Tarea 2

Bomberman

CC3501 - Modelación y Computación Gráfica para Ingenieros

Autor:	Alejandro González
RUT:	19.344.991-3
Profesor(a):	Nancy Hitschfeld
Auxiliares:	Pablo Pizarro
	Pablo Polanco

Índice

1. Descripción del Problema

Se desea llevar a cabo la implementación de uno de los clásicos del arcade, Bomberman, y todo lo que la lógica tras el mismo conlleva. Para ello se desea proceder bajo el paradigma del modelo vista controlador (MVC), definiendo bien los distintos elementos pertenecientes al juego.

Se tiene, además, una serie de restricciones y requerimientos en cuanto a la parte gráfica del mismo, que implican el uso de OpenGL, de la mano de Pygame, para darle mayor eficiencia, en cuanto a tiempo de ejecución del programa, además de una mayor fluidez.

Finalmente, en cuanto al modelo, se tienen ciertos atributos que los entes del juego han de cumplir, en interacción con las acciones realizadas por el usuario, mediante el controlador.

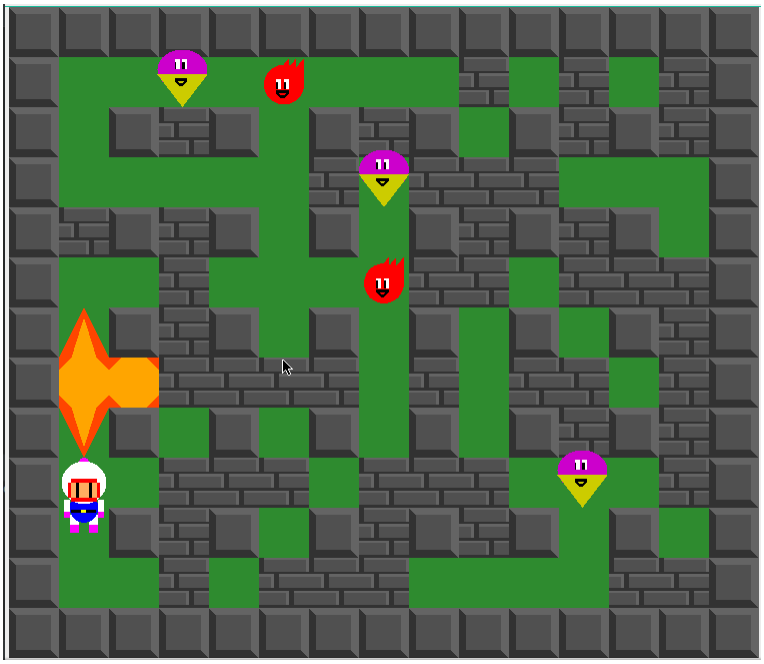

Figura 1: Bomberman Gameplay

2. Descripción de la Solución

Para efectuar lo solicitado, previo a la programación, se decidió utilizar el paradigma de programación orientada a objetos (OOP), puesto que permite realizar las interacciones entre los distintos personajes del modelo de una forma más natural. Se identificó el problema de tener referencias circulares dentro de las clases, por lo que se decidió crear una clase:

- **Characters**: que contiene todos los personajes del juego, estáticos o no, de manera que puedan interactuar implicitamente entre ellos sin tener que importarse de manera circular. A su vez, cada personaje tiene una referencia a 'Characters', de manera que puede acceder a todos los personajes que en ella yacen, de una manera simple y menos engorrosa.

El sistema de colisiones asume que todos los elementos del juego son rectángulos, independientemente de su forma en la pantalla. Por lo que cada personaje del juego posee una lista de rectángulos que lo representa implícitamente, para verificar si colisiona o no otro personaje. Para saber, si se pueden hacer ciertos movimientos, de delegó la tarea a otra clase llamada Physics. En resumen, se utilizó:

- **Rectangle**: Clase que representa rectángulos dentro de una grilla discreta (en este caso, pero es extendible). Utilizando simplemente dos puntos de la clase Vector (otorgada por el equipo docente). De manera que permite verificar de manera eficiente si dos rectángulos se overlapan o no.
- **Physiscs**: Clase que permite discretizar el modelo tras el juego, independientemente de la resolución de la pantalla, permite convertir desde su propia unidad discreta a pixeles y viceversa, además de encargarse de guardar todos los rectángulos de los personajes del juego, para poder verificar colisiones de manera más facil. Chequear si un movimiento es válido o no, o si un personaje dinámico puede moverse en una cierta dirección. Finalmente y una de sus tareas más importantes es hacer una lista de rectángulos disponible dentro de la grilla, para poder asignar posiciones aleatorias para los enemigos y bloques destructibles de manera rápida, escogiendo un número entre cero y el largo de dicha lista, eliminando los rectagulos que vayan siendo utilizados.

Entre los elementos estáticos del juego, se encuentran:

- **DBlock (Destructive Blocks)**: que representan a los bloques destructibles del juego. Son eliminados una vez que un elemento de la clase Fire le da la orden de destruirse, esto debido a que está en su radio de fuego. No es necesario asignarle una posición, puesto que toma la lista de posiciones disponibles otorgada por la clase *Physics* y escoge una aleatoriamente, eliminándola de la lista.

Figura 2: Destructive Block.

- **Grid**: representa a todos los bloques no destructives del juego, guardándolos en su lista de rectángulos. En general, para tener una grilla lo más parecida al juego original, se le debe asignar valores impares, tanto a la cantidad de bloques horizontal, como vertical.
- **Fire**: aparece después de que una bomba explota, se expande en 4 direcciones de acuerdo al radio que poseen las bombas puestas por bomberman. Si existe un bloque destructible en alguna dirección, termina de expandirse y destruye el bloque. Su duración es de 2 segundos con respecto al timer del juego.

Figura 3: Fuego

- **Bomb**: son las bombas puestas por bomberman. Una vez pasados 3 segundos desaparece y genera un fuego en su posición. Bomberman puede caminar sobre ella hasta que sale de su rectángulo deja de overlapar el de la bomba (una vez puesta), luego actúa como muro.

Figura 4: Bomba

- **Exit**: clase de un solo rectángulo que permite a bomberman salir de la pantalla, una vez que no hayan enemigos en la pantalla.

Figura 5: Salida

- **PowerUp**: activan ciertas mejoras para Bomberman, las 3 implementadas son:
  - **Radius**: un mayor radio para las bombas, con un radio máximo de 10.
  - **Bomb** una bomba adicional, con un límite de 10 bombas.
  - **Speed**: duplica la velocidad del personaje, con un máximo del largo del paso del personaje divido por 4.

Figura 6: A la izquierda el powerup de radio, al medio el de bombas extras y a la derecha el de velocidad.

No es necesario asignarle una posición en el constructor, pues toma la lista de bloques destructibles y escoge uno de ellos al azar, para posicionarse bajo el.

Ahora bien, en cuanto a los personajes dinámicos del juego, se encuentran:

- **Bomberman**: Personaje que responde a las teclas presionadas por el usuario, para moverse y colocar bombas. Si es tocado por un fuego una vez dado un paso, muere y desaparece al mismo tiempo que

el fuego se extingue. Si es impactado por un enemigo, muere tras un delay de dos segundos. Tambien es capaz de matar enemigos y destruir bloques utilizando sus bombas. Una vez elimina a todos sus enemigos, puede salir de la etapa utilizando la salida generada.

Figura 7: Bomberman

En cuanto a su movimiento, cada vez que se clickea una tecla para moverse en una dirección, se mueve siempre la misma cantidad de puntos definidos en la grilla discreta, de manera que, independientemente de la velocidad otorgada, pueda pasar entre las distintas aberturas de la grilla.

Posee un contador de bombas, el cual decrece cuando coloca una, y aumenta en uno cuando una explota. Además de almacenar el radio actual de las mismas, que puede crecer con los power ups.

- **Enemy**: Representan a los enemigos de Bomberman, se mueven en linea recta hasta encontrar un obstaculo, escogiendo de manera aleatoria su próxima dirección de desplazamiento y moviéndose en aquella dirección (si puede, claro).

Figura 8: Enemigos

Para cada elemento del modelo se realizó su diseño en OpenGL, que eran básicamente estáticos y sin movimiento mas que traslaciones en la pantalla.

Se decidió agregar la música original del juego (correspondiente a Super Bomberman de SNES).

3. Discusión sobre las dificultades encontradas

Si bien parece un juego simple, las mecánicas tras él son variadas, teniendo cada una su propia complejidad, por lo que es menester pensar bien qué hacer antes de comenzar a implementar.

Se puede mencionar:

- La primera complejidad que se tuvo fue el hecho de trabajar sobre pixeles o con una unidad discreta, paralela a los pixeles de la pantalla, y que por motivos de extensibilidad del proyecto se optó por lo último. Añadiendo más complejidad inicialmente, pero que a la larga ahorraba mucho tiempo, puesto que se trabajó con una unidad de medida standard, correspondiente a las constantes definidas inicialmente en el constructor de la clase *Physics*.
- Otro problema que se tuvo, fue la organización de los distintos tipos de elementos. En efecto, se debía tener una lista de enemigos, una lista de bombas, una lista de fuegos, de bloques destructibles, etc. Por lo que se creo la clase *Characters*, que llevaba registro de todo ello. Por otra parte, la clase *Physics* llevaba registro de todos los rectángulos pertenecientes a cada tipo de personaje en una lista, así como cada personaje guardaba sus propios rectángulos, para despues, al momento de ser destruído, saber cuales destruir en Physics.
- El movimiento de los personajes dinámicos fue un problema luego de que estos aumentasen su velocidad, por lo que se decidio establecer un paso de largo fijo cada vez que se movían, y al momento de aumentar la velocidad, simplemente aumentaba la velocidad al realizar dicho paso, de manera que la posición de los personajes siempre calce con el inicio y fin de los bloques de la grilla. Notar que los personajes no responden a interacciones con el usuario mientras estén realizando dicho paso, en efecto, la ignoran y siguen moviéndose en la última dirección solicitada, hasta completar el largo del paso.
- El tiempo del juego fue otro inconveniente. Inicialmente se utilizaba el tiempo de la máquina (con la clase datetime), pero luego se reemplazó, para tener un juego determinístico, en el sentido de que independiente de los fps, siempre ocurran las acciones según el tiempo establecido.
- Escoger la posición aleatoria. Inicialmente, cada objeto que debía obtener una posición aleatoria, revisaba todos los posibles rectángulos que podía tomar y luego escogía uno al azar. Algoritmo reemplazado por tener una lista de todas las posiciones disponibles en la clase *Physics*, escogiendo una de la lista, al azar.
- Poner la bomba centrada en la dirección de Bomberman. Puesto que se debió tomar en cuenta además qué tan adentro de los rectángulos se estaba.
- Los diseños en OpenGL, algunos requirieron desempolvar ciertos conocimientos de geometría en coordenadas polares y combinar ello con coordenadas cartesianas, calculando muchas veces mal las traslaciones.

4. Conclusiones

Se aplicaron los distintos conceptos aprendidos en cátedra, tales como el paradigma de modelo vista controlador, para la organización del juego. Por otra parte, se reutilizó código aportado por los auxiliares, eliminando ciertas cosas que si bien eran útiles, no se requirieron en este juego.

Se comprendió la eficiencia del dibujar utilizando OpenGL, pero a la vez, existe el trade off de velocidad en el procesamiento del juego, y por ende su tasa de refresco (que favorece al usuario), pero es lenta de escribir (que no favorece al desarrollador, vs la utilización de sprites, que en modelos más complejos puede afectar la fluidez, pero son mucho más faciles de implementar.

Finalmente, basta destacar la utililidad de desarrollar de manera iterativa, puesto que se puede comenzar desde un juego muy básico, agregando los distintos niveles de complejidad, ya sea en cuanto a los elementos de la vista, o requerimientos de jugabilidad, o las interacciones entre los elementos del modelo.