# Práctica 1: Asteroids 1.0

## Tecnología de la Programación de Videojuegos. UCM

El objetivo de esta práctica es implementar en C++/SDL una versión simplificada del juego *Asteroids* lanzado por Atari en 1979 (véase `https://es.wikipedia.org/wiki/Asteroids`). Se utilizará la librería SDL para manejar toda la entrada/salida del juego y se introducirá el uso de una arquitectura basada en Entidades y Componentes. Es recomendable mantener el diseño de estados y máquina de estados utilizado en las prácticas de TPV1. A continuación, se detalla la funcionalidad principal que se busca en esta práctica:

- El juego cuenta con dos actores principales: el caza y los asteroides.
- El objetivo del caza es **destruir los asteroides disparándoles**.
- El caza tiene tres **vidas** y cuando choca con un asteroide explota y pierde una vida; si tiene más vidas, el jugador puede jugar otra **ronda**. El número de vidas del caza se debe mostrar en la esquina superior izquierda.
- El caza está equipado con un arma que puede **disparar**.
- Al comienzo de cada ronda colocamos **10 asteroides** aleatoriamente en los bordes de la ventana, con velocidad aleatoria. Los cazas de cualquier ronda anterior, si la hubiera, desaparecen antes de empezar la nueva ronda.
- Algunos asteroides **actualizan su vector de velocidad** continuamente para avanzar en la dirección del caza.
- Cuando el caza destruye un asteroide este **debe desaparecer** de la ventana.
- Los asteroides tienen un **contador de generaciones** con valor inicial aleatorio entre 1 y 3. Si se destruye un asteroide (“padre”) con generación mayor que 1, **se generan dos asteroides** nuevos (“hijos”) con una generación menos que la del padre. La posición de los asteroides hijos es cercana a la del padre y su vector de velocidad se obtiene a partir del vector de velocidad del padre.
- **El juego termina** cuando el caza no tiene más vidas (pierde) o destruye a todos los asteroides (gana).
- Antes de iniciar la partida, entre las rondas, al terminar una partida o al pausar una partida, se deben mostrar **mensajes adecuados** y pedir al jugador que pulse SPACE para continuar. El juego **se podrá pausar** en cualquier momento pulsando SPACE.
- 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. Todos los archivos de sonido e imágenes necesarios para esta práctica están disponibles en el campus virtual (pero su uso no es obligatorio).

## Detalles de implementación.

### Diseño de clases y entidades

1. **Entidad Fighter (Caza):** Es una entidad para representar el caza con los siguientes componentes:
   1.1. **Transform:** mantiene las características físicas y mueve la entidad sumando la velocidad a la posición.
   1.2. **DeAcceleration:** desacelera el caza automáticamente en cada iteración del juego (p.ej., multiplicando la velocidad por 0.995), y si la magnitud del vector de velocidad llega a ser menor de un límite muy pequeño (p.ej, 0.05) cambia el vector de velocidad a (0 ,0).
   1.3. **Image:** dibuja el caza usando `fighter.png`.
   1.4. **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.
   1.5. **FighterCtrl:** controla los movimientos del caza. Pulsando ← o → gira 5.0f grados en la dirección correspondiente y pulsando ↑ acelera. Al acelerar hay que reproducir el sonido `thrust.wav`. Recuerda que la desaceleración es automática, no se maneja en este componente.
   1.6. **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().currTime()` para consultar el tiempo actual).
   1.7. **ShowAtOpposideSide:** cuando el caza sale de un borde (por completo) empieza a aparecer en el borde opuesto.
2. **Entidad Asteroid (Asteroide):** Es una entidad para representar un asteroide (pertenece al grupo `_grp_ASTEROIDS`) con los siguientes componentes (hay dos tipos de asteroides, `A` y `B`):
   2.1. **Transform:** mantiene las características físicas y mueve la entidad sumando la velocidad a la posición.
   2.2. **FramedImage:** dibuja el asteroide usando la imagen `asteroids.png` para el tipo `A` y `asteroids_gold.png` para el tipo `B`. Hay que cambiar el *frame* cada 50ms para conseguir el efecto de la rotación (y no cambiando la rotación en el componente `Transform`).
   2.3. **ShowAtOpposideSide:** cuando el asteroide sale de un borde (por completo) empieza a aparecer en el borde opuesto.
   2.4. **Generations:** mantiene el número de generaciones del asteroide, y tiene métodos para consultarlo, cambiarlo, etc.
   2.5. **Follow:** actualiza el vector de velocidad para que siga al caza. Se usa sólo para asteroides de tipo B.
3. **Entidad Bullet (Bala):** Es una entidad para representar una bala (pertenece al grupo `_grp_BULLETS`) con los siguientes componentes:
   3.1. **Transform:** mantiene las características físicas y mueve la entidad sumando la velocidad a la posición.

3.2. **Image:** dibuja la bala usando la imagen `fire.png`
3.3. **DisableOnExit:** desactiva la bala, es decir llama a su `isAlive(false)`, cuando sale por completo de la ventana.

4. **AsteroidsManager:** Crear una clase `AsteroidsManager` (en la carpeta `game`) para manejar la creación y destrucción de asteroides. La clase tendrá que tener una referencia al `Manager` (pasarlo en la constructora) y un atributo para contador para el número de asteroides actuales. Además, tiene que tener los siguientes métodos:
   4.1. **createAsteroids(int n)**: añade `n` asteroides al juego. Cuando genera un asteroide, con probabilidad de 70% genera uno de tipo `A` y el 30% uno de tipo `B` (se puede decidir usando la condición `sdlutils().rand().nextInt(0,10)<3`). No pueden existir más de 30 asteroides a la vez en la pantalla.
   4.2. **addAsteroidFrequently():** genera un asteroide nuevo cada 5 segundos (aparte de los 10 al principio de cada ronda) llamando a `createAsteroids(1)`.
   4.3. **destroyAllAsteroids():** desactiva todos los asteroides actuales en el juego (llama a `setAlive(false)`) de cada uno. No se puede mantener una lista de asteroides, usa la lista de grupo correspondiente del `Manager`.
   4.4. **onCollision(Entity *a):** recibe una entidad representando un asteroide que haya chocado con una bala, lo desactiva y genera otros 2 asteroides dependiendo de su número de generaciones. Recuerda que no pueden existir más de 30 asteroides a la vez en la pantalla.

## Colisiones

Para comprobar colisiones implementa un método `checkCollisison()` que comprueba choques con todos los asteroide activos, usando el método `Collisions::collidesWithRotation`, de la siguiente manera:

a) Si choca con el caza, desactiva todos los asteroides (usando `destroyAllAsteroids()` del `AsteroidsManager`), desactiva todas las balas, quita una vida al caza, marca el juego como `PAUSED` o `GAMEOVER` (depende de si quedan vidas) y pon al caza en el centro de la ventana con velocidad cero y rotación cero.
b) Si choca con una bala activa, desactiva la bala y llama a `onCollison` del `AsteroidsManager` pasándole el asteroide para destruirlo, etc**.** Si no hay más asteroides, gana el caza. Se puede implementar en la clase `Game` o en una clase nueva `CollisionsManager` (ponerla en la carpeta `game`). Si lo haces en una clase separada necesitas pasarle referencias al `Manager` y al `AsteroidsManager`.

## Aceleración del caza

El movimiento del caza está basado en empujones, eso hace que el juego sea más difícil. Además, la desaceleración se hace de manera automática, el jugador no puede desacelerar. Para acelerar, suponiendo que `vel` es el vector de velocidad actual y `r` es la rotación, el nuevo vector de velocidad `newVel` sería `vel+Vector2D(0,-`

`1).rotate(r)*thrust` donde `thrust` es el factor de empuje (usa p.ej. 0.2f). Además, si la magnitud de `newVel` supera un límite `speedLimit` (usa p.ej., 3.0f) modifícalo para que tenga la magnitud igual a `speedLimit`, es decir modifícalo a `newVel.normalize()*speedLimit`.

## Posición, dirección, rotación, y tamaño de la bala

Sea `pos` la posición del caza, `vel` su vector de velocidad, `r` su rotación, `w` su anchura, y `h` su altura. Asumiendo además un tamaño de bala de 5 de ancho y 20 de alto, el siguiente código calcula la posición y la velocidad de la bala:

```
bPos = p
      +Vector2D(w/2.0f,h/2.0f)
      -Vector2D(0.0f,h/2.0f+5.0f+12.0f).rotate(r)
      -Vector2D(2.0f,10.0f);
bVel = Vector2D(0.0f,-1.0f).rotate(r)*(vel.magnitude()+5.0f);
```

La rotación es la misma que la del caza.

## 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 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.

Para los asteroides de tipo `B`, el componente `Follow` tiene que girar el vector de velocidad en un grado en cada iteración para que el asteroide vaya hacia el 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)`.

### Dividir un asteroide

En el método `onCollision(Entity *a)`, al destruir un asteroide `a` la idea es desactivarlo y crear otros dos con generación como la de `a` menos uno (si el número de generaciones de `a` es 0 no se genera nada). La posición de cada nuevo asteroide tiene que ser cercana a la de `a` y tiene que moverse en una dirección aleatoria.

Por ejemplo, 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 velocidad de cada asteroide nuevo:

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

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

## Pautas generales obligatorias

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

- La solución debe estar estructurada en tres carpetas (dentro de la carpeta `TPV2\src` de la plantilla):
  - `src\ecs`: contendrá los ficheros propios de la **arquitectura** tales como "Component.h", "Entity.h", "Manager.h", etc.
  - `src\components`: contendrá los componentes que se creen.
  - `src\game`: contendrá la clase `Game` y el archivo `ecs_def.h`
  - El archivo `main.ccp` tiene que estar directamente en la carpeta `src`, es decir, al mismo nivel que las otras carpetas.
- 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.

## Funcionalidades opcionales (2 puntos adicionales máximo=

- Crea un archivo con el nombre `resources/config/asteroid.resources.json` y úsalo para definir referencias a todos los recursos necesarios (imágenes, sonidos, etc). *Como en la demo de la plantilla.*
- Introduce la posibilidad de configurar el juego desde un archivo de configuración de formato JSON, `resources/config/asteroid.config.json`. Puedes elegir qué valores se pueden configurar, p.ej., el número de asteroides inicial, la frecuencia de generación de asteroides, el número máximo de asteroides que puedan existir a la vez, 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.
- ¡Puedes inventar cualquier funcionalidad adicional que quieras!

# 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.