Práctica 2: Arrows 2.0

Curso 2019-2020. Tecnología de la Programación de Videojuegos 1. UCM

Fecha de entrega: 27 de noviembre de 2019

El objetivo fundamental de esta práctica es introducir el uso de 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/extensiones:

1. En la escena habrá también, a parte de los elementos de la práctica 1, una serie de mariposas volando de manera aleatoria por la pantalla. Al empezar un nivel aparecerá un número determinado de mariposas (en el nivel 1 serán 10 por ejemplo), y cuando una (punta de) flecha impacte con una mariposa, ésta morirá, restándose puntos (la mitad de los que suma explotar un globo). La partida ahora también acabará cuando no queden mariposas en la escena.

2. Cuando se explote un globo, aleatoriamente y con una cierta probabilidad (por ejemplo con 1/3), se generará, en la posición donde estaba el globo, un premio que va dentro de una burbuja, que irá descendiendo por la escena hasta desaparecer por la parte inferior. La burbuja se explotará si una punta de flecha impacta con ella liberando entonces el premio, que solo entonces podrá cogerse pulsando encima de su textura con el ratón. Habrá distintos tipos de premios (obligatoriamente al menos 2 distintos), cada uno con su letra y comportamiento correspondiente. Algunos ejemplos de premios podrían ser: flechas extras, flecha más grande, flecha más pequeña, paso de nivel, etc.

3. Al llegar a un número determinado de puntos (por ejemplo 100) el juego pasa de nivel. Cada nivel podrá tener un fondo distinto, distintos números de flechas, mariposas, distintas frecuencias de generación de globos, distintas velocidades máximas de globos y mariposas, etc. Puede haber un número máximo de niveles, y al pasar el último el juego acaba, o bien pasar de niveles de forma cíclica.

4. A parte de los puntos que se obtienen con cada explosión de globo, se obtendrán puntos extra cuando una misma flecha explote más de un globo. En concreto, se obtendrán como puntos extra $(\texttt{numHits}-1)^2 * \texttt{POINTS\_PER\_BALLOON}$, siendo `numHits` el número de globos que la flecha ha explotado, y `POINTS_PER_BALLOON` el número de puntos por explotar un globo.

5. El marcador que incluye los puntos actuales y las flechas disponibles debe visualizarse en la escena (como puede verse en el vídeo de demostración de la práctica 1).

6. Las partidas pueden guardarse y cargarse: Al iniciar el juego aparecerá un menú con dos opciones, jugar y cargar partida. 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.

Detalles de implementación

Cambio de nivel

Como se ha indicado, cuando se llega a una determinada puntuación (por ejemplo 100) o, en caso de existir, cuando se coge el premio de paso 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 determinar las características del nuevo nivel, y en particular, para indexar la textura del fondo correspondiente (usa para ello una suma sobre el tipo enumerado de los nombres de texturas). Al pasar de nivel se deben destruir todos los objetos de la escena excepto el marcador y el arco.

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 (excepto en clases sin nada de implementación, que solo tendrán fichero .h).

Clases GameObject, ArrowsGameObject 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 una aplicación SDL (métodos abstractos `render` y `update`, y destructora virtual). La clase `ArrowsGameObject`, también abstracta (en este caso por no tener un constructor público), hereda de `GameObject` y declara, y también define de forma genérica toda aquella funcionalidad común a los objetos del juego Arrows (métodos `render`, `update`, `getDestRect`, `getCollisionRect`, `loadFromFile` y `saveToFile`). Incluye atributos para la posición, el ancho, el alto, la dirección/velocidad de movimiento, un puntero a la textura del objeto, un puntero al juego, y finalmente, un iterador apuntando a la posición del objeto en la estructura de datos del juego. Define una constructora que inicializa todos los atributos a partir de los parámetros correspondientes, excepto el iterador, que se establecerá mediante la llamada al método `setItList`, que también debes definir (observa que cuando se construye el objeto aún no se dispone del iterador de su posición en la lista, y por tanto éste tomará valor mediante la llamada a dicho método).

Clase EventHandler: Se trata de una clase abstracta que simplemente declara el método virtual puro `handleEvent`. Es por tanto lo que se llama una interfaz. Las clases que capturen eventos, en concreto, `Bow` y `Reward` heredarán de ella (haciendo herencia múltiple), proporcionando una implementación concreta del método. De esta manera, para el manejo de eventos, el juego tendrá una lista de `EventHandler*`, que contendrá los objetos que pueden tratar eventos, y que se recorrerá llamando para cada objeto al método `handleEvent`.

Clases Bow, Arrow y Balloon: Su significado y funcionalidad es básicamente la misma que en la práctica 1, con la salvedad de que ahora heredan parte de ella de `ArrowsGameObject`, añadiendo los atributos y métodos que sean necesarios en cada caso, y (re)definiendo según corresponda los métodos heredados. Una diferencia respecto a la práctica 1 es que los métodos `update` son en este caso todos `void` (ver más abajo la sección sobre la eliminación dinámica de objetos).

Clase ScoreBoard: Se corresponde con el marcador del juego. Hereda de `GameObject` e implementa correspondientemente los métodos `render` y `update` (este último posiblemente vacío). Incluye como atributos los puntos y el número de flechas.

Clase Butterfly: Modela a las mariposas del juego heredando de `ArrowsGameObject` y redefiniendo convenientemente los métodos `update` y `render`.

Clase Reward: Modela a los premios del juego y hereda de `ArrowsGameObject` y de `EventHandler`. (Re)define por tanto los métodos `update`, `render` y `handleEvent`. Añade al menos un nuevo atributo para el estado del premio (si está dentro de una burbuja o no). 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<GameObject*>`) y por tanto los recorridos y búsquedas sobre ella deben hacer uso de iteradores. Es conveniente guardar una segunda copia de los punteros a ciertos objetos de manera explícita y separada, como por ejemplo el marcador o las flechas (para la implementación de las colisiones), en este caso mediante otra lista (tipo `list<Arrow*>`). El juego también debe incluir una lista polimórfica (tipo `list<EventHandler*>`) para manejar la gestión de eventos. Finalmente, como se explica más abajo, incluye una lista con los iteradores a los objetos que corresponde destruir al acabar el bucle de actualizaciones.

Eliminación dinámica de objetos

Como en la práctica 1, un objeto puede saber que debe dejar de existir en su método `update`. En tal caso, en lugar de devolver un booleano, se invocará al método `killObject` del juego quien llevará a cabo el borrado del objeto. Para no borrar al objeto mientras éste sigue ejecutando su método `update` (lo que podría fácilmente ocasionar errores de accesos no válidos a memoria), el juego guardará una lista con las posiciones de los objetos (como iteradores) que deben borrarse en cuanto acabe el bucle de actualizaciones. Para evitar búsquedas innecesarias, cada objeto del juego guardará un iterador a su posición en la lista de objetos, que le pasará como parámetro a la llamada a `killObject`, y que precisamente se almacenará en la lista mencionada para su posterior borrado.

Jerarquía de excepciones

Debes implementar al menos las siguientes clases para manejar excepciones:

ArrowsError: 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 `ArrowsError` 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 `ArrowsError` 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 `ArrowsError` y se utiliza para los errores provocados en la lectura de los ficheros de datos del juego (fichero de partida guardada, y en caso de usarlo, fichero de nivel). Debes detectar al menos dos tipos de errores de formato.

Guardado/carga de partidas

Los ficheros (en formato texto) con partidas guardadas empezarán con la información propia del objeto `Game` (el nivel, la puntuación, etc), viniendo a continuación el número de objetos de la escena y secuencialmente cada objeto (arco, flechas, globos, premios, etc.) cada uno con su formato correspondiente (con sus partes comunes de acuerdo a la jerarquía de objetos). La lectura/escritura de cada objeto se realizará mediante la llamada polimórfica al método `loadFromFile`/`saveToFile` del objeto en cuestión. Opcionalmente puedes usar una constructora que reciba un flujo de entrada como parámetro en lugar del método `loadFromFile`.

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. 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/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 relevante 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.
- 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.
- Implementa el soporte necesario para que el menú inicial se gestione desde la propia ventana SDL y que no haya que ir a la consola para ninguna acción (incluyendo la lectura del código numérico de los ficheros).
- 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 (27 de noviembre), cada uno 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. Ese mismo texto debes subirlo también en el cuadro de texto (sección "texto en línea") asociado a la entrega.

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. 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 12 de noviembre: Una parte de la nota (no más del 20 %) se obtendrá mediante una entrega intermedia, que tendrá lugar en la sesión de laboratorio del día 12 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`.