPG.
- Adición de efectos de sonidos para la propulsión, las colisiones, y los disparos.
- Un sistema más complejo de puntuación que cuente varias partidas seguidas.
- Visualización de los puntos de daño, al estilo de juegos RPG.
- Mejoras visuales, como diferentes colores para golpes críticos, o misiles bloqueados.
- Instalador para SO.
- Rediseño de las naves.
- Menú con opciones de juego.

Anexo

Estructura de clases

Jugadores



Naves