

TP Videojuegos 2

Práctica 2

Fecha Límite: 05/05/2020 a las 09:00.

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 la plantilla del juego que vimos en clase (PacMan y las estrellas). A continuación proporcionamos más detalles sobre el diseño, pero es sólo una recomendación, se puede usar otro diseño.

Gestión de Memoria

El juego tiene dos factorías de entidades basadas en `ObjectPool`: una para los asteroides y otra para las balas. Son muy parecidas a la clase `StarsPool`. Recuerda que es muy importante inicializar las factorías con el tamaño necesario antes de que empiece el juego llamando al método `init(...)`.

ASTEROIDSPool

En la constructora de la clase `AsteroidPool` hay que inicializar todas las entidades del pool con los componentes necesarios (`Transform`, `ImageComponent` y `AsteroidLifeTime`). Además, la clase tiene que tener el siguiente método para construir un asteroide (se pueden añadir y/o modificar los parámetros):

```
inline Entity* construct_(Vector2D pos, Vector2D vel, double width, double height, int generations) ...
```

Recuerda que para añadir una entidad al juego usando esta factoría hay que usar el parámetro de tipo al llamar a `addEntity`: `mngr_->addEntity<AsteroidsPool>(...)`

BULLETPool

En la constructora de la clase `BulletsPool` hay que inicializar todas las entidades del pool con los componentes necesarios (`Transform` y `ImageComponent`). Además, la clase tiene que tener el siguiente método para construir un asteroide (se pueden añadir y/o modificar los parámetros):

```
inline Entity* construct_(Vector2D pos, Vector2D vel, double width, double height) ...
```

Recuerda que para añadir una entidad al juego usando esta factoría hay que usar el parámetro de tipo al llamar a `addEntity`: `mngr_->addEntity<BulletsPool>(...)`

Componentes

```
// Identificadores de componentes (corresponden a componentes con los mismos nombres)
//
Transform          // el de antes, pero usando struct -- incluido en la plantilla
ImageComponent     // incluye una textura -- incluido en la plantilla
AsteroidLifeTime   // incluye el número de generaciones del asteroid
GameState          // incluye el estado del juego (parado, terminado, etc.) y el ganador
Score              // incluye el total de puntos
Health             // incluye el número de vidas (rondas) que quedan al caza
```

Grupos

```
// Identificadores de grupos
//
_grp_Asteroid      // los asteroids pertenecen a este grupo
_grp_Bullet        // las balas pertenecen a este grupo
```

Handlers

```
// Identificadores de handlers
//
_hdlr_Fighter      // handler para la entidad del caza
_hdlr_GameState    // handler para la entidad del estado del juego
```

Sistemas

```
// Identificadores de sistemas
//
_sys_GameCtrl      // sistema de control del juego (para empezar el juego, etc.)
_sys_Asteroids     // sistema de los asteroids (para mover los asteroides)
_sys_Bullets       // sistema de las balas (para mover las balas)
_sys_Fighter       // sistema del caza (para gestionar el movimiento del caza)
_sys_FighterGun    // sistema del arma (para disparar -- arma para el caza)
_sys_Collisions    // sistema de colisiones (para comprobar todas las colisiones)
_sys_Render        // sistema de rendering (para dibujar las entidades, etc.)
```

GAMECTRLSYSTEM

```
class GameCtrlSystem: public System {
public:
    // - a este método se le va a llamar cuando muere el caza.
    // - desactivar todos los asteroides y las balas.
    // - actualizar los componentes correspondientes (Score, Heath, GameState, ...).
    void onFighterDeath() ...

    // - a este método se le va a llamar cuando no haya más asteroides.
    // - desactivar todas las balas.
    // - actualizar los componentes correspondientes (Score, GameState, ...).
    void onAsteroidsExtencion() ...

    // - crear una entidad, añade sus componentes (Score y GameState) y asociarla
    //   con el handler _hndlr_GameState.
    void init() override ...

    // - si el juego está parado y el jugador pulsa ENTER empieza una nueva ronda:
    //   1) añadir 10 asteroides llamando a addAsteroids del AsteroidsSystem.
    //   2) actualizar el estado del juego y el número de vidas (si es necesario)
    //       en los componentes correspondientes (Score, Heath, GameState, ...).
    void update() override ...
}
```

ASTEROIDSSYSTEM

```
class AsteroidsSystem: public System {
public:
    // - añadir n asteroides al juego como en la práctica 1 pero usando entidades.
    // - no olvidar añadir los asteroides al grupo _grp_Asteroid.
    void addAsteroids(int n) ...

    // - desactivar el asteroide "a" y crear 2 asteroides como en la práctica 1.
    void onCollisionWithBullet(Entity *a, Entity *b) ...

    // - si el juego está parado no hacer nada.
    // - mover los asteroides como en la práctica 1.
    void update() override ...
private:
    std::size_t numOfAsteroids_;
}
```

BULLETSSYSTEM

```
class BulletsSystem: public System {
public:
    // - añadir una bala al juego, como en la práctica 1 pero usando entidades.
    // - no olvidar añadir la bala al grupo _grp_Bullet
    void shoot(Vector2D pos, Vector2D vel, double width, double height) ...

    // - desactivar la bala "b"
    void onCollisionWithAsteroid(Entity *b, Entity *a) ...

    // - si el juego está parado no hacer nada.
    // - mover las balas y desactivar las que se salen de la ventana
    void update() override ...
}
```

COLLISIONSYSTEM

```
class CollisionSystem: public System {
public:
    // - si el juego está parado no hacer nada.
    // - comprobar colisiones usando el esquema abajo (nota las instrucciones break
    //   y continue, piensa porque son necesarias).
    void update() override {
        ...
        for (auto &a : /* asteroides */ ) {
            ...
            if ( /* hay choque entre "a" y el caza */ ) {
                // - llamar a onCollisionWithAsteroid(a) del FighterSystem
                break;
            }
            for (auto &b : /* balas */ ) {
                if ( !b->isActive() ) continue;
                if ( !a->isActive() ) break;
                if ( /* hay choque entre 'a' y 'b' */ ) {
                    // - llamar a onCollisionWithAsteroid(...) del BulletsSystem.
                    // - llamar a onCollisionWithBullet(...) del AsteroidsSystem.
                    // ...
                }
            }
        }
    }
}
```

FIGHTERSYSTEM

```
class FighterSystem: public System {
public:
    // - poner el caza en el centro con velocidad 0 y rotación 0. No hace falta
    //   desactivar la entidad (no dibujarla si el juego está parado en RenderSystem).
    void onCollisionWithAsteroid(Entity *a) ...

    // - crear la entidad del caza, añadir sus componentes (Transform, Health, etc.)
    //   y asociarla con el handler _hndlr_Fighter.
    void init() override ...

    // - si el juego está parado no hacer nada.
    // - actualizar la velocidad del caza y moverlo como en la práctica 1.
    void update() override ...
}
```

FIGHTERGUNSYSTEM

```
class FighterGunSystem: public System {
public:
    // - si el juego está parado no hacer nada.
    // - si el jugador pulsa SPACE, llamar a shoot(...) del BulletsSystem para añadir
    //   una bala al juego -- se puede disparar sólo una bala cada 0.25sec.
    void update() override ...
}
```

RENDERSYSTEM

```
class RenderGunSystem: public System {
public:
    // - dibujar asteroides, balas y caza (sólo si el juego no está parado).
    // - dibujar el marcador y las vidas (siempre).
    // - dibujar el mensaje correspondiente si el juego está parado (como en la
    //   práctica 1: Press ENTER to start, Game Over, etc.)
    void update() override ...
}
```

Asteroids (la clase principal del juego)

En `initGame` inicializar la factorías con el tamaño adecuado y crear los sistemas. El bucle principal tiene que ser algo así

```
exit_ = false;
auto ih = InputHandler::instance();
while (!exit_) {
    SDL_SetRenderDrawColor(game_->getRenderer(), COLOR(0x00AAAAFF));
    SDL_RenderClear(game_->getRenderer());
    Uint32 startTime = game_->getTime();

    ih->update();
    if (ih->keyDownEvent() && ih->isKeyDown(SDLK_ESCAPE)) {
        exit_ = true;
        break;
    }

    entityManager_>refresh();
    gameCtrlSystem_>update();
    ...

    Uint32 frameTime = game_->getTime() - startTime;
    if (frameTime < 10)
        SDL_Delay(10 - frameTime);
    SDL_RenderPresent(game_->getRenderer());
}
```