TP Videojuegos 2
Práctica 1

Fecha Límite: 11/03/2024 a las 17:00.

En esta práctica vamos a desarrollar una variación del juego clásico Asteroids.

Antes de empezar

Primero: leer todo el enunciado antes de empezar.

Descarga la plantilla de proyecto **SDL** desde el campus virtual y verifica que funcione correctamente. Descarga **"ecs_2.zip"** o **"ecs_3.zip"** y añade la carpeta **ecs** al proyecto directamente en la carpeta **src.** Añade uno de los juegos de ejemplo que vienen con **"ecs_2.zip"** o **"ecs_3.zip" al proyecto,** copiando las carpetas **game** y **components** y el archivo **main.cpp** directamente en la carpeta **src.** La estructura de la carpeta **src** tiene que quedar así:

```
src/components
src/game src/ecs
src/sdlutils
src/json
src/utils
src/main.cpp
```

Verifica que el juego funcione correctamente. Ahora puedes usar este proyecto como la base para la práctica. Los componentes que desarrollamos tienen que ir en la carpeta **src/components** y el resto de clases en **src/game**. *Es recomendable usar filtros para organizar los archivos*.

Antes de entregar, debes borrar todo lo no pertenece a tu práctica de las carpetas **resources** y **src/components** y limpiar el contenido de **game/ecs_defs.h**.

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 a todos los asteroides (gana). Al comienzo de cada ronda colocamos **10** asteroides aleatoriamente en los bordes de la ventana y cada **5** segundos añadimos un asteroide.

Cuando se crea un asteroide elegimos un vector de velocidad aleatorio para avanzar en la dirección de la zona central, pero algunos asteroides pueden modificar este comportamiento usando componentes: (1) siempre cambia su vector de velocidad para que siga al caza; (2) cambia su vector de velocidad para ir hacia un destino aleatorio y cuando llega cambia su destino de manera aleatoria.

Los asteroides tienen un *contador de generaciones* con valor inicial aleatorio entre **1** y **3**. Cuando el caza destruye un asteroide **"a"**, el asteroide debe desaparecer de la ventana, y si su contador de generaciones es positivo creamos **2** asteroides nuevos. El contador de generación de cada asteroide nuevo es como el de **"a"** menos uno, su posición es cercana a la de **"a"** y su vector de velocidad se obtiene a partir del vector de velocidad de **"a"** (más adelante explicaremos cómo calcular la posición y la velocidad).

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. Entre las rondas y al terminar una partida, se debe mostrar mensajes adecuados y pedir al jugador que pulse alguna tecla para continuar.

Se debe reproducir sonidos correspondientes cuando el caza dispara una bala, cuando el caza acelera, cuando un asteroide explota, cuando el caza choca con un asteroide, etc. Archivos de sonido e imágenes están disponibles en el campus virtual (pero se pueden usar otros).

Ver el video para tener una idea de lo que tienes que implementar. En los detalles a continuación, los componentes y las entidades son una recomendación, se pueden usar otros componentes/entidades pero hay que justificarlo durante la corrección.

Requisitos Generales (obligatorios)

1. Crea un archivo con el nombre **"resources/config/asteroid.resources.json"** y úsalo para definir referencias a todos los recursos necesarios (imágenes, sonidos, etc). Si este archivo ya existe con la versión de **resources.zip** que está en el campus, actualízalo según la necesidades de tu programa.
2. Hay que usar la versión **"ecs_2.zip"** o **"ecs_3.zip"** (descargar desde el campus virtual), no se puede usar otra versión. Es recomendable usar **"ecs_3.zip"**.
3. La última versión de **resources.zip** (en el campus virtual) incluye todos los archivos de imágenes y sonidos necesarios para esta práctica.

4. La estructura de las carpetas tiene que ser parecida a la que uso en los ejemplos. Usando la plantilla de proyecto **TPV2**, hay que tener **3** carpetas adicionales: **src/ecs**, **src/components** y **src/game**). Poner los componentes en la carpeta **components** y la clase **Game** y el archivo **ecs_def.h** en la carpeta **game**. El archivo **main.cpp** tiene que estar directamente en la carpeta **src**, es decir al mismo nivel de las otras carpetas.
5. Hay que inicializar todos los atributos en las constructoras (en el mismo orden de la declaración).
6. Al usar la directiva **"#include"**, escribe el nombre del archivo *respetando minusculas y mayusculas*. El problema es que `Windows` no distingue entre mayúsculas y minúsculas pero otros sistemas operativos sí.
7. Usar **assert** para comprobar la validez de valores.
8. Usar el formateo (indentation) automático de **Visual Studio**, así el código es mucho más fácil de leer.
9. Entregar un **zip** que incluye solo las carpetas **TPV2/src** y **TPV2/resource**. Debes borrar todo lo no pertenece a tu práctica de las carpetas **resources** y **src/components** y limpiar el contenido de **game/ecs_defs.h**.

Requisitos Opcionales

1. Introduce la posibilidad de configurar el juego desde un archivo de configuración de formato **JSON**, p.ej., **"resources/config/asteroid.config.json"**, parecido a lo que hemos hecho para el juego **Ping Pong** en clase. Puedes elegir qué valores se pueden configurar, p.ej., el número de asteroides inicial, la frecuencia de generación de asteroides, la velocidad máxima del caza, el grado de rotación del caza, el número de asteroides que generamos al destruir un asteroide, la velocidad de las balas, etc.
2. ¡Puedes inventar y añadir al juego cualquier funcionalidad adicional que quieras!

Entidad del caza

Es una entidad para representar el caza con los siguientes componentes:

1. **Transform:** mantiene las características físicas y mueve la entidad sumando la velocidad a la posición.
2. **DeAcceleration:** desacelera el caza automáticamente en cada iteración del juego (p.ej., multiplicando la velocidad por **0.995f**), y si la magnitud del vector de velocidad llega a ser menor de un límite muy pequeño (p.ej, **0.05f**) cambia el vector de velocidad a **(0.0f,0.0f)**.
3. **Image:** dibuja el caza usando **"fighter.png"** (o cualquier otra imagen que quieras).
4. **Health:** mantiene el número de vidas del caza y las dibuja en alguna parte de la ventana, p.ej., mostrando **"heart.png"** tantas veces como el número de vidas en la parte superior de la ventana. Además, tiene métodos para quitar una vida, resetear las vidas, consultar el número de vidas actual, etc. El caza tiene **3** vidas al principio de cada partida.

5. **`FighterCtrl:`** controla los movimientos del caza. Pulsando ← o → gira **`5.0f`** grados en la dirección correspondiente y pulsando ↑ acelera *(más información en el apéndice)*. Al acelerar hay que reproducir el sonido **"`thrust.wav`".** Recuerda que la desaceleración es automática, no se maneja en este componente.

   Recuerda que para comprobar si el usuario ha pulsado una tecla, puedes usar la condición compuesta **`(ihdrl.keyDownEvent() && ihdrl.isKeyDown(SDL_SCANCODE_RIGHT))`** o la condición **`ihdrl.isKeyDown(SDL_SCANCODE_RIGHT)`**. Usa la primera opción para obtener un movimiento más fluido cuando la tecla queda pulsada continuamente, porque **`SDL`** genera eventos **`SDL_KEYDOWN`** cada unos milisegundos (en este caso a lo mejor quieres reducir el ángulo de giro de **`5.0f`** a **`1.0f`**).

6. **`Gun:`** pulsando **`S`** añade una bala al juego y reproduce el sonido **"`fire.wav`"**. Se puede disparar sólo una bala cada **`0.25sec`** (usar **`sdlutils().virtualTimer()`** para consultar el tiempo actual). Las balas no van a ser entidades, este componente tiene que mantener un pool de balas (con un número máximo, algo como `20`) y re-usarlas. Ver el apartado Más información sobre el componente GUN.

7. **`ShowAtOpposieSide:`** cuando el caza sale de un borde (por completo) empieza a aparecer en el borde opuesto.

Entidad de un asteroide

Es una entidad para representar un asteroide (pertenece al grupo **"`ecs::grp::ASTEROIDS`"**) con los siguientes componentes:

1. **`Transform:`** mantiene las características físicas y mueve la entidad sumando la velocidad a la posición.
2. **`ImageWithFrames:`** dibuja el asteroide usando una de las imágenes **"`asteroids.png`"** o **"`asteroids_gold.png`"**. Hay que cambiar el frame cada **`50ms`** para conseguir el efecto de la rotación (y no cambiando la rotación en el componente **`Transform`**). En la primera versión puedes usar el componente **`Image`** con la imagen star.png por ejemplo, y cuando tienes todo funcionando puedes implementar este componente.
3. **`ShowAtOpposieSide:`** cuando el asteroide sale de un borde (por completo) empieza a aparecer en el borde opuesto.
4. **`Generations:`** mantiene el número de generaciones del asteroide, y tiene métodos para consultarlo, cambiarlo, etc.
5. **`Follow:`** actualiza el vector de velocidad para que siga al caza. Si **"`v`"** es el vector de velocidad del asteroide, **"`p`"** su posición, y **"`q`"** la posición del caza, se puede conseguir este efecto cambiando el vector de velocidad del asteroide a **"`v.rotate(v.angle(q-p) > 0 ? 1.0f : -1.0f)`"**. Es un componente opcional, no todos los asteroides lo tienen, depende de lo que elegimos en el momento de creación (ver el apartado Cómo crear un asteroide).
6. **`TowardDestination`**: al principio elige un destino aleatorio y actualiza el vector de velocidad para que el asteroide vaya hacia ese destino, y cuando llega al destino (distancia menor de **`10.0f`** por ejemplo) cambia el destino a otro aleatorio. Para conseguir este efecto, si **"`v`"** es el vector de velocidad del asteroide, **"`p`"** su posición, y **"`q`"** es el destino, cambia el vector de

velocidad del asteroide a **"v.rotate(v.angle(q-p))"**. Es un componente opcional, no todos los asteroides lo tienen, depende de lo que elegimos en el momento de creación (ver el apartado Cómo crear un asteroide).

Fachada para los asteroids

Usamos el patrón de diseño **Façade** para simplificar el trabajo con los asteroides. Primero definimos esta clase abstract (en la carpeta **game**):

```
class AsteroidsFacade {
public:
	AsteroidsFacade() {
	}
	virtual ~AsteroidsFacade() {
	}
	virtual void create_asteroids(int n) = 0;
	virtual void remove_all_asteroids() = 0;
	virtual void split_astroid(Entity *a) = 0;
};
```

Creamos una clase **AsteroidsUtils** (en la carpeta **game**) que hereda de la clase **AsteroidsFacade.** Sus métodos tienen el siguiente comportamiento:

1. **create_asteroids(int n)**: añade **n** asteroides al juego (ver el apartado Cómo crear un asteroide).
2. **remove_all_asteroids():** desactiva todos los asteroides actuales en el juego (i.e., marcando las entidades como muertas y llamando a **refsresh()** del manager). No se puede mantener una lista local de asteroides, hay que usar la lista del grupo correspondiente del **Manager**.
3. **split_astroid(Entity *a):** recibe una entidad (si usas **ecs_3.zip** el tipo seria **entity_t**) representando un asteroide que haya chocado con una bala, lo desactiva y genera otros **2** asteroides dependiendo de su número de generaciones (ver el apartado Cómo dividir un asteroide).

Fachada para el caza

Usamos el patrón de diseño **Façade** para simplificar el trabajo con el caza. Primero definimos esta clase abstract (en la carpeta **game**):

```
class FighterFacade {
public:
	FighterFacade() {}
	virtual ~FighterFacade() {}
	void virtual create_fighter() = 0;
```

```
    void virtual reset_fighter() = 0;
    void virtual reset_lives() = 0;
    int virtual update_lives(int n) = 0;
};
```

Creamos una clase **FighterUtils** (en la carpeta **game**) que hereda de la clase **FighterFacade.** Sus métodos tienen el siguiente comportamiento:

1. **create_fighter()**: crea la entidad del fighter con sus componentes, lo coloca en el centro de la ventana (el centro del fighter tiene que estar en el centro de la ventana), e inicializa las vidas a **3**.
2. **reset_fighter()**: resetea la posición del caza y sus componente **GUN** (para desactivar la balas activas si la hay - ver el apartado Más información sobre el componente GUN).
3. **reset_lives()**: vuelve a poner la vidas a **3**.
4. **update_lives(int n)**: añadir **n** vidas al caza (lo usamos con **-1** para quitar una vida).

Estados del juego

Usamos el patrón de diseño **State** para poder cambiar de un estado de juego a otro fácilmente. Primero definimos la siguiente clase abstracta:

```
class GameState {
public:
    GameState() {}
    virtual ~GameState() {}
    virtual void enter() = 0;
    virtual void leave() = 0;
    virtual void update() = 0;
};
```

La clase **Game** llamará a **enter()** cuando cambie al estado, llamará a **leave()** cuando sale del estado, y a **update()** desde el bucle principal (este update se encarga de actualizar las entidades si es necesario y dibujarlas – es como el cuerpo del bucle principal – ver el apartado La clase Game).

Vamos a tener **5** estados (clases que heredan de **GameState**):

1. **NewGameState**: en su método **update()**, muestra el mensaje **"press any key to start a new game",** y cuando el usuario pulsa cualquier tecla resetea las vidas del caza (usando **reset_lives()** de la fachada correspondiente) y cambia al estado **NewRoundState**.
2. **NewRoundState**: en su método **update()**, muestra el mensaje **"press ENTER to start the round"** y cuando el usuario pulsa la tecla **ENTER** resetea la posición del caza (usando **reset_fighter()** de la fachada correspondiente), borra todos lo asteroides actuales (usando **remove_all_asteroids()** de la fachada correspondiente), añade **10** asteroides al juego

(usando **`create_asteroids(10)`** de la fachada correspondiente), y cambia al estado **`RunningState`**.

3. **`RunningState`**: en su método **`update()`**:
   a. Si hay **`0`** asteroides cambia al estado **`GameOverState`** y sale del método.
   b. Si el usuario pulsa la tecla **`P`** cambia al estado **`PauseState`** y sale del método.
   c. Actualizar la entidad del caza y las entidades de los asteroides (usando sus **`update`**).
   d. Comprobar colisiones entre caza, asteroides y balas. Para eso implementa un método que comprueba choques (usando el método **`Collisions::collidesWithRotation`**) para cada *asteroide vivo* **`X`** de la siguiente formas:
      i. Si hay un choque entre **`X`** y el caza, quita una vida al caza y si le quedan más vidas cambia al estado **`NewRoundState`** y si no le quedan vidas cambia al estado **`GameOverState`**. Si hay choque no tiene que seguir comprobando.
      ii. Si hay choque entre **`X`** y una bala, divide **`X`** (usando **`split_astroide(X)`** de la fachada correspondiente) y marca la bala como no usada (ver el apartado Más información sobre el componente GUN).

      Para recorrer a todos los asteroides mejor usar un *bucle **`for`** con índice desde **`0`** hasta número actual de asteroides*, y no usar bucle **`foreach`**, porque podemos añadir más asteroides al juego cuando chocan con balas.
   e. Dibujar la entidad del caza y las entidades de los asteroides (usando sus **`render`**).
   f. Quitar las entidades muertas usando el método **`refresh()`** del manager.
   g. Añadir un asteroide nuevo al juego si han pasado 5 segundos desde la última vez que hemos añadido un en este punto (usar el timer virtual **`sdlutils().virtualTimer()`**).
4. **`PausedState`**: en su método **`enter()`**, tiene que *parar el tiempo virtual* usando el timer virtual **`sdlutils().virtualTimer()`** y en su método **`leave()`** reanudarlo. En su método **`update()`** muestra el mensaje **"press any key to resume the game",** y cuando el usuario pulsa cualquier tecla cambia al estado **`RunningState`**.
5. **`GameOverState`**: en su método **`enter()`** decide qué mensaje mostrar (**`"Game Over Loser! Press ENTER to continue."`** o **`"Game Over Champion! Press ENTER to continue."`**) depende de que si hay **`0`** asteroides en el juego o más (si hay **`0`** el caza habrá eliminado a todos, y sí no el caza habrá muerto), y en su método **`update()`** muestra el mensaje y cuando el usuario pulsa cualquier tecla cambia al estado **`NewGameState`**.

La clase Game

Añadir un método que devuelve el **`Manager`** porque lo vamos a necesitar desde otras partes de juego, como los estados, las fachadas, etc.

Hacer la clase **Game** Singleton para garantizar que hay solo una instancia y tener acceso global a ella. Para eso puedes usar **"utils/Singlton"**. No olvidar de cambiar la constructora de la clase **Game** a **private**, y cambiar el uso de **Game** en **main.cpp**.

Si no quieres usar Singleton, tendrás que pasar una referencia a la instancia de la clase **Game** a los estados y las fachadas.

Para gestionar los estados, podemos añadir a la clase **Game** algo parecido al siguiente código:

```
enum State {
	RUNNING, PAUSED, NEWGAME, NEWROUND, GAMEOVER
};

inline void setState(State s) {
	GameState *new_state = nullptr;
	switch (s) {
	case RUNNING:
		new_state = runing_state_;
		break;
	case PAUSED:
		new_state = paused_state_;
		break;
	case NEWGAME:
		new_state = newgame_state_;
		break;
	case NEWROUND:
		new_state = newround_state_;
		break;
	case GAMEOVER:
		new_state = gameover_state_;
		break;
	default:
		break;
	}
	current_state_->leave();
	current_state_ = new_state;
	current_state_->enter();
}
private:
...
GameState *current_state_;
GameState *paused_state_;
```

```
    GameState *runing_state_;
    GameState *newgame_state_;
    GameState *newround_state_;
    GameState *gameover_state_;
};
```

En el método **init()** la clase **Game**: inicializar **SDLUtil** usando **SDLUtils::init("ASTEROIDS", 800, 600,"resources/config/asteroids.resources.json")**, crea las fachadas, crea el caza usando la fachada correspondiente, y crea los estados.

En el método **start()**, el bucle principal tiene que ser algo parecido al siguiente:

```
bool exit = false;
auto &ihdlr = ih();
while (!exit) {
    Uint32 startTime = sdlutils().currRealTime();
    ihdlr.refresh();
    if (ihdlr.isKeyDown(SDL_SCANCODE_ESCAPE)) {
        exit = true;
        continue;
    }
    current_state->update();
    Uint32 frameTime = sdlutils().currRealTime() - startTime;
    if (frameTime < 10)
        SDL_Delay(10 - frameTime);
}
```

APÉNDICE

Cómo acelerar el caza

El movimiento del caza está basado en empujones, eso hace que el juego sea más difícil. Además, como hemos visto antes, la desaceleración se hace de manera automática, el jugador no puede desacelerar. Recuerda que el caza puede mirar hacías una parte y

Para acelerar, suponiendo que **"vel"** es el vector de velocidad actual y **"r"** es la rotación, el nuevo vector de velocidad **"new_vel"** sería **"vel+Vector2D(0,-1).rotate(r)*thrust"** donde **"thrust"** es el factor de empuje (prueba con distintos valores, p.ej. **0.2f** o **0.1f**). Además, si la magnitud de **"new_vel"** supera un límite **"speed_limit"** (usa p.ej., **3.0f**) modifícalo para que tenga la magnitud igual a **"speedLimit"**, es decir modifícalo a **"new_vel.normalize()*speed_limit"**.

Más información sobre el componente GUN

Puedes usar la siguiente clase como base para tu implementación del componente **GUN**:

```
class Gun: public ecs::Component {
public:
	...
	struct Bullet {
		bool used = false;
		Vector2D pos;
		Vector2D vel;
		int width;
		int height;
		float rot;
	};
	constexpr static auto max_bullets = 20;
	typedef std::array<Bullet, max_bullets> bullets_array;
	typedef bullets_array::iterator iterator;

	void reset();
	iterator begin() {
		return bullets_.begin();
	}
	iterator end() {
		return bullets_.end();
	}
	...
private:
	void shoot(Vector2D p, Vector2D v, int width, int height, float r);
	bullets_array bullets_;
	...
};
```

El array **bullets_** representa un pool de balas, donde cada bala es de tipo **Bullet** que incluye la características físicas de la bala y un atributo **used** que indica si la bala está en uso.

El método **reset()** debe marcar las balas como no usadas. El método **render()** debe dibujar todas las balas usadas (usando por ejemplo **"fire.png"**). El método **update()** debe mover todas la balas usadas (añade velocidad a posición) y añadir una bala al juego si el usuario pulsa **S** usando el método **shoot(...)**. Para calcular la posición, velocidad y rotación de la bala se puede usar el siguiente código donde **"p"** la posición del caza, **"v"** su vector de velocidad, **"r"** su rotación, **"w"** su anchura, y **"h"** su altura:

```
int bw = 5;
int bh = 20;
Vector2D c = p + Vector2D(w / 2.0f, h / 2.0f);
Vector2D bp = c - Vector2D(bw / 2, h / 2.0f + 5.0f +bh).rotate(r) - Vector2D(bw / 2, bh / 2);
Vector2D bv = Vector2D(0, -1).rotate(r) * (v.magnitude() + 5.0f);
```

```
float br = Vector2D(0, -1).angle(bv);
shoot(bp, bd, bw, bh, br);
```

Recuerdas que se puede disparar sólo una bala cada **0.25sec** (usar **sdlutils().virtualTimed()** para consultar el tiempo actual).

El método **shoot(...)** busca una bala no usada en el array **bullets_**, y si encuentra una la marca como usada y la inicializa con la información correspondiente. Para que la búsqueda de una bala no usada sea eficiente, usa una búsqueda circular empezando desde la posición siguiente a la última bala usada (en la anterior llamada a **shoot**).

Los métodos **begin()** y **end()** nos permiten recorrer al array **bullets_** desde fuera de la clase para comprobar colisiones usando iteradores. Por ejemplo, si **fighterGUN** es un puntero al componente correspondiente podemos usar:

```
for (Gun::Bullet &b : *fighterGUN)
    if (b.used) { ... }
```

Cómo crear un asteroide

Al crear un nuevo asteroide, hay que asignarle una posición y velocidad aleatorias. Además, hay que elegir el vector de velocidad de tal manera que el asteroide se mueve hacia la zona central de la ventana.

Primero elegimos su posición **"p"** de manera aleatoria en los bordes de la ventana. Después elegimos una posición aleatoria **"c"** en la zona central usando **"c=(cx,cy)+(rx,ry)"** donde **"(cx,cy)"** es el centro de la ventana y **"rx"** y **"ry"** son números aleatorios entre **-100** y **100**. El vector de velocidad **v** sería:

```
float speed = sdlutils().rand().nextInt(1,10)/10.0f;
Vector2D v = (c-p).normalize()*speed;
```

El número de generaciones del asteroide es un número entero aleatorio entre **1** y **3**. La anchura y altura del asteroide dependen de su número de generaciones, p.ej., **10.0f+5.0f*g** donde **"g"** es el número de generaciones.

Además, de manera aleatoria, decide si quieres añadir uno de los componentes **Follow** o **TowardsDestination**. Puedes usar imágenes distintas para cada configuración de asteroides.

Cómo dividir un asteroide

Al destruir un asteroide **"a"** la idea es desactivarlo y crear otros **2** con el número de generaciones como **"a"** menos uno (si su número de generaciones de es **0** no se genera nada). La posición de cada nuevo asteroide tiene que ser cercana al de "**a**" y tiene que moverse en una dirección aleatoria.

Suponiendo que **"p"** y **"v"** son la velocidad y la posición de "**a**", **"w"** su anchura, **"h"** su altura, se puede usar el siguiente código para calcular la posición y la velocidad de cada asteroide nuevo:

```
int r = sdlutils().rand().nextInt(0,360);
Vector2D pos = p + v.rotate(r) * 2 * std::max(w,h).
Vector2D vel = v.rotate(r) * 1.1f
```

Recuerda que la anchura y altura del nuevo asteroide dependen de su número de generaciones.