Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Industriales

Programación de Sistemas

Departamento de Electrónica y Automática

Práctica 2: Programación de un videojuego

Programación en ANSI C++ de una *lista enlazada* (ObjectList) para la gestión de un videojuego (estilo *Asteroids*)

Celia Ramos Ramírez (18295)
Gonzalo Quirós Torres (17353)
Josep María Barberá Civera (17048)

3° GITI

20 de junio de 2021

Índice

Introducción

La industria de los videojuegos es sabida como una de las que mayor economía y trabajo producen. Y además, de las que más impacto ha tenido en el mundo de la programación. Resultan por tanto la mejor oportunidad para introducirse en el mundo de la programación y de los gráficos.

En esta memoria se presenta una posible implementación en ANSI C++ para gestionar un videojuego estilo *Asteroids* mediante el uso de la librería gráfica multiplataforma OpenGL.

El entorno de programación utilizado ha sido Visual Studio Code. Se ha gestionado el trabajo mediante el uso de GitHub[1] y la extensión *Live Share* que ofrece VSCode.

El videojuego elegido es el famoso juego arcade *Asteroids* de Atari de 1979 [1]. Se han implementado las clases con sus atributos y métodos necesarios así como la lógica del juego para su correcto funcionamiento. Los requisitos cubiertos a grandes rasgos han sido:

- Creación de una clase *ObjectList* que gestiona una lista enlazada de los objetos del juego.
- Creación de la clase *Alien*, necesaria para instanciar el objeto *theUFO*.
- Integración del *Ovni* en la lógica del juego.
- Ajuste del sistema de puntuaciones.

Además de estos requisitos, se ha implementado alguna característica extra como que de un Ovni aparezca otro algunas veces cuando este sea destruido, o una nueva clase llamada *Angel* que otorga vidas al ser capturado por la nave. Dichos extras otorgan emoción al juego.

[1] Toda la documentación mencionada con respecto a este proyecto (y que puede ser útil para el seguimiento de su desarrollo) está disponible en el siguiente repositorio: https://github.com/jbarciv/Asteroids-with-openGL

Diagramas

2.1 Diagramas de Clases (UML)

Para entender correctamente la relación entre las clases, tanto de las ya dadas en el fichero original como de las creadas posteriormente, se ha realizado un diagrama de clases en formato *UML* para ello se ha empleado el programa StarUML®.

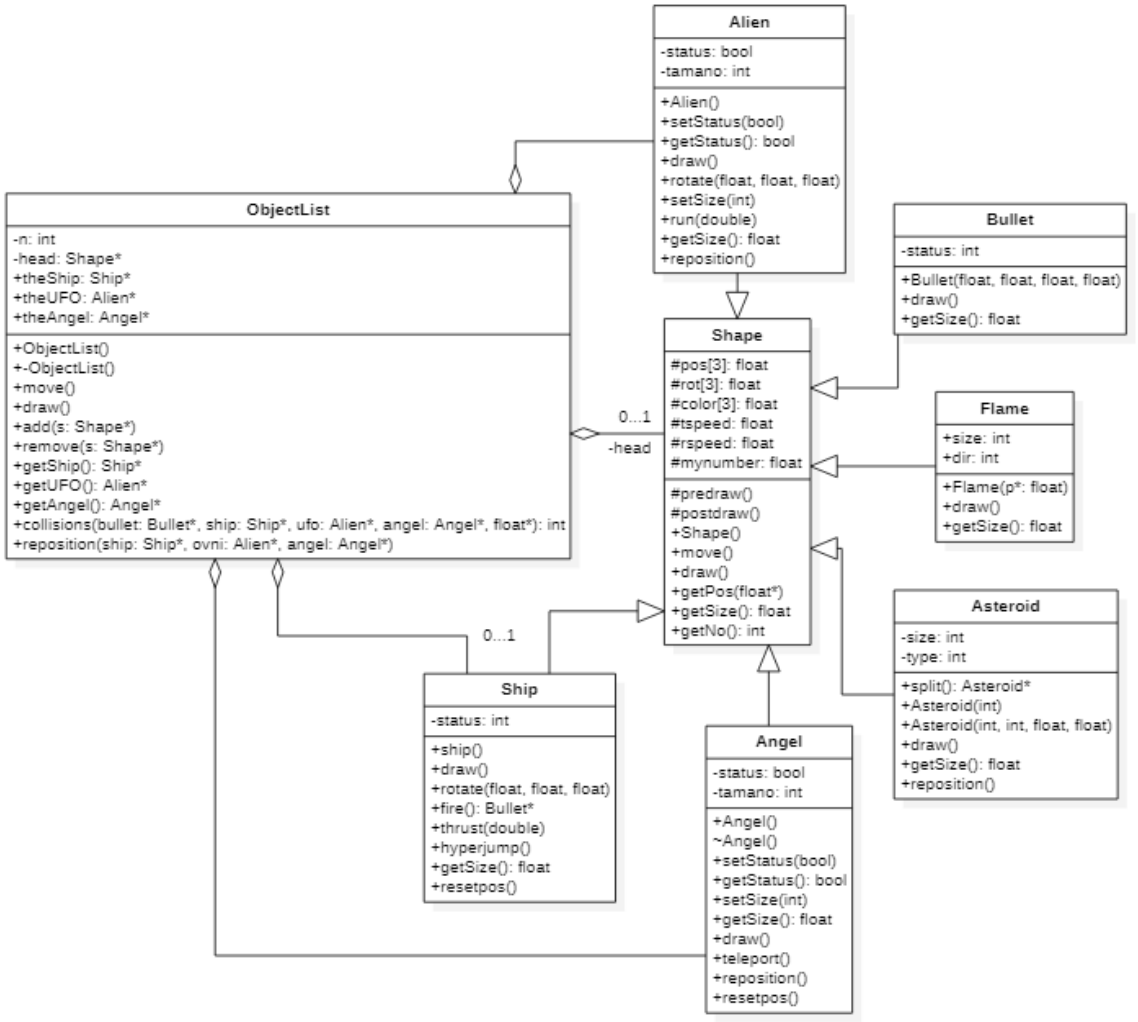

Figura 1: *Diagrama de clases del juego mediante* StarUML

Como puede verse en la Ilustración 1, la clase *ObjectList* agrega punteros a las clases *Alien*, *Ship* y *Angel*, además de contener un puntero tipo *Shape* llamado *head* que apunta a la cabeza de la lista (pero que en la implementación realizada no se utiliza). Al igual que la clase *Ship*, las clases *Alien*, *Angel*, *Bullet*, *Flame* y *Asteroid* son hijas (heredan) de la clase *Shape*. La clase *Alien* tiene mucho en común con la clase *Ship*, pues realizan prácticamente lo mismo, con la diferencia de que el *theUFO* (instancia de la clase *Alien*) se mueve automáticamente y de forma errática sin ser necesaria la interacción con el usuario. Por comodidad se han añadido dos punteros, uno al ovni y otro a la astronave (*theShip*), que simplificarán mucho el código tanto en la lógica del juego como en la implementación de la clase *ObjectList*. A parte de

las clases aquí representadas, existe otro archivo de cabecera llamado *commonstuff*, que almacena funciones cortas, parámetros y llamadas a librerías usadas en todos los archivos fuente del programa.

2.2 Diagramas de Objetos

2.2.1. Creación de los objetos

El siguiente diagrama ilustra desde dónde se crean los objetos que se van a usar en el resto del programa:

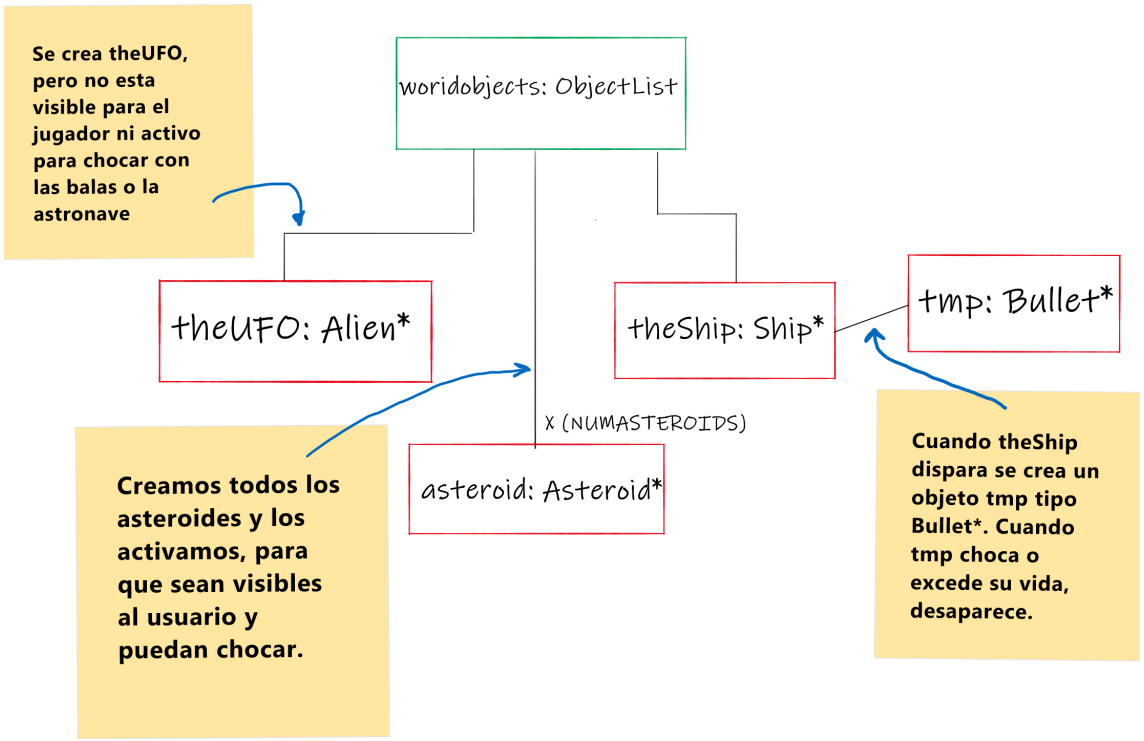

Figura 2: *Diagrama de creación de los objetos*

El diagrama de la Ilustración 2 representa el mecanismo de creación de los objetos más importantes que se lleva a cabo en el constructor de *ObjectList* y la creación de la bala a través del método FIRE() del objeto *theShip*. Se crean también todos los asteroides, donde su número concreto viene definido por el parámetro NUMASTEROIDS definido en *commonstuff.hpp*. Los asteroides son inmediatamente cargados en la interfaz gráfica mediante la función $push\_front()$ de la plantilla <list> a diferencia del OVNI (*theUFO*), que será cargado desde la lógica del juego cuando se cumplan las condiciones especificadas en el algoritmo.

A continuación, en otro pequeño diagrama, se verá el comportamiento de uno de los objetos *Asteroid* cuando es impactado por una bala o por la astronave.

2.3 Diagramas de comunicación

2.3.1. Comportamiento de un asteroide tras una colisión

En el siguiente diagrama (ver Ilustración 3) se observa el comportamiento de un objeto *Asteroid* tras una colisión, ya sea con una bala o con la astronave.

Figura 3: *Asteroide tras colisión*

Cuando un asteroide colisiona, surgen dos asteroides donde antes había uno con un tamaño disminuido en una unidad respecto al asteroide original. En realidad, lo que sucede, como se ve en el diagrama es que el asteroide original disminuye su tamaño en una unidad y en el mismo lugar surge otro con el tamaño del original también disminuido en una unidad. Evidentemente si el tamaño es 1 (esto es, SMALL), la función *Split()* no llega a ejecutarse porque el asteroide es destruido antes en la función *collision* de *ObjectList*.

Además, se incluye también en las siguiente ilustración (ver FIG. 4 cómo funciona la función COLLISION() y cómo se comunica con las clases que gestiona.

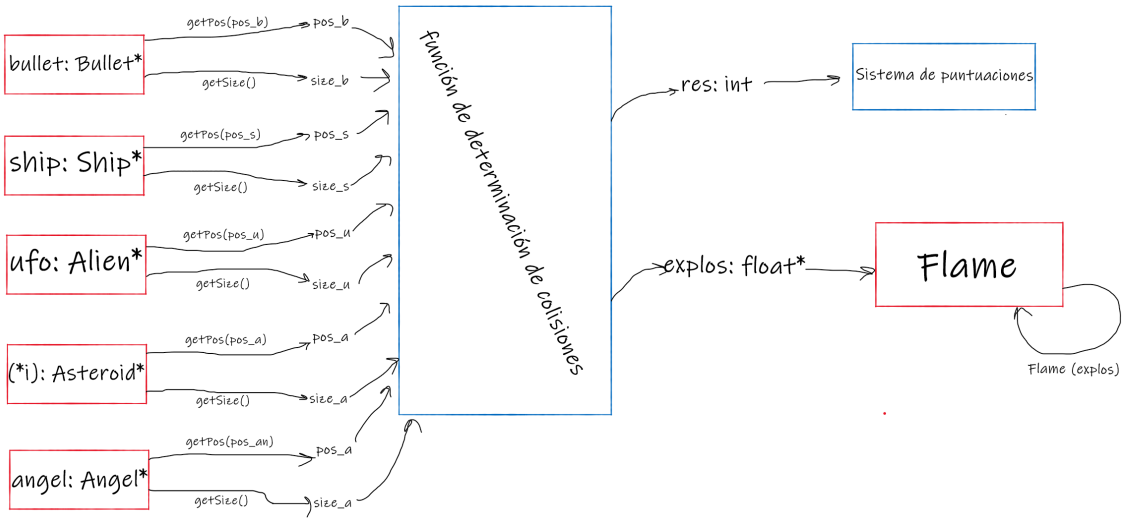

Figura 4: *Diagrama de comunicación en la función collision()*

Descripción de funciones y estructuras del juego

3.1 Clase ObjectList

La clase *ObjectList* hereda características de la plantilla <list> lo que permite implementar de forma muy sencilla una lista enlazada. Esta lista enlazada es necesaria para gestionar todos los objetos que han sido creados en el constructor del objeto *ObjectList* de forma eficiente. Por un lado, esta clase declara los objetos que serán usados durante el resto del programa y también realiza algunas funciones básicas como son eliminar estos objetos, moverlos o dibujarlos.

Además de declarar los objetos, la clase *OjectList* también hace visibles objetos ya creados mediante su función ADD(), que será muy útil para hacer visible el *ovni* cuando tenga que aparecer y hacer visibles los nuevos asteroides creados fruto de una colisión. Por comodidad, en la propia clase se han implementado dos métodos que devuelven un puntero a la *nave* y al *ovni*: GETSHIP() y GETUFO(), respectivamente. Estos punteros permitirán acceder a los métodos de ambas clases más fácilmente que si no estuvieran implementados.

Por otro lado, la función principal de la clase ObjectList es la función collisions() que devuelve entero. El entero devuelto permitirá identificar al tipo de colisión que se ha producido y permitirá a la lógica del programa, implementada en el archivo fuente mainAsteroids.cpp, gestionar las puntuaciones y eliminar o añadir objetos cuando proceda. En primer lugar, la función asigna tamaños y posiciones de los objetos principales esto es la nave, el ovni y la bala. posteriormente comprueban las distancias entre algunos de estos objetos susceptibles de colisionar, esto es: asteroides y astronave, asteroides y bala, OVNI y bala y astronave y ovnis.

- En el primer caso, el choque entre un asteroide y la astronave, el asteroide se divide o se destruye dependiendo de su tamaño y la astronave se destruye y si no era la última, una nueva astronave se reposiciona en el centro.
- Cuando una bala choca con un asteroide el asteroide se divide o se destruye dependiendo de su tamaño y la bala se destruye.
- Cuando la bala choca con el ovni tanto el ovni como la bala se destruyen, aunque puede ocurrir que donde había un ovni inmediatamente surja otro no ocurre todas las veces y el comportamiento es aleatorio.
- Cuando una astronave choca con un ovni o viceversa, el ovni queda intacto pero la astronave se destruye reposicionándose en el centro si todavía quedan vidas y si no es así se termina el juego.

En cada nueva llamada a la función COLLISIONS() se comprueban todos y cada uno de los elementos de la lista enlazada, obteniéndose sus posiciones y sus tamaños mediante un bucle FOR(). Dentro del bucle se obvian tanto la bala como la astronave pues se pasan como parámetros a la llamada de la función COLLISIONS() y no es necesario volver a comprobarlo, si bien es cierto que si se tienen en cuenta a la hora de

comprobar las distancias entre los elementos que se han mencionado anteriormente. La comprobación de las distancias se hace mediante la función MYDISTANCE() definida en *commonstuff.hpp*.

En todos los casos, cuando se detecta una colisión, se retorna un entero que indica el tipo de colisión y entre qué elementos se ha producido. Dependiendo del entero se actualiza la puntuación del jugador. Además, en un vector dedicado a tal efecto se marcan las coordenadas de la explosión. La duración de la explosión será definida en la lógica del juego.

Otra función muy importante, es REPOSITION(), se encarga de reposicionar la nave cuando esta es destruida. También se encarga de reposicionar los asteroides cuando la reposición de la nave en el centro no es posible.

3.2 Clase Alien

La clase *Alien* hereda características de la clase *Shape*, lo que le otorga rasgos muy similares al resto de figuras del juego. Esto facilita la integración con el resto del código, pues su tratamiento es muy similar al resto de figuras, aunque tiene características propias que son tratadas específicamente.

La clase *Alien* comparte muchas similitudes con la clase *Ship* en cuanto a lo que estructuración se refiere, pero su movimiento se controla autónomamemte desde un generador aleatorio de direcciones en uno de sus métodos llamado RUN(). Para que su movimiento sea autómático es necesario llamar al método ALIEN::RUN() desde el método ALIEN::DRAW(). A continuación se incluye el código que lo controla:

```
void Alien::run()
{
  if (time(NULL)-ref > 3){
    time(&ref);
    rot[Y] = RAND_FRAC()*360;
    tspeed[X] = ALIEN_SPEED*sin(D2R*rot[Y]);
    tspeed[Y] = ALIEN_SPEED*cos(D2R*rot[Y]);
  }
}
```

Movimiento errático del Alien *gracias a la función run()*

3.3 Clase Angel

La explicación de esta clase se realiza con más profundidad en la sección de propuestas de mejora.

Guía de uso

Figura 5: *Instrucciones del juego Asteroids implementado*

Pruebas

Las pruebas se dividían en dos bloques: pruebas de juego y de funcionamiento del código. Probar el juego fue un proceso entretenido y agradable una vez conseguido que el programa se ejecutara ya que eran fallos muy visuales y resultaba fácil detectar los fallos y su lugar en el código. Comprobar el funcionamiento del código fue, sin embargo, un poco más tedioso porque el juego no llegaba a ejecutarse y el código no generaba errores, por lo que encontrar los fallos fue un proceso arduo.

Nos hemos podido centrar en el funcionamiento de cada clase que compone el programa aislándolo y comprobando su comportamiento en el juego anulando los demás componentes. Por ejemplo, una vez los asteroides funcionaban correctamente, se establecio su número en cero para poder centrarnos en el *ovni* y el *angel*. Finalmente, cada uno de los integrantes del equipo probó el funcionamiento del juego en su totalidad.

Reparto de Roles

Para el reparto de tareas se decidió seguir el mismo proceso que para el trabajo anterior. En los primeros pasos de la investigación se realizaron reuniones y recopilación de ideas en las cuales cada integrante del grupo desarrolló poco a poco un boceto de la clase *ObjectList*.

Una vez se consiguió implementar dicha clase, se repartieron las tareas de desarrollo del resto del código entre Linux y Windows para seguir elaborando el programa y realizar pruebas en ambos sistemas operativos.

Al mismo tiempo se realizaba el diseño de los diagramas UML que servía de apoyo para escribir los códigos de los elementos que conforman el juego.

Cuando se consiguió una versión funcional se repartieron entre el equipo tareas de optimización y solución de errores.

Finalmente, con el programa finalizado, surgió la idea de implementar una nueva clase para añadir un toque original al proyecto. Dos integrantes desarrollaron e implementaron la nueva adición mientras el tercero se ocupaba de la parte documentativa del proyecto. Mientras se manteniendo al día la memoria y se redactaba las descripciones necesarias acerca de este.

Propuestas de mejora y valoración personal

Al tener un carácter clásico, el juego resulta sencillo y en ello reside su gracia. Evidentemente, hay infinidad de mejoras o extensiones que se podrían implementar. Sin embargo la principal preocupación del equipo era conseguir la versión básica del juego funcionando para familiarizarse con el entorno de OpenGL y posteriormente, si se tenía el tiempo necesario, se implementaría alguna mejora.

Una vez conseguido el juego en su versión básica como se ha mencionado con anterioridad, a excepción del disparo del *ovni* en dirección a la *nave*. Se decide incorporar un *power up* que recide el nombre de *Angel* en el juego. Este nuevo elemento se trata de una tetera predefinida en *OpenGL* que aparece después de 40 segundos de juego para otorgar al jugador la posibilidad de conseguir una vida extra si se acerca a ella y la 'recoge'.

Referencias

[1] Wikipedia contributors. Asteroids (video game), 06 2021.