Práctica 2: Arkanoid 2.0

Curso 2018-2019. Tecnología de la Programación de Videojuegos. UCM

Fecha de entrega: 3 de diciembre de 2018

El objetivo fundamental de esta práctica es introducir el uso de la herencia y el polimorfismo en la programación de videojuegos mediante SDL/C++. Para ello, partiremos del Arkanoid 1.0 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/extensiones:

1. Cuando se destruyen todos los bloques del mapa de bloques el juego pasa de nivel (otro mapa diferente que se carga del fichero correspondiente). El objetivo del juego puede ser o bien llegar a una determinada puntuación (la cual tendría que mostrarse de alguna manera), o bien acabar con todos los niveles del juego.

2. Al destruir un bloque, de manera aleatoria y con una cierta probabilidad, caerá un premio desde el bloque destruido hacia abajo, de manera que si al caer toca con la plataforma, se aplicará el efecto correspondiente del premio. Al menos deben existir los siguientes premios: pasar de nivel (letra L), obtener vida extra (letra R), y alargar/acortar la plataforma (letras E/S), que se restablecerá al coger otro premio o pasar de nivel.

3. Las partidas pueden guardarse y cargarse: Al iniciar el juego aparecerá un menú con dos opciones, jugar y cargar partida. La opción debe seleccionarse con el ratón. En caso de seleccionarse la opción de cargar partida, se introducirá a continuación el código numérico de la partida y se cargará la partida a partir del fichero correspondiente (en caso de encontrarse). De manera análoga, mientras se está jugando, si se pulsa la tecla *s* seguida de un código numérico y seguida de *intro*, la partida se guardará con el código numérico introducido. Se darán detalles en clase sobre esta funcionalidad.

Detalles de implementación

Cambio de nivel

Como se ha indicado, cuando se destruyen todos los bloques de un nivel (o cuando se coge el premio de pasar de nivel), el juego pasa al siguiente nivel. El juego incluye por tanto un nuevo atributo que indica el nivel actual. Éste se utiliza para formar el nombre del fichero del siguiente nivel, el cual tiene la forma `levelX.ark` siendo X el nivel correspondiente. Para ello puedes hacer uso de un flujo de tipo `stringstream`, que permite manejar *strings* como flujos de texto y por tanto aprovechar las funcionalidades para insertar y extraer datos en flujos de texto. Al pasar de nivel el mapa de bloques se destruye creándose uno nuevo, los premios que hubiese en la escena se destruyen, y el resto de objetos se mantienen, recolocándose en sus posiciones iniciales si corresponde.

Diseño de clases

A continuación se indican las nuevas clases y métodos que debes implementar obligatoriamente y las principales modificaciones respecto a las clases de la práctica 1. Deberás implementar 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 módulo C++ con sus correspondientes ficheros .h y .cpp.

Clases GameObject, ArkanoidObject y jerarquía de objetos del juego: La clase raíz de la jerarquía es la clase abstracta `GameObject`, que declara la funcionalidad común a todo objeto de un juego SDL (métodos abstractos `render`, `update` y `handleEvent`). La clase `ArkanoidObject`, también abstracta, hereda de `GameObject` y declara toda aquella funcionalidad común a los objetos del juego Arkanoid (métodos `loadFromFile`, `saveToFile` y `getRect`). Incluye atributos para la posición, el ancho, el alto y un puntero a la textura del objeto. Define una constructora que inicializa todos los atributos a partir de los parámetros correspondientes, y proporciona una implementación genérica de los métodos en los casos en los que esto sea posible.

Clases Wall, BlocksMap y Block: Su significado y funcionalidad es la misma que en la práctica 1, con la salvedad de que ahora heredan parte de ella de `ArkanoidObject`, añadiendo los atributos y métodos que sean necesarios en cada caso, y (re)definiendo según corresponda los métodos heredados.

Clase MovingObject: Hereda de `ArkanoidObject` y proporciona el soporte común a los objetos del juego que tienen movimiento. Incluye por tanto un atributo para la dirección/velocidad y (re)define los métodos correspondientes para que lo tengan en cuenta. Por ejemplo, los métodos `loadFromFile` y `saveToFile` deberán también cargar/guardar la velocidad.

Clases Paddle, Ball y Reward: El significado y funcionalidad de las clases `Paddle` y `Ball` es la misma que en la práctica 1, con la salvedad de que ahora heredan parte de ella de `MovingObject`, añadiendo los atributos y métodos que sean necesarios en cada caso, y (re)definiendo según corresponda los métodos heredados. La clase `Reward` modela a los premios del juego y también hereda de `MovingObject`, (re)definiendo los atributos/métodos que sean necesarios según corresponda. Se darán detalles en clase sobre la forma de gestionar los distintos tipos de premios y sus acciones asociadas.

Clase Game: Añade el soporte necesario para llevar a cabo las nuevas funcionalidades indicadas. En concreto, ahora los objetos del juego deben guardarse en una lista polimórfica (tipo `list<ArkanoidObject*>`) y por tanto los recorridos y búsquedas sobre ella deben hacer uso de iteradores.

Jerarquía de excepciones

Debes implementar al menos las siguientes clases para manejar excepciones:

ArkanoidError: Hereda de *logic_error* y sirve como superclase de todas las demás excepciones que definiremos. Debe proporcionar por tanto 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 `ArkanoidError` y se utiliza para todos los errores relacionados con la inicialización y uso de SDL. Utiliza las funciones `SDL_GetError`, `IMG_GetError` y `TTF_GetError` (si corresponde) para obtener un mensaje detallado sobre el error de SDL que se almacenará en la excepción.

FileNotFoundError: Hereda de `ArkanoidError` y se utiliza para todos 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 `ArkanoidError` y se utiliza para los errores provocados en la lectura de los ficheros de nivel y de partida guardada por formato incorrecto. Debes detectar al menos 3 errores diferentes de formato (tamaño incorrecto del mapa, valor de color incorrecto, etc.).

Formato de ficheros

Los ficheros (en formato texto) con partidas guardadas empezarán con la información propia del objeto `Game` (el nivel, la puntuación, etc), viniendo a continuación secuencialmente los objetos de la escena del juego (mapa de bloques, paredes, plataforma, pelota y premios) cada uno con su formato correspondiente (con sus partes comunes de acuerdo a la jerarquía de objetos). En los casos en los que haga falta saber cuántos objetos de un determinado tipo hay, se indicará con un número entero antes de la secuencia de objetos en cuestión. En la parte obligatoria esto solo ocurre con los premios. Por tanto, después de la pelota, y antes del primer premio, vendrá un número entero indicando cuántos premios vienen a continuación. La lectura/escritura de cada objeto se realizará mediante la llamada polimórfica al método `loadFromFile`/`saveToFile` del objeto en cuestión.

Ciclo de vida de los premios

Cuando el juego detecta una colisión de la pelota con un bloque (método `Game::collides`), con una cierta probabilidad, se creará un objeto de tipo `Reward` de un tipo de premio aleatorio en la posición en la que estaba el bloque, que se añadirá a la lista de objetos de la escena. A partir de ese punto, el objeto toma el control de su existencia y el juego simplemente realiza sobre él las llamadas a los métodos `render` y `update`. Cuando el premio llegue a la zona de la plataforma, éste le preguntará al juego si hay colisión contra la plataforma, y en tal caso hará la llamada al método del juego que implemente la acción asociada al premio. Finalmente, el propio premio solicitará al juego que le elimine de la escena.

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 (mediante `typeid`).
- 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 );
  ```

  Para obtener información más detallada y legible debes incluir en todos los módulos el fichero `checkML.h` (disponible en la plantilla en el proyecto `HolaSDL`).
- 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. 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)

- Implementa más clases de premios. Algunos ejemplos: Multiplicación de la pelota, plataforma con pegamento (la pelota siempre queda pegada y se lanza con la pulsación de una tecla), plataforma con cañón láser que dispara mediante una tecla, etc.
- Monstruitos (o naves enemigas): Implementa el soporte para que vayan apareciendo monstruitos moviéndose por el espacio de la pantalla (al estilo del juego original).
- Implementa el soporte para que los parámetros de configuración del juego (tamaño de la ventana y objetos, velocidades del juego, información de ficheros de texturas, etc.) se lean desde un fichero. De esta forma se podrían cambiar parámetros de configuración sin necesidad de recompilar nada.
- Utiliza el paquete `TTF` de SDL para manejar los distintos textos (incluyendo el contador) que se utilizan en el juego. Se darán detalles en clase.

Entrega

En la tarea del campus virtual *Entrega de la práctica 2* y dentro de la fecha límite (3 de diciembre), cada uno de los miembros del grupo, debe subir un fichero comprimido (.zip) que contenga los archivos de código .h y .cpp del proyecto, las imágenes que se utilicen, y, 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 (individuales) sobre la implementación. Las entrevistas se realizarán en las dos sesiones de laboratorio siguientes a la fecha de entrega, o si fuese necesario en horario de tutorías.

Entrega intermedia el 20 de noviembre: Un 20 % de la nota se obtendrá mediante una entrega intermedia, que tendrá lugar en la sesión de laboratorio del día 20 de noviembre, en la que el profesor revisará el estado actual de vuestra práctica. En particular, se espera que vuestra práctica integre ya de manera adecuada la jerarquía de objetos indicada (aún sin ninguna funcionalidad adicional respecto a la práctica 1) y la lista polimórfica de objetos en la clase `Game`