TP Videojuegos 2
Práctica 2
Fecha Límite: 08/04/2024.

Leer todo el enunciado antes de empezar con la implementación.

En esta práctica vamos a desarrollar una variación muy sencilla del juego PacMan usando el patrón de diseño ECS **con sistemas** (al contrario de la práctica 1 que usaba solo ECS). Hay que usar `ecs_4.zip`, y para la comunicación entre sistemas es obligatorio usar el mecanismo de enviar/recibir mensajes**.**

Es muy recomendable descargar uno de los ejemplos correspondientes y estudiarlo antes de empezar con la implementación de la práctica 2.

A continuación se propone un diseño de sistemas, componentes, mensajes, etc., que podéis usar. Es sólo una recomendación, se puede modificar o usar otro diseño pero hay que justificarlo durante la defensa de la práctica.

Descripción del juego

Comportamiento básico

Las entidades de juego son `PacMan`, `Fantasmas` y `Frutas` (las frutas aparecen como cerezas 🍒 en el comportamiento básico).

Al principio de una ronda ponemos las frutas de forma de un grid, como se ve en el demo. El objetivo del `PacMan` es comer todas las frutas. Si consigue comerlas todas acaba el juego y gana, si choca con un fantasma pierde una vida (inicialmente tiene `3`) y tendrá la posibilidad de jugar otra ronda si tiene más vidas, en otro caso pierde y tendrá la posibilidad de empezar una nueva partida. El estado de las cerezas se mantiene entre las rondas. Los fantasmas desaparecen cuando acaba una ronda.

El `PacMan` empieza la ronda en el centro, con vector de velocidad `Vector2D(0.0f,0.0f)`. Pulsando ↑ cambiamos su vector de velocidad a `Vector2D(0.0f,-3.0f).rotate(rot)` donde `rot` es su rotación. Pulsando ↓ cambiamos su vector de velocidad a `Vector2D(0.0f,0.0f)`, es decide deja de moverse. Pulsando ← restamos `90.0f` grados de la rotación y rotamos el vector de velocidad en `-90.0f` grados. Pulsando → sumamos `90.0f` grados y rotamos el vector de velocidad en `90.0f`. Esto hace que se mueva en la dirección donde apunta su boca. Cuando el `PacMan` choca con los laterales para, es decir ponemos su vector de velocidad a `Vector2D(0.0f,0.0f)`.

Durante una ronda, generamos un fantasma cada `5` segundos pero habrá como mucho `10` fantasmas en la pantalla. Los fantasmas siempre salen de una esquina aleatoria, y para que muevan como se ve en el video, en cada iteración, con probabilidad de `0.005` actualizamos su vector de velocidad a

`(posPM-posF).normalize()*1.1f` donde `posPM` es la posición del `PacMan` y `posF` es la posición del fantasma. Esto hace que el fantasma se mueva hacia el `PacMan` pero no de manera continua. Probar con otros valores en lugar de `1.1f` y `0.01` para conseguir un comportamiento razonable. Los fantasmas no salen de la pantalla, cuando chocan con algún lado invertimos la coordenada correspondiente del vector de velocidad para que se vayan hacia dentro. Los fantasmas no pasan de ronda, es decir cuando empieza una ronda siempre hay `0` fantasmas.

Frutas milagrosas e inmunidad del PacMan

La *fruta milagrosa* es una fruta que puede estar en `2` estados: *normal* y *milagroso*. En el estado *normal* se comporta como las cerezas, pero en el estado *milagroso* aparecerá como una pera (🍐) en lugar de una cereza (🍒) y si el `PacMan` la come, tendrá inmunidad durante `10` segundos:

1. Los fantasmas actuales cambian de color rojo a azul.
2. No generamos fantasmas nuevos.
3. Cuando el `PacMan` choca con un fantasma, muere el fantasma.
4. Si el `PacMan` come otra fruta milagrosa no se obtiene más inmunidad.

Pasados los `10` segundos, todo vuelve a la normalidad.

Al principio del juego, cuando generamos frutas, con probabilidad del `0.1` se genera una fruta milagrosa (y con el `0.9` una cereza normal) y se le asigna una frecuencia `N` (número aleatorio entre `10` y `20` segundos – cada una puede tener otro `N`). Durante la ronda, una fruta milagrosa se convierte en milagros cada `N` segundos (recuerda que cada una tiene su `N`) y queda en este estado durante `M` segundos (un número aleatorio entre `1` y `5` que hay que elegir en el momento en el que la fruta se convierte en milagrosa – cada una tiene su `M`).

Cuando empieza una ronda, hay que resetear los contadores de tiempo (el de generacion de fantasmas, el de frutas milagrosas, etc.)

Hay que reproducir sonidos correspondientes para los distintos eventos (ver lo archivos en la carpeta de la tarea).

Propuesta de diseño

Mensajes

Recuerda que hay que definir los mensajes en el archivo **`game/messages_defs.h`**. El archivo **`ecs.h`** hace referencia a este archivo. Es recomendable tener los siguientes tipos de mensajes pero no es obligatorio (lo nombres indican a qué evento corresponden):

```
using msgId_type = uint8_t;
enum msgId : msgId_type {
  _m_NEW_GAME, _m_ROUND_START, _m_ROUND_OVER, _m_GAME_OVER,
```

```
    _m_PACMAN_FOOD_COLLISION, _m_PACMAN_GHOST_COLLISION,
    _m_IMMUNITY_START, _m_IMMUNITY_END
};

struct Message {
    msgId_type id;
      union {
              // Añadir sub-structs para los mensajes que llevan más información
            }
     }
```

Sistemas

Es recomendable tener los siguiente sistemas (pero no es obligatorio):

1. **`PacManSystem`**: responsable del `PacMan`. El método `update` mueve el `PacMan` según hemos explicado arriba. Cuando empieza una nueva ronda resetea su posición. Cuando empieza una partida resetea las vidas.
2. **`GhostSystem`**: responsable de generar y mover los fantasmas. Cuando acaba una ronda quita todos los fantasmas actuales. Cuando el `PacMan` choca con un fantasma, si el `PacMan` tiene inmunidad desaparece el fantasma, y si no tiene inmunidad muere el `PacMan` (y envía un mensaje correspondiente de fin de ronda/partida y cambia al estado `NewRoundState` o `GameOverState` – ver los estados abajo). Recuerda que no se generan fantasmas cuando el `PacMan` tiene inmunidad. Cuando cambia el estado de la inmunidad del `PacMan` tiene que cambiar la imagen que se usa para dibujar los fantasmas.
3. **`FoodSystem`**: responsable de las frutas. Cuando empieza una partida coloca las frutas en forma de grid, y algunas serán milagrosas según hemos explicado arriba. En el método `update` hay que actualizar el estado de la frutas milagrosas según hemos explicado arriba. Cuando el `PacMan` choca con una fruta desaparece la fruta, y cuando no hay más frutas envía un mensaje que se haya acabado la partida y cambia de estado as `GameOverState` (ver los estado abajo).
4. **`ImmunitySystem`**: si el `PacMan` choca con una fruta milagrosa que está en estado *milagroso*, avisa que haya empezado un periodo de inmunidad (si no se encuentra en uno ya). En el `update`, si actualmente se encuentra en un estado de inmunidad y hayan pasado `10` segundos desde que su inicio, lo apaga enviando un mensaje correspondiente.
5. **`CollisionsSystem:`** comprobar colisiones entre los varios tipos de entidades y envía mensajes correspondientes.
6. **`RenderSystem`**: Dibuja las frutas, fantasmas, el `PacMan` y sus vidas, etc. Si quieres, puedes añadir un método `render` en la clase `System` e incorporar esta funcionalidad en los distintos

sistemas. Recuerda que hay que dibujar los fantasmas y el `PacMan` usando varios frames para tener la animación que se ve en el demo.

Componentes

A parte del `Transfrom` y `FramedImage/Image` que lo tienen todos, el `PacMan` tendrá un componente para indicar el número de vidas y otro para indicar si es inmune. Las frutas tendrán un componente para indicar si son milagrosas (con toda la información necesaria para para poder cambiar de estado, es decir `N`, `M`, cuando se activó, etc).

Estados de juego

Los estados del juego son parecidos a los de la práctica 1

1. **`RunningState`**: su `update` llama al `update` de los sistemas y al `refresh` del `Manager` (la llamada a `refresh` se puede poner en el bucle principal). Además, si usuario pulsa `P` cambia la estado `PauseState`.
2. **`PauseState`**: si el usuario pulsa cualquier tecla cambia al estado `RunningState`.
3. **`NewGameState`:** si el usuario pulsa cualquier tecla, envía un mensaje de que ha empezado una partida nueva, y cambia al estado `NewRoundState`.
4. **`NowRoundState`:** si el usuario pulsa `ENTER`, envía un mensaje de que ha empezado una ronda, y cambia al estado `RunningState`.
5. **`GameOverState`**: si el usuario pulsa cualquier tecla cambia al estado `NewGameState`.

La clase Game

La clase `Game` es parecida a la clase correspondiente de la práctica 1. El método `init` tiene que crear los sistemas y los estados. El bucle principal tiene que incluir sólo una llamada al `update` del estado actual y a `flush` del `Manager` para enviar mensajes pendientes (si usas el mecanismo de enviar mensajes con delay). El destructor tiene que liberar la memoria de los estados.