TP Videojuegos 2

Práctica 2

Fecha Límite: 07/04/2022.

Leer todo el enunciado antes de empezar con la implementación.

En esta práctica vamos a desarrollar el juego Asteroids usando el patrón de diseño ECS donde los componentes tienen sólo datos y los comportamientos están definidos en sistemas. Hay que usar ecs_3.zip, y para la comunicación entre sistemas es obligatorio usar el mecanismo de enviar/recibir mensajes. Es muy recomendable descargar el ejemplo correspondiente del juego Ping Pong y estudiarlo antes de empezar con la implementación de la práctica 2. El juego desarrollado en esta práctica tiene que tener el mismo comportamiento como el de la práctica 1.

A continuación se propone un diseño de sistemas y componentes que podéis usar. Es sólo una recomendación, se puede modificar o usar otro diseño. Se puede reutilizar código de la práctica 1, es muy recomendable, no reescribáis todo, simplemente mover los comportamientos a los sistemas correspondientes. Los componentes tienen que ser `struct` y sin definir métodos `render` y `update`.

El bucle principal tiene que incluir sólo llamadas a `update` de los sistemas, a `refresh` del `Manager` para borrar entidades no vivas, y a `flush` del `Manager` para enviar mensajes pendientes (si usas el mecanismo de enviar mensajes con delay).

Propuesta de diseño de sistemas y componentes

Los componentes de las distintas entidades son como en la práctica 1, pero el componente `State` que usábamos para mantener el estado del juego ya no hace falta, porque vamos a mantener el estado como un atributo en el sistema correspondiente (`GameCtrlSystem`). Tampoco el componente `GameCtrl` hace falta ya que `GameCtrlSystem` se encarga de esa funcionalidad. También se puede no usar `Image` y decidir en el sistema correspondiente qué imagen usar (por lo menos para las balas). `FramedImag` sigue siendo necesario porque lleva información distinta para cada entidad. Para distinguir entre los tipos de asteroides se puede usar un nuevo componente `AsteroidType` que incluye un atributo para indicar el tipo (la práctica 1 tenía 2 tipos).

El juego tendrá los siguiente sistemas:

1. `Game Control System`
2. `Asteroids System`
3. `Bullets System`
4. `Fighter System`
5. `Fighter's Gun System` (se puede incorporar en `Bullets System` o `Fighter System`)
6. `Collisions System`
7. `RenderSystem`

A continuación se explica el objetivo de cada sistema y sus métodos. Puedes añadir/quitar métodos, añadir/quitar parámetros, etc.

TP Videojuegos 2, Práctica 2, Fecha Límite: 07/04/2022.

Game Control System

Es un sistema para mantener el estado del juego, decidir cuándo acaba una ronda, cuando acaba el juego, etc. También permite al jugador empezar una ronda (o nueva partida) pulsando `SDLK_SPACE`.

```
class GameCtrlSystem: public System {
public:

	// Reaccionar a los mensajes recibidos (llamando a métodos correspondientes).
	void receive(const Message &m) override ...

	// Inicializar el sistema, etc.
	void initSystem() override ...

	// Si el juego no está parado y el jugador pulsa SDLK_SPACE cambia el estado
	// como en la práctica 1, etc. Tiene que enviar mensajes correspondientes cuando
	// empieza una ronda o cuando empieza una nueva partida.
	void update() override ...

private:
	// Para gestionar el mensaje de que ha habido un choque entre el fighter y un
	// un asteroide. Tiene que avisar que ha acabado la ronda, quitar una vida
	// al fighter, y si no hay más vidas avisar que ha acabado el juego (y quien
	// es el ganador).
	void onCollision_FighterAsteroid() ...

	// Para gestionar el mensaje de que no hay más asteroides. Tiene que avisar que
	// ha acabado la ronda y además que ha acabado el juego (y quien es el ganador)
	void onAsteroidsExtinction() ...

	Uint8 winner_; // 0 - None, 1 - Asteroids, 2- Fighter
	Uint8 state_;  // El estado actual del juego (en lugar del componente State)
```

ASTEROIDS SYSTEM

Es un sistema responsable de los asteroides (crear, destruir, etc.)

```
class AsteroidsSystem: public System {
public:

	// Reaccionar a los mensajes recibidos (llamando a métodos correspondientes).
	void receive(const Message &m) override …

	// Inicializar el sistema, etc.
	void initSystem() override …

	// Si el juego está parado no hacer nada, en otro caso mover los asteroides como
	// en la práctica 1 y generar 1 asteroide nuevo cada 5 segundos (aparte
	// de los 10 al principio de cada ronda).
	void update() override …

private:
	// Para gestionar el mensaje de que ha habido un choque de un asteroide con una
	// bala. Desactivar el asteroide “a” y crear 2 asteroides como en la práctica 1,
	// y si no hay más asteroides enviar un mensaje correspondiente.
	void onCollision_AsteroidBullet(Entity *a) …

	// Para gestionar el mensaje de que ha acabado la ronda. Desactivar todos los
	// asteroides, y desactivar el sistema.
	void onRoundOver() …

	// Para gestionar el mensaje de que ha empezado una ronda. Activar el sistema y
	// añadir los asteroides iniciales (como en la práctica 1).
	void onRoundStart() …

	// El número actual de asteroides en el juego (recuerda que no puede superar un
	// límite)
	Uint8 numOfAsteroids_;

	// Indica si el sistema está activo o no (modificar el valor en onRoundOver y
	// onRoundStart, y en update no hacer nada si no está activo)
	bool active_;
}
```

BULLETS SYSTEM

Es un sistema responsable de las balas (crear, destruir, etc.)

```
class BulletsSystem: public System {
public:

	// Reaccionar a los mensajes recibidos (llamando a métodos correspondientes).
	void receive(const Message &m) override …

	// Inicializar el sistema, etc.
	void initSystem() override …

	// Si el juego está parado no hacer nada, en otro caso mover las balas y
	// desactivar las que salen de la ventana como en la práctica 1.
	void update() override …

private:

	// Para gestionar el mensaje de que el jugador ha disparado. Añadir una bala al
	// juego, como en la práctica 1. Recuerda que la rotación de la bala sería
	// vel.angle(Vector2D(0.0f,-1.0f))
	void shoot(Vector2D pos, Vector2D vel, double width, double height) …

	// Para gestionar el mensaje de que ha habido un choque entre una bala y un
	// asteroide. Desactivar la bala “b”.
	void onCollision_BulletAsteroid(Entity *b) …

	// Para gestionar el mensaje de que ha acabado la ronda. Desactivar todas las
	// balas, y desactivar el sistema.
	void onRoundOver() …

	// Para gestionar el mensaje de que ha empezado una ronda. Activar el sistema.
	void onRoundStart() …

	// Indica si el sistema está activo o no (modificar el valor en onRoundOver y
	// onRoundStart, y en update no hacer nada si no está activo)
	bool active_;
}
```

FIGHTER SYSTEM

Es un sistema responsable del caza (moverlo, etc.)

```
class FighterSystem: public System {
public:

	// Reaccionar a los mensajes recibidos (llamando a métodos correspondientes).
	void receive(const Message &m) override …

	// Crear la entidad del caza, añadir sus componentes, asociarla con un handler
	// correspondiente, etc.
	void initSystem() override …

	// Si el juego está parado no hacer nada, en otro caso actualizar la velocidad
	// del caza y moverlo como en la práctica 1 (acelerar, desacelerar, etc).
	void update() override …

private:
	// Para reaccionar al mensaje de que ha habido un choque entre el fighter y un
	// un asteroide. Poner el caza en el centro con velocidad (0,0) y rotación 0. No
	// hace falta desactivar la entidad (no dibujarla si el juego está parado).
	void onCollision_FighterAsteroid() …

	// Para gestionar el mensaje de que ha acabado una ronda. Desactivar el sistema.
	void onRoundOver() …

	// Para gestionar el mensaje de que ha empezado una ronda. Activar el sistema.
	void onRoundStart() …

	// Indica si el sistema está activo o no (modificar el valor en onRoundOver y
	// onRoundStart, y en update no hacer nada si no está activo)
	bool active_;

}
```

FIGHTER'S GUN SYSTEM

Es un sistema responsable del arma, es decir para disparar. En lugar de implementarlo como un sistema separado, se puede incorporar directamente en `BulletsSystem` o `FighterSystem`.

```
class FighterGunSystem: public System {
public:
	// Reaccionar a los mensajes recibidos (llamando a métodos correspondientes).
	void receive(const Message &m) override …

	// Inicializar el sistema, etc.
	void initSystem() override …

	// Si el juego no está parado y el jugador pulsa SDLK_S, enviar un mensaje
	// correspondiente con la características físicas de la bala (como en la
	// práctica 1). Recuerda que se puede disparar sólo una bala cada 0.25sec.
	void update() override …

private:
	// Para gestionar el mensaje de que ha acabado una ronda. Desactivar el sistema.
	void onRoundOver() …

	// Para gestionar el mensaje de que ha empezado una ronda. Activar el sistema.
	void onRoundStart() …

	// Indica si el sistema está activo o no (modificar el valor en onRoundOver y
	// onRoundStart, y en update no hacer nada si no está activo)
	bool active_;
}
```

COLLISIONS SYSTEM

Es un sistema responsable de comprobar colisiones entre el caza y los asteroides y entre las balas y los asteroides.

```
class CollisionsSystem: public System {
public:
	// Reaccionar a los mensajes recibidos (llamando a métodos correspondientes).
	void receive(const Message &m) override …

	// Inicializar el sistema, etc.
	void initSystem() override …

	// Si el juego está parado no hacer nada, en otro caso comprobar colisiones como
	// en la práctica 1 y enviar mensajes correspondientes.
	void update() override …

private:
	// Para gestionar el mensaje de que ha acabado una ronda. Desactivar el sistema.
	void onRoundOver() …

	// Para gestionar el mensaje de que ha empezado una ronda. Activar el sistema.
	void onRoundStart() …

	// Indica si el sistema está activo o no (modificar el valor en onRoundOver y
	// onRoundStart, y en update no hacer nada si no está activo)
	bool active_;
```

RENDER SYSTEM

Es un sistema responsable de renderizar todas las entidades, mensajes, etc. Otra posibilidad es definir un método `render` en la clase `System`, y distribuir este código entre los sistemas correspondientes en lugar de tener un sólo sistema para rendering.

```
class RenderSystem: public System {
public:
	// Reaccionar a los mensajes recibidos (llamando a métodos correspondientes).
	void receive(const Message &m) override ...

	// Inicializar el sistema, etc.
	void initSystem() override ...

	// - Dibujar asteroides, balas y caza (sólo si el juego no está parado).
	// - Dibujar las vidas (siempre).
	// - Dibujar los mensajes correspondiente: si el juego está parado, etc (como en
	//   la práctica 1)
	void update() override ...
private:
	// Para gestionar los mensajes correspondientes y actualizar los atributos
	// winner_ y state_. Otra posibilidad es simplemente consultar el valor de
	// winner_ y state_ directamente a GameCtrlSystem (en este caso no hacen falta
	// los siguientes métodos y atributos)
	void onRoundStart() ...
	void onRoundOver() ...
	void onGameStart() ...
	void onGameOver() ...

	Uint8 winner_; // 0 - None, 1 - Asteroid, 2- Fighter
	Uint8 state_;  // El estado actual de juego (como en GameCtrlSystem)
}
```