

Práctica 3: Space Invaders 3.0



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

Fecha de entrega: 17 de diciembre de 2023

El objetivo fundamental de esta práctica es introducir una arquitectura escalable para el manejo de los estados de un juego. Para ello, partiendo del Space Invaders 2.0 de la práctica anterior, extenderemos el juego con los siguientes nuevos estados:

- Al arrancar el programa aparecerá el *menú principal*, que permitirá al menos empezar una partida nueva, cargar una partida guardada a partir del código numérico usado al guardarla (en caso de no ser válido se informa, preferiblemente mediante `SDL_ShowSimpleMessageBox`, y se continua en el menú) y salir del juego.
- Mientras se está jugando, si se pulsa la tecla `[Esc]`, el juego se detiene y se visualiza el *menú pausa*, que permite al menos: reanudar la partida, guardar la partida (para lo cual se solicitará el código numérico, permaneciendo en el menú), cargar una partida guardada y salir del juego.
- Finalmente, cuando se acabe la partida deberá visualizarse el *menú fin*, una pantalla en la que además de informar al usuario si ha ganado o perdido, aparecerá un menú con opciones para volver al menú principal y para salir de la aplicación.

En todos los menús, será posible usar el ratón para seleccionar la opción elegida.

Por otro lado, se extenderá la mecánica del juego para que al derribar al ovni se desencadenen con cierta probabilidad recompensas capturables o efectos adversos. Consideraremos dos de ellas: (1) la extensión opcional de la práctica anterior que daba invulnerabilidad al cañón durante un cierto tiempo ahora se conseguirá al capturar un escudo  que caiga del ovni cuando este sea alcanzado, y (2) en su lugar podría caer una bomba  que causa daño al jugador y los búnkeres como un láser alienígena, pero tiene dos vidas y por tanto puede atacar dos veces. Una tercera posibilidad es que no ocurra nada. Las probabilidades de las distintas opciones se pueden escoger libremente.

Detalles de implementación

Estados del juego

El juego utilizará una máquina de estados para manejar las transiciones entre estados del juego. Implementa por tanto la clase `GameStateMachine`, que incluye como atributo una pila de estados (tipo `stack<GameState*>`) y métodos `pushState`, `replaceState`, `popState`, `update`, `render` y `handleEvent`. Los métodos `update`, `render` y `handleEvent` de la clase `Game` delegarán respectivamente en los métodos `update`, `render` y `handleEvent` del estado actual a través de los métodos homónimos de la máquina de estados. Debes implementar al menos las siguientes clases para manejar estados:

Clase `GameState`: es la clase raíz de la jerarquía de estados del juego y tiene al menos tres atributos: la colección de objetos del juego (`GameList<GameObject, true>`, que se explica en la página 3), los manejadores de eventos (`List<EventHandler*>`, véase más adelante) y el puntero al juego. Implementa los métodos `update`, `render` y `handleEvent`, y también `addEventListener` y `addObject` para añadir oyentes y objetos al estado.

Clase `PlayState`: implementa el juego propiamente dicho, así que incluye gran parte de los atributos y funcionalidad que antes teníamos en la clase `Game`. Los objetos de la escena se comunicarán con este estado como antes se comunicaban con `Game` (`damage`, `hasDied`, etc.). Además de la lista de `GameObject` heredada de `GameState`, este estado guardará una lista adicional con todos los objetos de la escena (`GameList<SceneObject>`) para calcular las colisiones.

Clases MainMenuState, PauseState y EndState: implementan respectivamente los estados del juego correspondientes a los menús *principal*, *pausa* y *fin* como subclases de `GameState`. El escenario de cada menú estará compuesto por objetos de tipo `Button` e imágenes estáticas. En el material de la práctica se proporcionan texturas para los botones y las imágenes de fondo.

Observa que ahora la clase `Game` quedaría solo con los siguientes atributos básicos: los punteros a `SDL_Window` y `SDL_Renderer`, el array de texturas y la máquina de estados. La aplicación terminará cuando la pila de estados quede vacía o alternativamente utilizando un booleano `exit`. De hecho, esta clase podría pasar a llamarse `SDLApplication` pues ya no tiene nada referente al juego propiamente dicho. Los objetos de tipo `GameObject` ahora guardarán un puntero al `GameState` del que forman parte en lugar de al `Game` (lo necesitarán para llamar a su `hasDied`, por ejemplo). Además, los objetos de tipo `SceneObject` guardarán un puntero a su `PlayState` (ese `PlayState` es el mismo `GameState` al que pertenecen, pero C++ no permite refinar el tipo de un atributo, por lo que es necesario usar dos atributos o `static_cast`).

Botones y eventos

Los menús del juego permiten elegir entre diversas opciones mediante botones, en los que el jugador puede hacer click. Estos botones se manejan como objetos del juego, es decir, saben dibujarse, actualizarse (si es necesario), reaccionan a eventos de la SDL y emiten sus propios eventos.

Clase Button: por lo dicho en el párrafo anterior, es subclase de `GameObject` y `EventHandler`, con atributos para su textura y para la función o funciones a ejecutar en caso de ser pulsado (de tipo `Button::Callback`, un alias de `std::function<void(void)>`), que se invocarán desde el método `handleEvent`. Los callbacks se registrarán mediante un método público `connect` de la clase. Recuerda que se puede crear un objeto función para invocar al método de un objeto con cualquiera de

```
button.connect([this]() { método(); });           // expresión Lambda
button.connect(std::bind(&Clase::método, this));   // puntero al método + objeto
```

Clase EventHandler: se trata de una clase abstracta con un único método virtual puro `handleEvent` que recibe un `SDL_Event` (en la terminología de otros lenguajes, esto sería una interfaz). La clase `Game` se encargará como hasta ahora de capturar los eventos con `SDL_PollEvent`, pero en esta práctica los retransmitirá a todos los oyentes registrados de tipo `EventHandler`, para lo que tendrá que guardar una lista de punteros como se ha visto en clase. Las clases que capturan eventos de la SDL, `Cannon` y `Button`, implementarán esta interfaz y establecerán en la definición del método cómo responder a los eventos.

Recompensas y bombas

Al derribar un ovni en esta práctica, además de sumar 100 puntos al marcador, este podrá soltar al juego una recompensa capturable por el jugador o una bomba. Ambos se implementarán como objetos de la escena (`SceneObject`).

Clase Reward: representa una recompensa que se puede obtener al colisionar la nave con ella. Solo implementaremos un tipo de recompensa en esta práctica (la que otorga inmunidad durante un tiempo limitado a la nave), pero el propósito es que esta clase se pueda extender fácilmente para soportar otros efectos. El comportamiento de las recompensas hasta ser capturadas será siempre el mismo, caerán a velocidad constante desde el ovni y desaparecerán por la parte inferior del tablero si no han colisionado antes con la nave, pero sus texturas y efectos podrán ser distintos. Es por eso que la clase `Reward` implementará el comportamiento común, recibirá una textura para pintarse y un callback con la acción a realizar al colisionar con la nave. Para detectar la colisión con la nave, la clase `Reward` llamará en su método `update` a un nuevo método `mayGrantReward` de `PlayState` con su rectángulo de corte. Las recompensas no interactúan con otros objetos y no se salvan al guardar partida.

Clase Bomb: esta clase representa una bomba alienígena que se comportará igual que un láser pero dispondrá de dos vidas y tendrá una apariencia distinta. En cada colisión solo causará y recibirá un punto de daño, por lo que podría seguir moviéndose o atacando en la iteración siguiente.

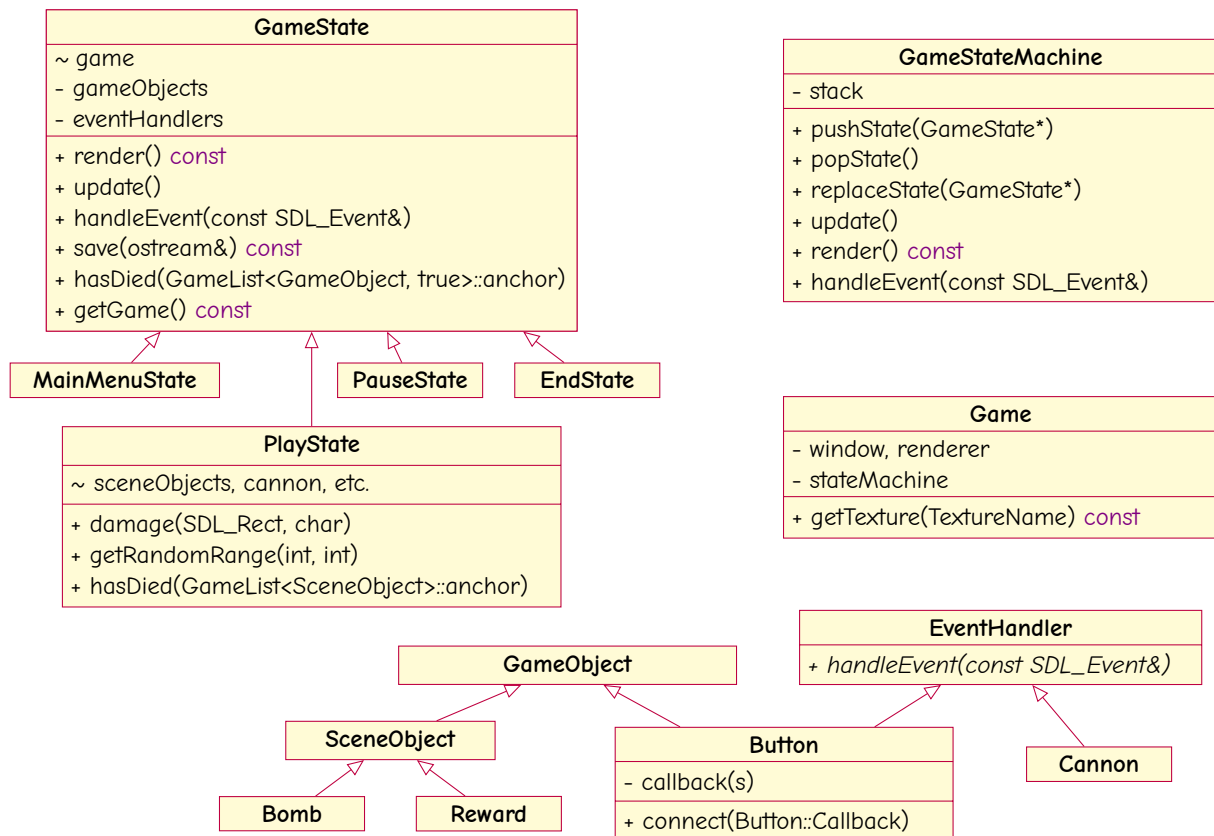


Figura 1: Diagrama incompleto de las nuevas clases del juego.

Borrado eficiente en las listas de objetos del juego y de la escena

En la práctica 2, la clase `Game` mantenía una lista de objetos de la escena de tipo `List<SceneObject*>`, que ahora pasa al estado `PlayState`. Con el fin de eliminar eficientemente los objetos al ser destruidos, utilizábamos una lista doblemente enlazada en lugar de un vector y guardábamos en cada objeto de la escena su iterador en esa lista. El objeto llamaba al método `hasDied` de `Game` con su iterador y el juego acababa llamando al `erase` de la lista con él. Sin embargo, no es seguro en general eliminar un elemento de una lista mientras se está iterando sobre ella, por lo que había que complicar la lógica del borrado con una lista temporal `toBeDeleted` o mecanismo equivalente.

En esta práctica, además de la lista de `SceneObject` de `PlayState`, que ahora solo se utilizará para calcular las colisiones, tendremos una lista de `GameObject` cada estado del juego sobre la que hacer `update`, `render` y demás. Queremos que los objetos también se eliminen eficientemente de esa lista, por lo que cada `GameObject` tendrá que almacenar un iterador a ella. Cuando un objeto del juego se elimine, llamará al método `hasDied` de `GameState` (que heredará el estado concreto al que pertenezca) con su iterador de `GameObject`. Si es además `SceneObject`, habrá de llamar también al método `hasDied` específico de `PlayState` con su otro iterador de `SceneObjects` para eliminarse de la lista para colisiones. La eliminación eficiente no se aplica a la lista de `EventHandler` de `GameState` pues asumimos que duran tanto como el estado o se eliminan todos simultáneamente.

Clase `GameList` (disponible en el CV): para evitar repetir el código que maneja los iteradores y las eliminaciones (con todas las complicaciones mencionadas), se proporciona una plantilla `GameList` que se podrá utilizar en ambos casos. La interfaz de esta clase es semejante a la de una lista de la STL con algunas particularidades adecuadas a este contexto: incluye un tipo `GameList::anchor` que usaremos como el `list::iterator` en la práctica anterior; sus métodos `push_back` y `push_front` llaman automáticamente al método `setListAnchor` con el objeto de tipo `anchor` correspondiente y su método `erase` recibe un objeto `anchor` y borra el elemento al que apunta de forma segura. Por comodidad, los métodos `push_back` y `push_front` reciben punteros, pero los iteradores de `GameList` devuelven objetos por referencia.

El tipo `GameList` es una plantilla

```
template<typename T, bool owns = false> class GameList { /* ... */ }
```

Su primer argumento es el tipo de los elementos **(sin puntero)**, pero tiene también un segundo argumento de tipo booleano **owns** que indica si la lista es *propietaria* de los elementos que contiene, es decir, si es responsable de liberar su memoria (con **delete**) cuando se eliminen. La lista de **SceneObject** de **PlayState** contiene solo algunos de los **GameObject** del estado del juego, aquellos que pueden colisionar, así que es más razonable que la lista general sea la propietaria de los objetos. Por eso, los tipos de las listas serán **GameList<GameObject, true>** y **GameList<SceneObject, false>**.

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, y 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 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. 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 (2 puntos adicionales máximo)

1. Implementa el soporte para que algunos parámetros del juego (ruta de las texturas, tiempos de recarga de los disparos, velocidad de movimiento, etc.) se carguen de un archivo de texto. Utiliza además una estructura de tipo tabla (**map** o **unordered_map**) para almacenarlas y obtenerlas a partir de un identificador unívoco (tipo **string**) para cada una (que estaría también en el fichero).
2. Utiliza el paquete **TTF** de SDL para manejar los distintos textos del juego (contador, botones, etc.). Es recomendable encapsular la interacción con la biblioteca en la clase **Font** que aparece en las diapositivas del tema 6 y definir un objeto del juego **Label** o semejante para colocar piezas de texto en la pantalla (**Button** podría ser subtipo de **Label**).
3. Implementa el soporte para que se registren de manera ordenada las puntuaciones (o tiempos) de las partidas acabadas junto con sus fechas y nicks asociados. Debes utilizar para ello una tabla (tipo **map**) indexada por puntuación y fecha.
4. Implementa un estado del juego para recoger el código numérico que hay que introducir al cargar/guardar una partida. Los dígitos deberán visualizarse según se vayan introduciendo.

Entrega

En la tarea *Entrega de la práctica 3* 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* como en entregas anteriores.