Como el otro trabajo cuando pase algo a latex lo pongo en rojo... vosotros trabajad siempre con color **“negro”**.

Reparto de tareas:

- Diagramas de comunicación: Gonzalo

- Descripción: Gonzalo

- Pruebas: chema

- Manual: Celia

- Descripcion reparto: todos

- Propuesta: celia + todos

Introducción

El trabajo que se presenta a continuación ha sido desarrollado en el entorno de programación Visual Studio Code. El objeto de este es elaborar una solución posible al código incompleto del famoso juego arcade Asteroids® mediante la implementación de las clases y métodos necesarios y su integración en la lógica del juego para su correcto funcionamiento.

Los requisitos que se han cubierto han sido:

* Creación de una clase ObjectList que interacciona con el resto de las clases y crea los objetos.
* Creación de la clase Alien, que es necesaria para crear el OVNI.
* Integración en la lógica del juego.
* Sistema de puntuaciones.
* Ajuste de parámetros.

//Lo siguiente me lo he inventado porque no sé por qué ocurre...

Además de estos requisitos, se ha implementado alguna característica extra como que de un OVNI aparezca otro algunas veces y otras no cuando el OVNI se destruye, lo que otorga emoción al juego porque el jugador no sabe cuándo pasará ni si el nuevo OVNI tendrá un tamaño distinto.

Diagramas

# Diagrama de clases

Para entender correctamente la interrelación de 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 con el programa StarUML®.

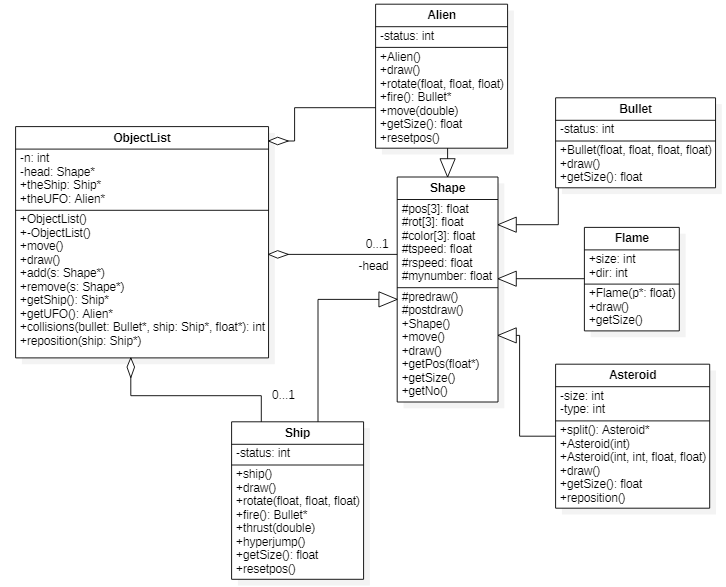


Ilustración 1: Diagrama de clases

Como puede verse en la Ilustración 1, se ha aprovechado la ya existente clase Shape para crear la clase Alien. Al igual que la clase Ship, la clase Alien es hija de la clase Shape y además la clase Alien tiene mucho en común con la clase Ship, pues hace prácticamente lo mismo, con la diferencia de que el OVNI (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, 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 como su nombre indica, almacena funciones cortas, parámetros y llamadas a librerías usadas en todos los archivos fuente del programa.

# Diagramas de objetos

## Creación de los objetos

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

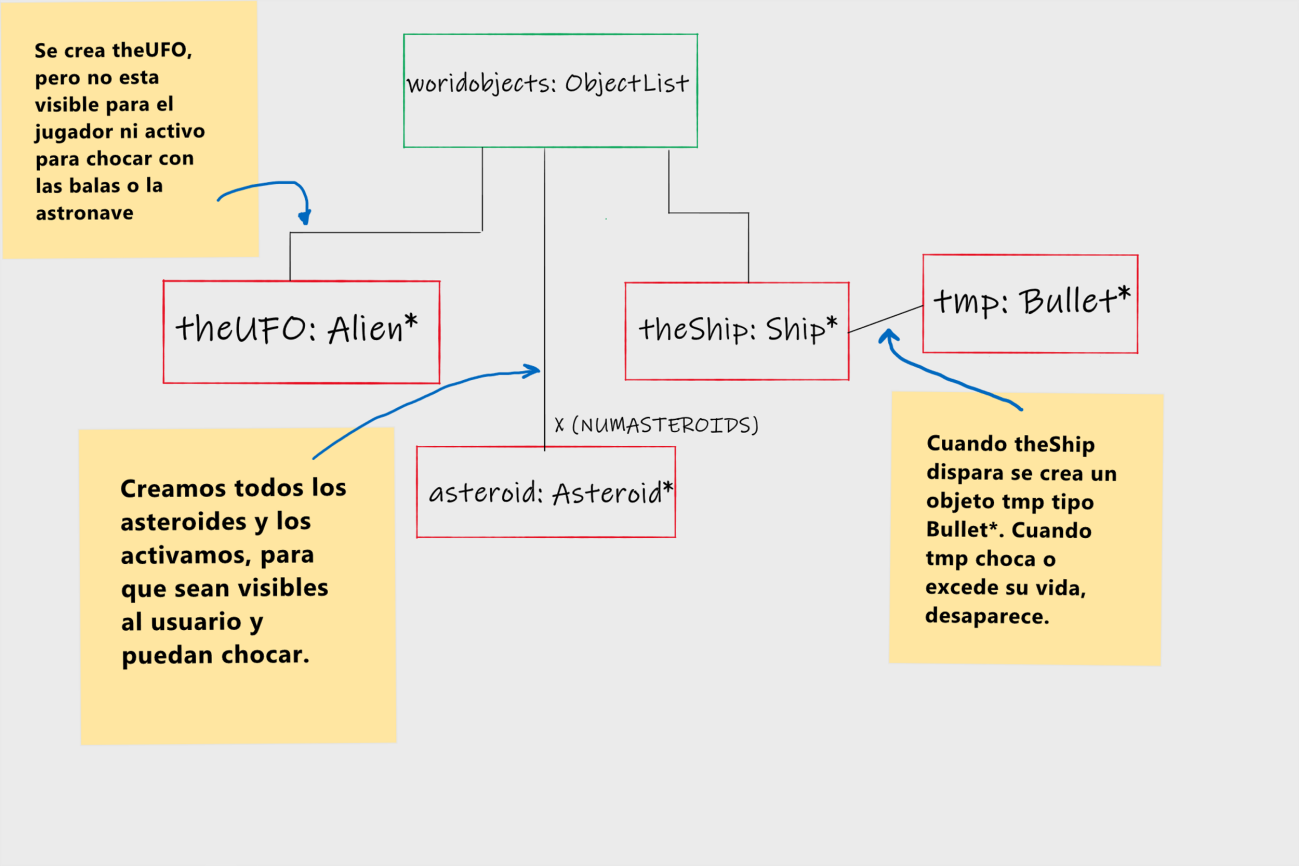


Ilustración 2: Creación de los objetos

El diagrama de la Ilustración 2 representa el mecanismo de creación de los objetos más importantes que se lleva a cabo en el constructor del objeto 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 de objetos, se verá el comportamiento de uno de los objetos asteroid cuando es impactado por una bala o por la astronave.

## Comportamiento de un asteroide tras una colisión

En el siguiente diagrama se observa el comportamiento de un objeto asteroid tras una colisión, ya sea con una bala o con la astronave.

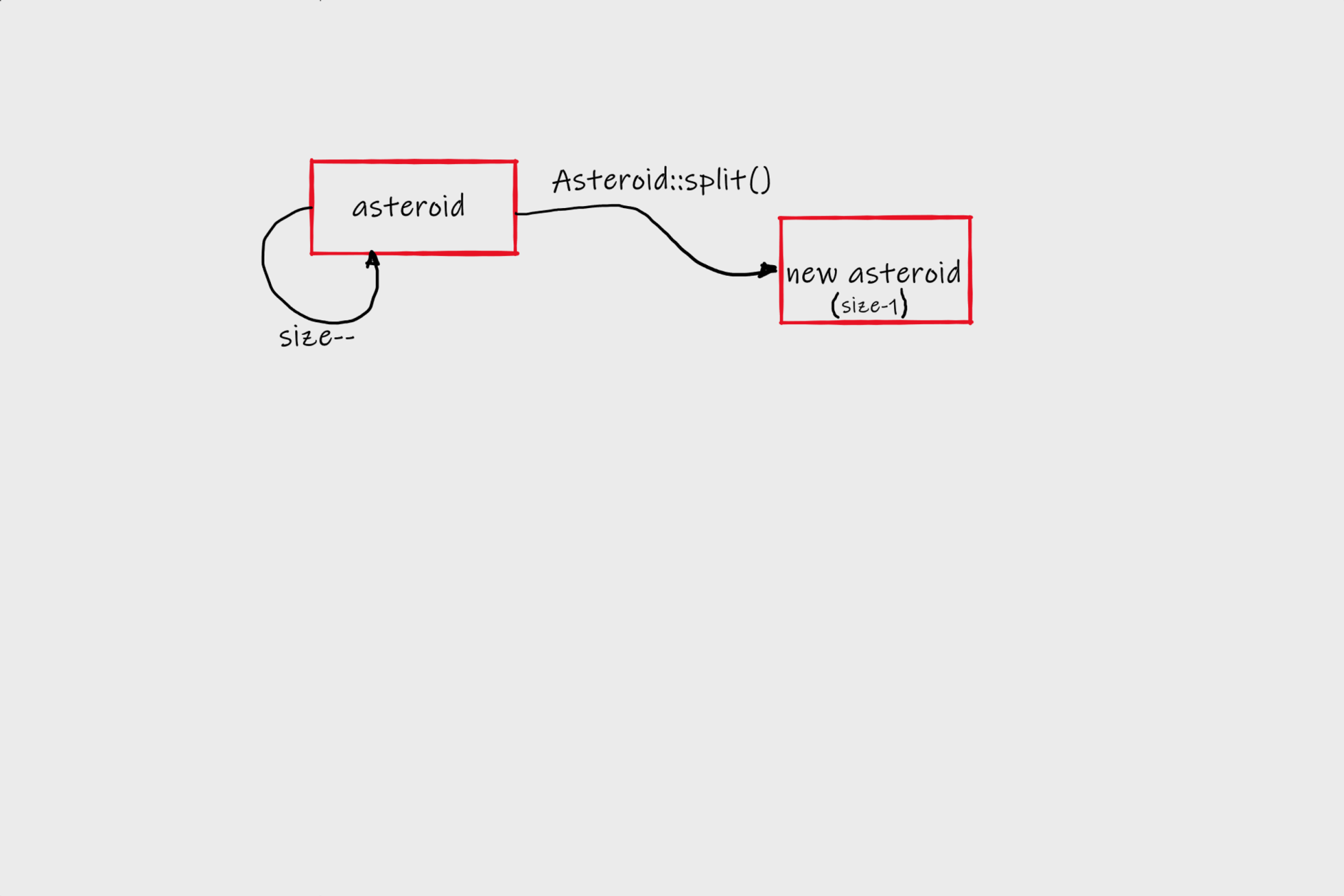


Ilustración 3: Asteroide tras colisión

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.

# 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, la clase ObjectList 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 la nave y del OVNI mucho 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, la función 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.

# Clase Alien

La clase Alien hereda características de la clase Shape, lo que le otorga características 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 estructuración, pero su movimiento se imprime automáticamente desde un generador aleatorio de direcciones en la lógica del juego. Para que su movimiento sea autónomo es necesario llamar al método Alien::run() desde el método Alien::draw()

# Reparto de Roles

Para el reparto de tareas se decide seguir el mismo proceso que para el anterior trabajo. Para los primeros pasos tuvieron lugar reuniones de investigación y recopilación de ideas y cada integrante del grupo desarrollaba poco a poco un boceto del object list.

Una vez se consiguió implementarlo, 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 medianamente funcional se repartieron entre el equipo tareas de optimización y solución de errores.

Finalmente, con el programa finalizado, surge la idea de implementar una nueva clase para añadir un toque original al proyecto. Dos integrantes desarrollan e implementan la nueva adicion mientras el tercero se ocupa de la parte documentativa del proyecto. Manteniendo al día la memoria y redactando las descripciones necesarias acerca de este.

# 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.

# propuesta de mejora y valoracion personal

Al tener un carácter clásico, el juego resulta sencillo y en ello reside su encanto. 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 anteriormente. Se decide incorporar un “power up” que recide el nombre de “ángel” 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’.

La implementación de éste resultó sencilla gracias a la familiaridad del equipo con el código y las clases ya implementadas después de muchas semanas de trabajo curando y desarrollando el código básico.