Práctica 2: Space Invaders 2.0

Curso 2023-2024. Tecnología de la Programación de Videojuegos 1. UCM

Fecha de entrega: 26 de noviembre de 2023

El objetivo fundamental de esta práctica es incorporar la herencia y el polimorfismo en la programación de videojuegos mediante C++/SDL. Para ello, partiremos de la práctica anterior y desarrollaremos una serie de extensiones que se explican a continuación. En cuanto a funcionalidad, el juego presenta las siguientes modificaciones o extensiones, algunas de ellas funcionalidades opcionales de la práctica anterior que ahora son obligatorias:

1. Como en el juego original, la formación de alienígenas desciende una posición de golpe cuando llega al borde de la escena y aumenta su velocidad en cada cambio de dirección. La partida acaba con derrota para el jugador humano si algún alienígena alcanza la horizontal del cañón.
2. El jugador obtiene puntos cada vez que un alienígena es destruido. Los alienígenas disparadores proporcionan 30 puntos, los verdes 20 y los rojos 10. La puntuación lograda hasta el momento se muestra continuamente en la ventana de juego.
3. Un OVNI como el del juego original atraviesa veloz y aleatoriamente la escena por encima de los alienígenas. Si el láser del jugador acierta a golpearle se obtendrán 100 puntos. La explosión del OVNI se mostrará durante unos milisegundos frames antes de desaparecer.
4. Se podrán guardar y cargar partidas mientras se está jugando. Al pulsar la tecla *S* seguida de un carácter numérico *k* se guardará la partida con nombre `saved`*k*`.txt`. Igualmente, al pulsar *L* seguida de un número se cargará la partida correspondiente o se mostrará un error si no está disponible. Si lo que sigue a *S* o *L* no es un carácter numérico se abortará la operación. Queda a la elección de los autores si pausar el juego mientras se espera al número o si se hace visible esta operación.

Detalles de implementación

Diseño de clases

A continuación se indican las modificaciones con respecto la práctica anterior que has de implementar obligatoriamente. Añade además los métodos (y posiblemente las clases) adicionales que consideres necesario para mejorar la claridad y reusabilidad del código. Como norma general, cada clase se corresponderá con la definición de un par de archivos .h y .cpp. Las nuevas clases de esta práctica son las siguientes:

Clase GameObject: la clase abstracta `GameObject` es la raíz de la jerarquía de objetos del juego y reúne la funcionalidad común a todos ellos. Su declaración incluye los métodos virtuales puros `render`, `update` y `save`, además de su destructora virtual y un atributo con un puntero al juego.

Clase SceneObject: es una subclase todavía abstracta de `GameObject` de la que descienden todos los personajes de la escena (es decir, todas las clases obligatorias de la práctica anterior). Mantiene la posición del objeto en la ventana, sus dimensiones y la vida restante como atributos. Declara un método virtual `hit` que recibe como argumento el rectángulo de ataque y el tipo de láser (alienígena o terrestre). Las subclases deberán implementar este método y recibir el daño si el láser es enemigo y el objeto interseca con el rectángulo, comprobación que se puede implementar en esta misma clase. Guardará además un iterador a la lista de objetos de la escena para facilitar su eliminación cuando corresponda (véase más abajo).

Clase ShooterAlien: representa al alien disparador de la práctica anterior y es una subclase de `Alien`, que ya no implementa la lógica del disparo. `ShooterAlien` reutiliza todo lo posible de `Alien` y se queda con aquellos atributos y operaciones específicas de este tipo de alienígena.

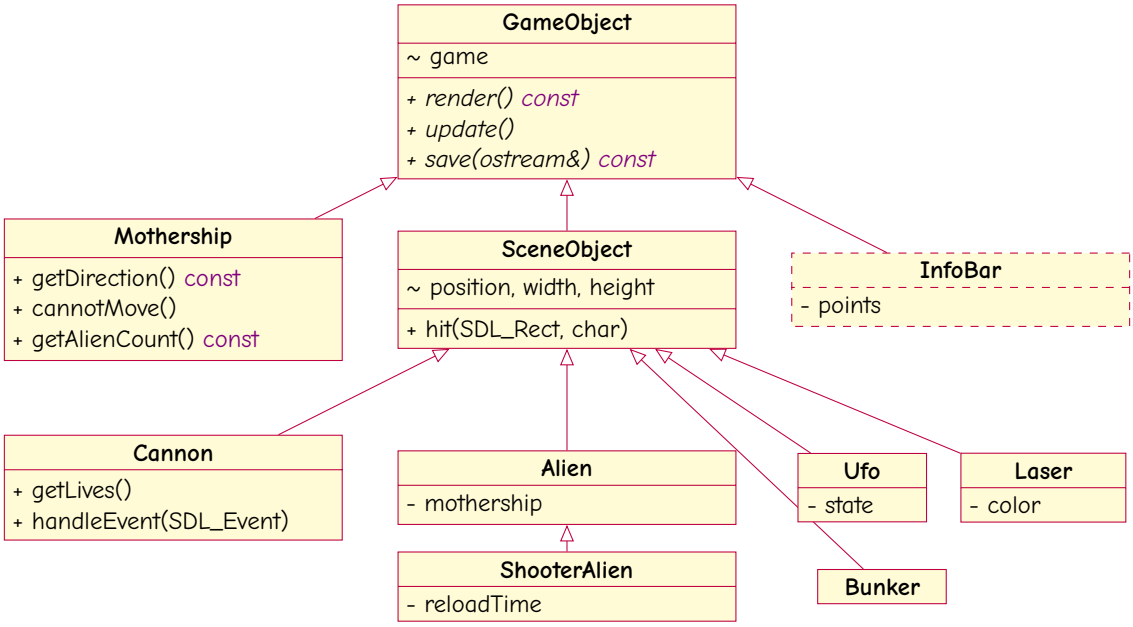


Figura 1: Digrama de clases principales del juego.

Clase Mothership: hereda de `GameObject` y coordina el movimiento de los alienígenas en la escena desde fuera de ella. Implementa los métodos `getDirection` y `shouldMove` para que los alienígenas conozcan la dirección de movimiento y si en ese ciclo les toca moverse; los métodos `cannotMove`, `alienDied` y `alienLanded`, a los que llamarán los aliens para informar de las circustancias correspondientes; y los métodos `haveLanded` y `getAlienCount` para informar al juego. Como ayuda para representar el estado del movimiento, más sofisticado que en la práctica 1, puede ser conveniente utilizar un enumerado cíclico `state` y un nivel `level` de descenso, que afectará a la velocidad de los alienígenas.

Clase Ufo: es el OVNI que atraviesa la escena en intervalos de tiempo aleatorios. Además de los atributos que hereda de `SceneObject`, utilizará un enumerado para almacenar su estado (visible, oculto o destruido, este último para la animación de la explosión). El tiempo aleatorio que permanece oculto se puede determinar de la misma forma que el tiempo de disparo de los alienígenas en la práctica anterior.

Las clases de la práctica 1 se han de adaptar de forma natural para hacerlas subclases de `SceneObject`. Algunas de ellas necesitan cambios adicionales:

Clase Alien: además de los cambios por la herencia de `SceneObject` y la segregación de `ShooterAlien`, incorporan como nuevo atributo un puntero al objeto `Mothership` del juego.

Clase Laser: el booleano `forAliens` se reemplaza por un carácter `color` con posibles valores `r` (rojo) y `b` (azul) para mayor simetría y para facilitar futuras extensiones (en lugar de un `char` se podrá usar un tipo enumerado). Como hasta ahora, los láseres rojos no afectan al cañon mientras que los láseres azules no afectan a los alienígenas. Los búnkeres se ven afectados por ambos colores.

Clase Game: en lugar de almacenar múltiples vectores para cada tipo de objeto del juego y tratarlos específicamente, utilizará una lista polimórfica de punteros a `SceneObject` (de tipo `std::list<SceneObject*>`). Los métodos `update`, `render` y demás solo tendrán que aplicar la misma operación a todos los objetos de la lista. Los objetos de tipo `GameObject` que no son `SceneObject` (`Mothership` y tal vez `InfoBar`) se pueden manejar por separado por simplicidad. Además, parte de su funcionalidad de `Game` se ha trasladado a `Mothership` y tendrán que añadirse nuevos métodos para manejar el guardado y la carga de partidas.

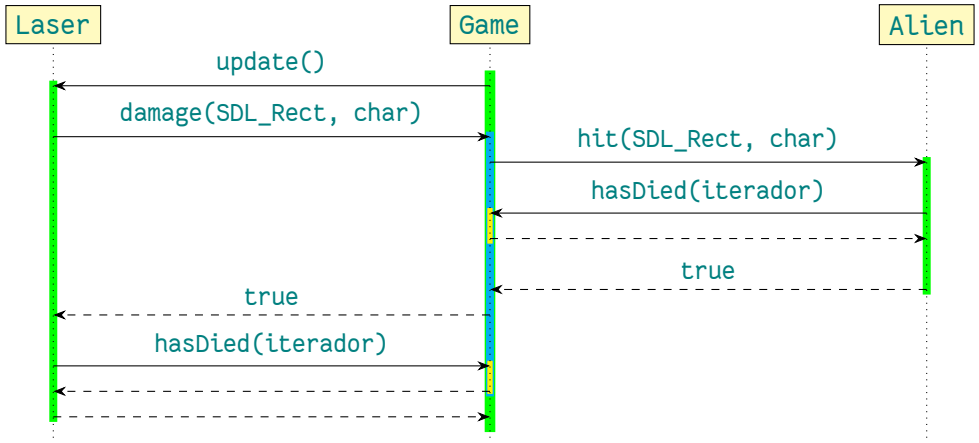
Lista de objetos de la escena y su eliminación

La clase `Game` mantiene una lista de punteros a todos los objetos de la escena (de tipo `SceneObject`) para invocar sus métodos virtuales `update`, `render`, `save` o `hit` independientemente de su tipo concreto (es decir, polimórficamente). La mayoría de estos objetos se crean y eliminan dinámicamente y para facilitar

su eliminación eficiente utilizaremos una lista enlazada (tipo plantilla `list` de la STL) y haremos que cada `SceneObject` guarde como atributo privado un iterador a su posición en la lista (de tipo plantilla `list::iterator`). Ya que el iterador no estará disponible hasta que el objeto haya sido añadido a la lista, operación que conviene hacer después de su creación, `SceneObject` deberá implementar un método `setListIterator` para fijar el iterador a posteriori. La clase `Game` deberá implementar un método público `hasDied` al que los objetos del juego llamarán con su iterador cuando hayan de ser eliminados.

Ataques y colisiones entre láseres y objetos

El siguiente diagrama ilustra parte de la secuencia de actualización (método `update`) de un objeto de tipo `Laser` en el momento en el que este choca con un objeto de tipo `Alien`, desencadenando la eliminación de ambos.



Tras moverse, el láser llama al método `damage` de `Game` para atacar la región cubierta por él. El juego entonces llama al método `hit` sobre todos los objetos de la escena, que comprobarán si el láser es enemigo y si interseca con su posición. Como el alienígena cumple esas condiciones se ha dado por atacado, ha llamado al método `hasDied` de `Game` para indicarle que ha de ser eliminado y ha devuelto `true` al salir del método `hit`. Gracias a esta información el láser sabrá que su ataque ha surtido efecto y que por tanto él también tiene que ser elimininado. Llamará también al método `hasDied` de `Game` pasándole su iterador.

Partidas guardadas y configuraciones iniciales

Con el propósito de permitir la distribución de mapas iniciales y partidas guardadas, el formato de estos archivos deberá seguir la siguiente especificación, que extiende la utilizada en la práctica anterior. Cada línea del archivo representa un elemento y comienza por un número que determina su tipo (0 = cañón, 1 = alienígena, 2 = alienígena disparador, 3 = nave nodriza, 4 = búnker, 5 = ovni y 6 = láser). El número 7 se utilizará para almacenar la puntuación de la partida (si se implementara la clase `InfoBar` de la parte opcional, este podría ser naturalmente el número que le corresponde a este elemento). La descripción de los objetos de la escena continúa con dos números para las coordenadas de su posición desde la esquina superior izquierda (asumiendo una pantalla de 800x600 unidades). Los parámetros adicionales de los objetos siguen el siguiente formato:

Cannon	vidas	espera

Alien	subtipo

ShooterAlien	subtipo	espera

Ufo	altura	estado	espera

Laser	color

Mothership	estado	nivel	espera

Algunos de estos elementos no tiene sentido que aparezcan en los mapas iniciales, pero por simplicidad no se hará distinción entre mapas iniciales y partidas guardadas.

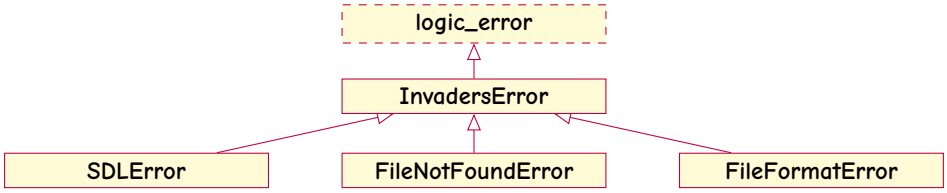
La lectura y escritura de la descripción de los objetos se ha de distribuir entre las diversas clases que mantienen dichos datos. El método virtual `save` de `GameObject`

```
virtual void save(std::ostream& out) const = 0;
```

debe implementarse en cada objeto del juego para guardar su estado. La lectura se puede implementar mediante constructores que reciban un argumento de tipo `std::istream&`. Por ejemplo, el método virtual `save` de `Cannon` delegará en `SceneObject` para guardar la posición del objeto. Si más tarde añadiésemos más datos comunes a todos los objetos no haría falta actualizar la implementación de `Cannon`.

Jerarquía de excepciones

El manejo de los errores del juego se ha de implementar mediante las siguientes clases de excepciones. Las excepciones irrecuperables se han de capturar en el `main` y la función `SDL_ShowSimpleMessageBox` puede utilizarse para mostrar el mensaje de error al jugador.



InvadersError: hereda de `std::logic_error` y sirve como superclase de todas las demás excepciones que definiremos, proporcionando la funcionalidad común necesaria. Reutiliza el constructor y método `what` de `logic_error` para el almacenamiento y uso del mensaje de la excepción.

SDLError: hereda de `InvadersError` y se utiliza para todos los errores relacionados con la inicialización y uso de SDL. Utiliza la funciones `SDL_GetError` para obtener un mensaje específico sobre el error de SDL que se almacenará en la excepción.

FileNotFoundError: hereda de `InvadersError` y se utiliza para los errores provocados al no encontrarse un fichero que el programa trata de abrir. El mensaje del error debe incluir el nombre del fichero en cuestión.

FileFormatError: hereda de `InvadersError` y se utiliza para los errores provocados en la lectura de los archivos de datos del juego (mapas y partidas guardadas). La excepción debe almacenar y mostrar el nombre de archivo y el número de línea del error junto con el mensaje.

Pautas generales obligatorias

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

- Se debe hacer un buen uso de la herencia y del polimorfismo de manera que el código sea claro, se eviten las repeticiones de código, distinciones de casos innecesarias, no se abuse de castings ni de consultas de tipos en ejecución.
- Asegúrate de que el programa no deje basura. La plantilla de Visual Studio incluye el archivo `checkML.h` que debes introducir como primera inclusión en todos los archivos de implementación.
- Todos los atributos deben ser privados o protegidos excepto quizás algunas constantes del juego definidas 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. Sé cuidadoso también con los nombres que eliges para variables, parámetros, atributos y métodos. Es importante que denoten realmente lo que son o hacen. Preferiblemente usa nombres en inglés.

Funcionalidades opcionales

1. Implementar la animación de los alienígenas. La textura proporcionada incluye dos variantes para cada tipo de alienígena que pueden alternarse secuencialmente.
2. Implementar el control del cañón mediante el ratón. La pulsación de cualquiera de los botones disparará el cañón y este se moverá a su velocidad habitual hacia la posición vertical del ratón en la ventana.
3. Implementa una clase `InfoBar` cuyo método `render` se encarga de mostrar en la ventana SDL una barra de información del juego que incluya al menos el número de vidas restantes y la puntuación actual.
4. Implementar un soporte primitivo para niveles en el juego, de tal forma que si el jugador elimina a todos los alienígenas se cargue un nuevo mapa para continuar jugando. Los niveles del juego serían una serie de archivos denominados `map`*`k`*`.txt` que el juego cargaría sucesivamente.
5. Añadir un nuevo efecto a la destrucción del OVNI con el que el jugador consiga invulnerabilidad por un tiempo limitado. La representación del cañón debería ser algo distinta mientras dure el efecto.

Entrega

En la tarea *Entrega de la práctica 2* del campus virtual y dentro de la fecha límite (ver junto al título), cualquiera 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.

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 (posiblemente individuales) sobre la implementación. Se darán detalles más adelante sobre las fechas, forma y organización de las entrevistas.