TP Videojuegos

Práctica 4

Fecha Limite: 25/03/2019 a las 09:00.

En esta práctica vamos a desarrollar una variación del juego clásico Asteroids. El enunciado consiste en varios apartados, en cada uno implementamos una parte del juego. Se recomienda leer todas las partes antes de empezar.

Descripción General

En el juego Asteroids hay 2 actores principales: *el caza* y *los asteroides*.

El objetivo del caza es destruir los asteroides disparándoles. El caza tiene 3 vidas y cuando choca con un asteroide explota y pierde una vida, si tiene más vidas el jugador puede jugar otra ronda. El juego termina cuando el caza no tiene más vidas (pierde) o destruye todos los asteroides (gana). Al comienzo de cada ronda colocamos **10** asteroides aleatoriamente en los bordes de la ventana, con velocidad aleatoria. Los asteroides tienen un *contador de generaciones* con valor inicial **3** (cuando se crean al comienzo de la ronda).

Cuando el caza destruye un asteroide **X**, el asteroide debe desaparecer de la pantalla, y si su contador de generaciones es positivo creamos **2** asteroides nuevos. El contador de generación de cada asteroide nuevo es como el de **X** menos uno, su posición es cercana a la de **X** y su vector de velocidad se obtiene girando el vector de velocidad de **X** y cambiando su magnitud para que sea más rápido (más adelante explicamos cómo calcular la posición y velocidad). El caza gana algunos puntos, depende del número de generaciones de **X**.

El caza está equipado con un arma que puede disparar (más detalles abajo). El número de vidas del caza se debe mostrar en la esquina superior izquierda y el total de puntos ganados en el centro. Entre las rondas y al terminar una partida, se debe mostrar mensajes adecuados y pedir al jugador que pulse `RETURN` para continuar.

Se debe reproducir sonidos correspondientes cuando el caza dispara una bala, cuando un asteroide explota, cuando el caza choca con un asteroide, etc. Además, cuando el juego está en marcha, no entre las rondas, se debe reproducir alguna música. Archivos de sonido, música e imágenes se pueden encontrar en el código proporcionado -- ver los archivos `Resources.h` y `Resources.cpp` -- pero se puede usar otros.

IMPORTANTE: No se puede usar memoria dinámica (i.e., no se puede usar **`new`**). Ver el video en el campus virtual para tener una idea de lo que tienes que implementar. Durante la corrección, aparte de la funcionalidad, se va a tener en cuenta la claridad y organización del código.

Parte 1: Funcionalidad básica del caza

El caza se representa mediante una clase `Fighter` (un `Container`) y componentes correspondientes. La clase tiene una constructora única y varios componentes:

```
class Fighter: public Container {
public:
    Fighter(SDLGame* game);
    virtual ~Fighter();
   // ...
private:
   //...
   // components
    ImageGC fighterImage_;
    NaturalMovePC naturalMove_;
    ShowUpAtOppositeSidePC oppositeSide_;
    RotationIC rotation_;
    ThrustIC thrust_;
    ReduceSpeedPC reduceSpeed_;
    GunIC normalGun_;
};
```

En la constructora hay que (1) fijar la anchura/altura del caza; (2) fijar la posición inicial del caza; y (3) inicializar todos los componentes y añadirlos; A continuación detallamos los componentes.

`ImageGC` (gráfica)**:** Ya existe con el código proporcionado. Hay que inicializar con la imagen que quieres usar, por ejemplo con la textura

```
getServiceLocator()->getTextures()->getTexture(Resources::Airplanes)
```

y el corte (clip) `{47,90,207,250}`.

`NaturalMovePC` (física)**:** Ya existe con el código proporcionado. Es para mover el caza (sumar la velocidad a la posición actual).

`ShowUpAtOppositeSidePC` (física)**:** Cuando el `Container` correspondiente sale por completo de un lado de la pantalla, cambia su posición para que empiece a aparecer en el lado opuesto. Se puede tener en cuenta la rotación del `Container`, pero no es necesario.

`RotationIC` (entrada)**:** Usando dos teclas, este componente gira el caza **α** o **-α** grados (p.ej., `c->setRotation(c->getRotation()+`**α**`)`). La velocidad no cambia, es decir el caza puede mirar hacia una dirección y mover hacia otra dirección. La constructora tiene que recibir las teclas y el valor de **α** como parámetros. En tu práctica usa las teclas `SDLK_LEFT`, `SDLK_RIGHT` y **α**`=5`.

`ThrustIC` (entrada)**:** El movimiento del caza está basado en empujones, eso hace que el juego sea más difícil. Pulsando una tecla, el componente empuja al caza hacia la dirección a donde mira, es decir cambia su velocidad a:

```
v = c->getVelocity()+(Vector2D(0,-1).rotate(c->getRotation())*thrust_
```

donde `thrust_` es un número entre 0 y 1. Si la magnitud del nuevo vector de velocidad, es decir `v.magnitude()`, supera un límite predeterminado cambiamos el vector de velocidad para que tenga ese límite como magnitud, p.ej., usando

```
v = v.normalize()*speedLimit_;
```

La constructora tiene que recibir valores para la tecla, `thrust_` y `speedLimit_`. En tu práctica usa la tecla SDLK_UP, `thrust_` =0.5 y `speedLimit_=2.0`

`ReduceSpeedPC` (física)**:** Este componente complementa el anterior para conseguir el efecto de empujón. Lo que hace es reducir la velocidad del `Container` correspondiente de manera continua, es decir cambia su velocidad a

```
c->getVelocity()*factor_
```

donde factor_ es un número entre 0 y 1. La constructora tiene que recibir el valor de `factor_`. En tu práctica usa el valor 0.995 por ejemplo -- puedes probar con varios.

`GunIC` (entrada): Este componente representa un arma para disparar. Al pulsar una tecla, el componente envía a todos un `msg::Shoot` con la posición, dirección y tipo de la bala que quiere disparar. De momento para el tipo de bala usamos `0` (vamos a usar varios en las siguientes prácticas/ejercicios). El siguiente código calcula la posición y la velocidad de la bala -- en principio, la posición es un punto cerca del frente del caza y la dirección es un vector de magnitud `1` en la dirección donde mira el caza:

```
Vector2D p = c->getPosition()+Vector2D(c->getWidth()/2.0,c->getHeight()/2.0) +
             Vector2D(0.0, -(c->getHeight()/2.0+5.0)).rotate(c->getRotation());
Vector2D d = Vector2D(0, -1).rotate(c->getRotation());
```

Se supone que la posición es el centro de la parte inferior del rectángulo que limita el objeto que corresponde a la bala (más adelante proporcionamos más detalles). Aparte de enviar el mensaje, escribe algo en la pantalla para comprobar que está funcionando porque todavía no vas a ver las balas en la pantalla.

Para probar lo que has hecho en esta parte, añade un atributo de tipo `Fighter` en la clase `AsteroidsGame` y añádelo a la lista de actores en el método `initGame()`.

Parte 2: Funcionalidad básica de los asteroides

Un asteroide se representa mediante una clase `Asteroid` (un `Container`) y componentes correspondientes. Es muy importante definir la constructora por defecto porque vamos a usar objetos de este tipo en un `GameObjectPool`. La clase tiene un atributo `generations_` de tipo int y métodos para consultar y cambiar su valor. Por defecto no tiene ningún componente.

Todos los asteroides se representan mediante una clase `Asteroids` que hereda de `GameObjectPool` (es muy importante leer el código de la clase `GameObjectPool` antes de seguir, es muy parecido a lo que hemos usado en el ejercicio 1):

```
class Asteroids : public GameObjectPool<Asteroid,50> {
public:
   Asteroids(SDLGame* game);
virtual ~Asteroids();
   // ...
private:
   // …
   // component for Asteroid
   ImageGC asteroidImage_;
   NaturalMovePC naturalMove_;
   RotatingPC rotating_;
   ShowUpAtOppositeSidePC showUpAtOppositeSide_;
};
```

En la constructora hay que inicializar los componentes `naturalMove_`, `rotating_`, `asteroidImage_`, y `showUpAtOppositeSide_` y añadirlos a todos los asteroides del object pool (ver getAllObjects() de GameObjectPool) -- son componentes de `Asteroid` y no de `Asteroids`.

A continuación detallamos los componentes que no hemos visto antes.

`RotatingPC` (física)**:** Es un componente para girar continuamente el `Container` correspondiente en **α** grados (usando el método `setRotation` del `Container`). La constructora tiene que recibir el valor de **α.**

Para probar lo que has hecho en esta parte, crea algunos asteroides en la constructora con tamaño `20x20` y posición y velocidad cualquiera. Añade un atributo de tipo `Asteroids` en la clase `AsteroidsGame` y añádelo a la lista de actores en el método `initGame()`. Recuerda que para crear un `Asteroid` usamos:

```
Asteroid *a = getUnusedObject();
a->setActive(true);
```

Parte 3: Funcionalidad básica de las balas

Una bala se representa mediante una clase `Bullet` (un `Container`) y componentes correspondientes. Es muy importante definir la constructora por defecto porque vamos a usar objetos de este tipo en un `GameObjectPool`. La clase tiene un atributo `power_` de tipo `int` y métodos para consultar y cambiar su valor (no lo vamos a usar en esta práctica). Por defecto no tiene ningún componente.

Todas las balas se representan mediante una clase `Bullets` que hereda de `GameObjectPool`:

```
class Bullets: public GameObjectPool<Bullet,10> {
public:
    Bullets(SDLGame* game);
    virtual ~Bullets();
    // ...
private:
    // ...

    // components for Bullet
    NaturalMovePC naturalMove_;
    DeactivateOnBorderExit deactivate_;
    ImageGC bulletImage_;
};
```

En la constructora inicializar los componentes `naturalMove_`, `naturalMove_`, `deactivate_` y `bulletImage_` y añadirlos a todas las balas del object pool (ver getAllObjects() de GameObjectPool) -- son componentes de `Bullet` y no de `Bullets`. Para `bulletImage_` se puede usar la imagen `Resources::WhiteRect` (un rectángulo blanco).

A continuación detallamos los componentes que no hemos visto antes.

`DeactivateOnBorderExit` (física)**:** Es un componente para desactivar el `Container` correspondiente (una bala) cuando sale por completo de la pantalla, para poder reutilizarla.

Para probar lo que has hecho en esta parte, define un método que crea una bala que sale desde el centro de la pantalla hacia arriba. Añade a `Bullets` un componente de entrada que llama a ese método al pulsar alguna tecla. Añade un atributo de tipo `Bullets` en la clase `AsteroidsGame` y añádelo a la lista de actores en el método `initGame()`. Recuerda que para crear un `Bullet` usamos:

```
Bullet *b = getUnusedObject();
b->setActive(true);
```

Parte 4: Funcionalidad básica del GameManager

La clase `GameManager` es la responsable de empezar el juego, comprobar colisiones, decidir cuándo acaba y mostrar mensaje sobre su estado:

```
class GameManager: public Container {
public:
    GameManager(SDLGame* game);
    virtual ~GameManager();
    // ...

private:
    // ...

    static int const maxLives_ = 3;
    bool running_;
    bool gameOver_;
    int score_;
    int lives_;
    int winner_;  // 0=none, 1=asteroids, 2=fighter

    // components of GameManager
    GameCtrlIC gameCtrl_;
    ScoreViewerGC scoreView_;
    GameStatusViewGC gameStatusView_;
    LivesViewer livesViewer_;
    FighterAsteroidCollision fighterAsteroidCollision_;
    BulletsAsteroidsCollision bulletsAsteroidsCollision_;
};
```

En la constructora inicializamos los atributos (`running_=false`, `gameOver_=true`, `score_=0`, `lives_=maxLives_`, `winner_=0`) y los componentes. Hay que definir métodos para consultar los valores de los atributos (no hacen falta métodos *públicos* para cambiar sus valores). A continuación detallamos los componentes.

`GameCtrlIC` (entrada)**:** Cuando el juego está parado (i.e., `running_` es `false`), al pulsar `RETURN` el componente envía a todos un `msg::Message` con el tipo `msg::GAME_START` si se está empezando una nueva partida y siempre envía un `msg::Message` con el tipo `msg::ROUND_START`. El orden es importante.

`ScoreViewerGC` (gráfica)**:** Muestra el marcador en alguna parte de la pantalla.

`LivesViewer` (gráfica)**:** Muestra las vidas que tiene el caza en alguna parte de la pantalla.

`GameStatusViewGC` (gráfica)**:** Muestra un mensaje **"`Game Over`"** con el nombre de ganador al final de la partida. Muestra un mensaje **"`Press ENTER to Continue`"** o **"`Press ENTER to Start a New Game`"**, depende si se está empezando una nueva partida o sólo una nueva ronda.

`FighterAsteroidCollision` (física)**:** Es responsable de comprobar si hay colisiones entre el caza y los asteroides. Tiene 2 atributos:

```
GameObject* fighter_;
const vector<Asteroid*>* asteroids_;
```

con valores iniciales `nullptr`. Más adelante explicamos de donde va a recibir los valores para estos atributos. En el método `update` del componente, si `fighter_` o `asteroids_` tienen valores comprobamos colisiones: para cada asteroide comprueba si cumplen los siguientes condiciones:

1. el juego está en marcha (si no, salir del `update` inmediatamente porque se ha acabado la ronda mientras estamos comprobando colisiones)
2. el asteroide y el fighter son activos (i.e., sus `isActive()` devuelve `true`)
3. hay colisión entre el asteroide y el caza (usar `Collisions::collidesWithRotation`)

Si cumplen todas las condiciones, envía a todos un `msg::FighterAsteroidCollision` con valores correspondientes.

`BulletsAsteroidsCollision` (física)**:** Es responsable de comprobar colisiones entre las balas y los asteroides. Tiene 2 atributos:

```
const vector<Asteroid*>* asteroids_;
const vector<Bullet*>* bullets_;
```

El comportamiento es muy parecido a lo de `FighterAsteroidCollision`: tiene que comprobar las mismas condiciones para *cada bala y asteroide* y si cumplen todas las condiciones envía a todos un `msg::BulletAsteroidCollision` con valores correspondientes.

Para probar lo que has hecho en esta parte, añade un atributo de tipo `Bullets` en la clase `AsteroidsGame` y añádelo a la lista de actores en el método `initGame()`. Tienes que ver el marcador, las vidas y el mensaje **"`Press ENTER to Start a New Game`".**

Parte 5: Reaccionar a mensajes

Para que todos los `Containters` que hemos definido en los apartados 1-4 reaccionen al recibir mensajes, sobrescribe el método `receive` en todos. Recuerda que la primera línea del método `recieve` tiene que ser `Container::receive(...)` porque `receive` de la clase `Container` es el responsable de reenviar los mensajes a todos los componentes del `Container` (en general no es necesario que sea la primera línea, eso depende de la funcionalidad que queremos).

Los componentes `FighterAsteroidCollision` y `BulletsAsteroidsCollision` van a recibir mensajes tambíen, para cada uno sobreescribe el método `recieve` (de la clase `Component`).

Los métodos `recieve`, en principio, tienen que tener un `switch` sobre el valor de **`msg.type_`** para reaccionar depende del tipo de mensaje.

Las clases `Fighter`, `Asteroids` y `Bullets` tienen que estar desactivados cuando el juego está parado (el `GameManger` siempre está activado), añade a sus constructoras la instrucción `setActive(false)` -- no olvidar de quitar todo el código que has añadido para probar esas clases.

Asigna un identificador a cada `Container`, llamando al método `setId(...)` en la constructora. Los identificadores se pueden definir en el archivo `Mensages_decl.h` dentro el `enum ObjectId`. No vamos a usarlos en esa práctica, todos los mensajes van a ser de tipo `msg::Broadcast`.

Para poder recibir mensajes, registra todos los `Containters` como observadores en el método `initGame()` de la clase `AsteroidsGame, p.ej., añadir addObserver(&gameManger_), etc.`

A continuación detallamos los mensajes al que tiene que reaccionar.

`Fighter:`

1. **`msg::GAME_START:`** envía un `msg::FighterInfo` a todos para pasar una referencia a sí mismo a todos los interesados (los componentes que comprueban colisiones).
2. **`msg::ROUND_START:`** pone el caza en su estado inicial (i.e., en el centro de la pantalla con velocidad `0`) y activa el `Container (i.e., setActive(true))`
3. **`msg::ROUND_OVER:`** desactiva el `Container (i.e., setActive(false))`

`Asteroids:`

1. **`msg::GAME_START:`** envía un `msg::AsteroidsInfo` a todos para pasar una referencia al vector de asteroides (ver `getAllObjects()` de `GameObjectPool`) a todos los interesados (los componentes que comprueban colisiones).

2. **msg::ROUND_START:** añade `10` asteroides a la pantalla. La posición de cada asteroide es aleatoria, pero tiene que estar en los bordes de la pantalla. La velocidad de cada asteroide es aleatoria (mejor lenta), p.ej., suponiendo que `p` es la posición del asteroide, se puede usar el siguiente código para que todos los asteroides vayan hacia el centro:

```
Vector2D c = Vector2D(getGame()->getWindowWidth() / 2, getGame()->getWindowHeight() / 2);
Vector2D v = (c - p).normalize() * (r->nextInt(1, 10) / 20.0);
```

   donde `r` es un generador de números aleatorios (ver la clase `ServiceLocator`).

3. **msg::ROUND_OVER:** desactiva todos los asteroides (ver el método `deactiveAllObjects()` de `GameObjectPool`). Desactiva el `Container` (i.e., `setActive(false)`).

4. **msg::BULLET_ASTEROID_COLLISION:** suponemos que el asteroide que ha chocado con una bala es **`X`:**

   a. desactiva el asteroide `X`.

   b. Envía a todos un `msg::AsteroidDestroyed` con el número de puntos ganados, p.ej., 4 menos el número de generación.

   c. si el número de generación de **`X`** es mayor de 1, genera 2 asteroides nuevos (usando el object pool). El número de generación de cada uno es como el de **`X`** menos 1, el tamaño es el 75% de tamaño de **`X`,** el vector de velocidad $\mathbf{V}_i$ de *i-esimo* nuevo asteroide es como el de **`X`** multiplicando por `1.1` para que se más rápido y girado en **`i*30`** grados para que vaya en otra dirección, la posición es como la de **`X`** más $\mathbf{V}_i$ -- puedes probar varias fórmulas para la posición y la velocidad.

   d. si el número de asteroides en la pantalla es cero, envía un `msg::Message` con el tipo `msg::NO_MORE_ASTEROIDS` a todos.

   e. Reproduce sonido de explosión, p.ej., usando `Resources::Explosion.`

Bullets:

1. **msg::GAME_START:** envía un `msg::BulletsInfoMsg` a todos para pasar una referencia al vector de balas (ver `getAllObjects()` del `GameObjectPool`) a todos los interesados (los componentes que comprueben colisiones).

2. **msg::ROUND_START:** activa el `Container (i.e., setActive(true))`

3. **msg::ROUND_OVER:** desactiva todas las balas (ver el método `deactiveAllObjects()` de `GameObjectPool`). Desactivar el `Container` (i.e., `setActive(false)`).

4. **msg::BULLET_ASTEROID_COLLISION:** desactiva la bala correspondiente.

5. **msg::FIGHTER_SHOOT:** suponiendo que el mensaje pide disparar una bala con posición **P** y dirección **D**, genera una bala (usando el object pool) con tamaño `1x5`, posición **P**-`Vector2D(width/2,height)`, velocidad **D*5** y rotación `Vector2D(0,-1).angle(`**D**`)`. Reproduce un sonido de disparo, p.ej., usando `Resources::GunShot`.

GameManager:

1. **msg::GAME_START:** pone `_gameOver` a `false`, `winner_` a 0 y `lives_` a `maxLives_`.
2. **msg::ROUND_START:** pone `_running` a `true` y empieza a reproducir música de fondo, p.ej., usando `Resources::ImperialMarch`
3. **msg::ASTEROID_DESTROYED:** sumar el número de puntos correspondientes al marcador.
4. **msg::NO_MORE_ASTEROIDS:** pone `_running` a `false`, `_gameOver` a `true`, `_winner` a 1, para la música de fondo, envía a todos un `msg::Message` con el tipo `msg::ROUND_OVER`, envía un `msg::Message` con el tipo `msg::GAME_OVER`
5. **msg::FIGHTER_ASTEROID_COLLISION:** reproduce sonido de explosión, para la música de fondo, pone `_running` a `false`, quita una vida al caza, envía a todos un `msg::Message` con el tipo `msg::ROUND_OVER`, y si no hay más vidas pone `gameOver_` a `true`, `_winner` a 2 y envía a todos un `msg::Message` con el tipo `msg::GAME_OVER`.

FighterAsteroidCollision:

1. **msg::ASTEROIDS_INFO:** modifica el atributo `_ateroids` con la información correspondiente.
2. **msg::FIGHTER_INFO:** modifica el atributo `_fighter con` la información correspondiente.

BulletsAsteroidsCollision:

1. **msg::ASTEROIDS_INFO:** modifica el atributo `_ateroids` con la información correspondiente.
2. **msg::BULLETS_INFO:** modifica el atributo `_bullets` con la información correspondiente.