TP Videojuegos
Ejercicio 1

En las primeras clases de este cuatrimestre, hemos visto cómo utilizar la composición en lugar de la herencia para definir comportamientos (entrada, física y renderización) de un `GameObject`. En este ejercicio vamos a practicar esta idea a partir del código de juego Ping-Pong que hemos escrito en clase. La mayor parte del código de este ejercicio será reutilizado en la práctica 1, en la que vamos a desarrollar una variante del juego **Asteroids**.

El objetivo general de este ejercicio es implementar un actor que representa a un avión de combate (*caza*) que pueda moverse en el espacio, rotar, acelerar, desacelerar y disparar.

El resto del enunciado es una guía *paso a paso* para hacer el ejercicio. En cada paso vas implementar un nuevo componente y utilizarlo para modificar el comportamiento del caza.

Paso 1: `ImageRenderer`

Implementa un `RenderComponent` que permite renderizar un `GameObject` como una imagen (proporcionada como `Texture`) teniendo en cuenta su dirección, es decir, girando la imagen de tal manera que coincida su frente con la dirección del `GameObject` correspondiente. Empieza desde la clase `ImageRenderer` (disponible en el código proporcionado) y modificarla para que haga también la rotación.

Suponemos que las imágenes (es decir los archivos png) tienen el frente siempre en la parte superior, es decir, en la dirección del vector `(0,-1)`. Al dibujar la imagen se debe rotar α grados, donde α es el ángulo entre el vector `(0,-1)` y el vector `o->getDirection()` del `GameObject o`. La clase `Vector2D` tiene un método para calcular el ángulo entre dos vectores. Recuerde que el eje vertical (y-axis) de nuestro sistema de coordenadas está invertido para facilitar el trabajo con SDL. La clase `Texture` tiene un método `render` que recibe el ángulo de rotación como parámetro y renderiza la textura con rotación correspondiente.

Pruebas: crea un `GameComponent` que represente al caza, y añadirle una instancia de `ImageRenderer` usando una de las texturas de imagen `Resources::SpaceShips` o `Resources::Airplanes` (elige cualquiera de las imágenes del correspondiente png mediante un `clip` apropiado). Hay que modifica la clase `ExampleGame` para declarar atributos correspondientes y modificar los métodos `initGame()` y `closeGame()` de manera adecuada. Cambia la dirección inicial del caza a algún vector para ver cómo gira la imagen.

Paso 2: `RotationInputComponent`

Implementa un `InputComponent` que permite girar un `GameObject`. El constructor debe recibir el ángulo de rotación α y dos teclas para la rotación (*clockwise* y *counter-clockwise*).

Cuando se pulsa la tecla de rotación *clockwise*, giramos el vector de dirección del `GameObject` correspondiente α grados, y cuando se pulsa la tecla de rotación *counter-clockwise*, giramos el vector de dirección del `GameObject` correspondiente $-\alpha$ grados. La clase `Vector2D` tiene un método para rotar un vector.

Pruebas: crea una instancia de `RotationInputComponent` que se controla usando las teclas ← y →, con un ángulo de rotación de 5 grados, y añádela al componente del caza.

Paso 3: `CircularMotionPhysics`

Implementa un `PhysicsComponent` que permite a un `GameObject` moverse de forma circular, es decir, cuando sale completamente de un lado aparece en el lado opuesto. En cada llamada al método `update` la posición del objeto cambia primero a `o->getPosition()+o->getVelocity()`, y luego se actualiza sus componentes `X` y `Y` dependiendo de si sale de los bordes de la ventana, etc.

Pruebas: crea una instancia de `CircularMotionPhysics` y añádela al componente del caza. Cambia la velocidad inicial del caza `(1,2)`, de lo contrario, no se moverá ya que la velocidad por defecto es `(0,0)`. Más adelante, cuando el caza tenga la posibilidad de acelerar, deberás volver a ponerlo la velocidad inicial a `(0,0)`.

Paso 4: `AccelerationInputComponent`

Ahora vamos a implementar un `InputComponent` que permite al caza acelerar y desacelerar. El constructor del componente recibe como parámetros dos teclas que se usarán para acelerar y desacelerar.

Aceleración: note que el caza mira en la dirección del vector `getDirection()` mientras se mueve en la dirección del vector `getVelocity()`. El proceso de acelerar no es simplemente aumentar la magnitud del vector de velocidad, sino también modificar la dirección del vector de velocidad gradualmente de tal manera que al final el caza se mueva en la misma dirección que está mirando, es decir, las direcciones de `getDirection()` y `getVelocity()` son iguales. Esto puede hacer cambiando la velocidad del `GameObject` a `getVelocity()+getDirection()*thrust`, donde el `thrust` es un número (de tipo `double`) que representa algún potencia que se aplica en el caza para cambiar su dirección de movimiento. Haz que el valor de `thrust` sea un parámetro del constructor. Además, la velocidad del caza debe tener un límite máximo que se proporciona como parámetro al constructor, cuando se exceda ese

límite se debe normalizar el vector de velocidad (es decir, hacer que su magnitud sea 1 usando `v.normalize())` y luego multiplicarlo por el límite de velocidad.

Desaceleración: Implementa una de las siguientes opciones (o las dos si quieres) para reducir la velocidad:

A. Reducir la magnitud del vector de velocidad por una constante `c`, eso se puede hacer de la siguiente manera. Si `v.magnitude()>=c>0`, entonces cambiamos el vector de velocidad a `v*((v.magnitude()-c)/v.magnitud())`, en otros casos lo cambiamos a `(0,0)`.

B. Reducir la magnitud del vector de velocidad por un *factor de reducción* entre 0 y 1, es decir, cambiamos el vector de velocidad a `v*reductionFactor`.

Los valores de `c` y `reductionFactor` deben ser parámetros del constructor. Además, si la magnitud del vector de velocidad es menor que algún *epsilon* (por ejemplo 0.1) lo cambiamos a `(0,0)`.

Pruebas: crea un componente `AccelerationInputComponent` controlado por las flechas ↑ y ↓. Prueba con diferentes valores para `thrust` (por ejemplo, entre 0.5 y 1) para ver cómo afecta el movimiento del caza.

Paso 5: `BulletsManager`

Crea una interfaz que representa a un *gestor de balas*. La interfaz debe incluir un sólo método `shoot(GameObject* o, Vector2D p, Vector2D v)`. Este método se usará para crear una bala con posición inicial `p` y velocidad `v`. Es sólo una interfaz, en el siguiente paso implementamos una clase que hereda de esa interfaz.

Paso 6: `StarWarsBulletsManager`

Implementa un `GameObject` que sea también un `BulletsManager`, es decir, hereda tanto de `GameObject` como de `BulletsManager`. La clase debe mantener una lista de balas (por ejemplo, usando `std::vector<GameObject*>`) que representen las balas que están actualmente en la pantalla. A continuación detallamos los métodos `shoot`, `update` y `render`.

`shoot`: crea un nuevo `GameComponent` con los componentes `FillRectRenderer` y `BasicMotionPhysics` (ambos disponibles en el código proporcionado), de tamaño suficientemente pequeño, por ejemplo, 2x2, y lo añade a la lista de balas. Hay que usar las mismas instancias de FillRectRenderer y BasicMotionPhysics con todas las balas.

`update`: llama al `update` de todas las balas y borra todas las balas que sale de los bordes de la ventana (no olvidar de liberar la memoria dinámica).

render: llama al `render` de todas las balas.

Pruebas: crea un `StarWarBulletsManager` y pruébalo llamando varias veces a `shoot` en `initGame()`. Eliminar esas llamadas después de probar la funcionalidad.

Step 7: `GunInputComponent`

Crea un `InputComponent` que represente un arma. El constructor del componente recibe un `BulletsManger*` y una tecla para disparar. Cuando se pulsa la tecla se debe llamar método `shoot` del gestor de balas con valores adecuados para la posición inicial y la velocidad de la bala, como detallamos a continuación.

Suponemos que `(x,y)` es la esquina superior-izquierda del rectángulo del `GamObject` del caza, `w` y `h` son la anchura y la altura de ese rectángulo, `d` es su dirección, y `v` es la velocidad.

La posición inicial de la bala debe ser el punto medio en el frente del caza: se puede calcular como `(x,y)+(w/2,h/2)+d*(h/2)`. La velocidad de la bala debe ser N más rápida que el caza, y por lo menos dos. La bala debe moverse en la dirección al que el caza está mirando y no en la dirección en la que está moviendo, es decir, la velocidad de la bala se calcula como `d*max(v.magnitute()*N,2.0)` -- Prueba con distintos valores de `N`.Recuerda que la magnitud del vector de dirección `d` es siempre 1 -- ver `GameObject.cpp`.

Pruebas: crea un `GunInputComponent` controlado por la tecla Space Bar y añádelo al componente del caza.

Step 8: Disparos per intervalo de tiempo

Modifique el constructor de `GunInputComponent` para que reciba dos parámetros adicionales: `shotsPerInterval` y `timeInterval` (de tipo `Uint8` y `Uint32`). El componente debe permitir al jugador disparar como máximo `shotsPerInterval` veces cada `timeInterval` milisegundos.

Pruebas: modifica el `GunInputComponent` que has creado para utilizar esos parámetros.

Comentario: el código proporcionado incluye un `RendeComponent` llamado `SkeletonRenderer`, cuando se añade a un `GameComponent` dibuja el rectángulo y los vectores de velocidad y dirección. Puede ayudarte a depurar tu programa.