Práctica 2: Asteroids 2.0

Tecnología de la Programación de Videojuegos. UCM

El objetivo de esta práctica es avanzar en el uso de una arquitectura ECS. Para ello partiremos del *Asteroids* 1.0 de la práctica anterior y desarrollaremos una serie de modificaciones para usar el patrón de diseño ECS donde los componentes tienen sólo datos y los comportamientos están definidos en sistemas. Para la comunicación entre sistemas es obligatorio usar el mecanismo de enviar/recibir mensajes. El juego desarrollado en esta práctica tiene que tener el mismo comportamiento que el de la práctica 1. A continuación se detallan los requisitos de esta práctica:

- Los **comportamientos** están definidos en **sistemas**.
- Los **componentes** sólo tienen **datos**.
- Los **componentes** tienen que ser **struct**.
- Los componentes no tendrán método **update** ni **render**, ni **handleEvents**.
- Se utilizará un mecanismo de paso de mensajes con **retardo**.
- El bucle principal de juego sólo hará llamadas **update de los sistemas**, **refresh del manager** para borrar entidades no vivas, y **flush del manager** para enviar mensajes pendientes.
- El juego tendrá los siguientes sistemas:
    - GameCtrlSystem
    - AsteroidsSystem
    - BulletSystem)
    - FighterSystem)
    - CollisionsSystem
    - RenderSystem
- El estado del juego (pausa) se puede controlar a través de una máquina de estados o por un atributo en el sistema correspondiente (**GameCtrlSystem**).
- El componente **GameCtrl** ya no hace falta, ya que **GameCtrlSystem** se encarga de esa funcionalidad.
- No usaremos **Image** ya que el sistema correspondiente decidirá que imagen usar. **FrameImage** sigue siendo necesario, porque lleva información distinta para cada entidad.

Detalles de implementación.

GameCtrlSystem

Mantiene el estado del juego, o se comunica con la máquina de estados para transitar entre ellos. Decide cuándo acaba una ronda, cuando acaba el juego, etc.

```
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)
```

AsteroidsSystem

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_;
}
```

BulletSystem

Sistema responsable de las balas (crearlas, destruirlas, 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_;
}
```

FighterSystem

Sistema responsable del caza (moverlo, disparar…).

```
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). Además,
        // si el juego no está parado y el jugador pulsa la tecla de disparo, enviar un
        // mensaje con las características físicas de la bala. Recuerda que se puede disparar
        // sólo una bala cada 0.25sec.
        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_;
}
```

CollisionsSystem

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_;

}
```

RenderSystem

Sistema responsable de renderizar todas las entidades, mensajes, etc.

```
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 correspondientes: 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_.
        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)
}
```

Pautas generales obligatorias

A continuación, se indican algunas pautas generales que vuestro código debe seguir:

- Se mantendrá la estructura de carpetas de la práctica 1. Los nuevos ficheros propios de la arquitectura (`System.h`,…) se incorporarán a la carpeta `src/ecs`. Se incorpora una carpeta `src/systems` a la estructura donde deben estar los ficheros relativos a los sistemas que se creen.
- Asegúrate de que el programa no deje basura. Para que Visual te informe debes escribir al principio de la función `main` esta instrucción

```
CrtSetDbgFlag( CRTDBG_ALLOC_MEM_DF | CRTDBG_LEAK_CHECK_DF );
```

- Todos los atributos deben ser privados excepto quizás algunas constantes del juego en caso de que se definan como atributos estáticos.
- Define las constantes que sean necesarias. En general, no deben aparecer literales que pudiesen corresponder con configuraciones del programa en el código.
- No debe haber métodos que superen las 30-40 líneas de código.
- Escribe comentarios en el código, al menos uno por cada método que explique de forma clara qué hace el método. Se cuidadoso también con los nombres que eliges para variables, parámetros, atributos y métodos. Es importante que denoten realmente lo que son o hacen. **Usa nombres en inglés**.
- Hay que inicializar todos los atributos en las constructoras (en el mismo orden de la declaración).
- Al usar la directiva `#include`, escribe el nombre del archivo *respetando minusculas y mayusculas*.
- Usar el formateo (indentation) automático de **Visual Studio**.
- No dejar en la entrega clases o recursos que no se usan.

Entrega

En la *tarea del campus virtual* y *dentro de la fecha límite*, **cada uno de los miembros del grupo**, debe subir un fichero comprimido (**.** `zip`) que contenga la carpeta de la solución y el proyecto limpio de archivos temporales (asegúrate de borrar la carpeta oculta .vs y ejecuta en Visual Studio la opción "limpiar solución" antes de generar el .zip. La carpeta debe incluir un archivo `info.txt` con los nombres de los componentes del grupo y unas líneas explicando las funcionalidades opcionales incluidas y/o las cosas que no estén funcionando correctamente. Ese mismo texto debes subirlo también en el cuadro de texto (sección "texto en línea") asociado a la entrega.

Además, para que la práctica se considere entregada, deberá pasarse una *entrevista* en la que el profesor comprobará, con los dos autores de la práctica, su funcionamiento en ejecución, y si es correcto realizará preguntas (individuales) sobre la implementación. Las entrevistas se realizarán en las sesiones de laboratorio siguientes a la fecha de entrega, o si fuese necesario en horario de tutorías.