Entrega Avance Juego

BlockBreaker

Integrantes:

Matias Altamirano
Catalina Marin
Bastian Jorquera

Análisis del juego original

El juego BlockBraker original presenta 4 clases;

1. Block: Renderiza los bloques individuales, ve su color, su tamaño y su posición.
2. Paddle: Renderiza la plataforma, ve su color, su tamaño y su posición, además detecta las teclas del pc para moverse según lo pida el usuario.
3. PingBall: Renderiza la pelota, ve su color, su tamaño, su posición, su dirección al moverse, su velocidad, si está quieta o no, ve si colisiona con la plataforma o con los bloques.
4. BlockBreaker: Se encarga de ver las características de la pantalla, los textos que aparecen en ella, crea a los bloques, renderiza la posición inicial y los valores de todos los atributos de las clase, verifica si el nivel terminó o hay un game over.

Analizando qué mejoras se podían hacer al código a simple vista encontramos que el marcador de puntaje no se reseteaba una vez que se acababan las vidas, lo cual es un error, también vimos que las partidas eran muy largas y el rebote de la pelota era muy predecible, y por último vimos que faltaban pistas de audio.
A nivel de estructura de código se pudo observar que que PingBall y Block tenian un manejo de colisiones, y más evidente se observó que Block, Paddle y PingBall, tenian atributos en común; posición “x” e ”y”, altura y ancho, además todos se renderizar con el mismo comando, con esto se vió una redundancia en el uso de este código.
Con estas cosas en mente se implementaron varios cambios en el código abordando las problemáticas y relaciones anteriores.

Cambios realizados al juego

Se realizaron múltiples cambios en el sentido de organización del código y sus clases, pero al mismo tiempo fueron pocos los cambios perceptibles a nivel práctico en la jugabilidad, a continuación se presentarán los cambios realizados y las ventajas de estos cambios.

Cambios visibles y jugables:

Ya analizadas la problemáticas del juego base primero veremos cambios de jugabilidad respecto a la versión inicial de trabajo;

1. Actualmente si se muestra un conteo de puntos funcional, cuando las vidas llegan a 0 el puntaje se reinicia, esto se logra implementando un método de reinicio de datos cuando el juego acaba.

2. El segundo cambio fue que la pelota ahora rebota de forma aleatoria al chocar con la plataforma móvil, permitiendo que su rebote sea impredecible, con esto se vio acortado el tiempo de las partidas, ya que antes podía rebotar en un ángulo de poca

o demasiada inclinación de forma consecutiva, lo que generaba que la pelota fuera con una trayectoria muy predecible y poco cambiante. También se ajustó la velocidad de la pelota, su tamaño y se arreglo el cambio de color de la misma. Son pequeños cambios pero el objetivo es hacer más divertido y disfrutable el juego, lo que consideramos que se cumplió.

Mejoras a nivel de código:

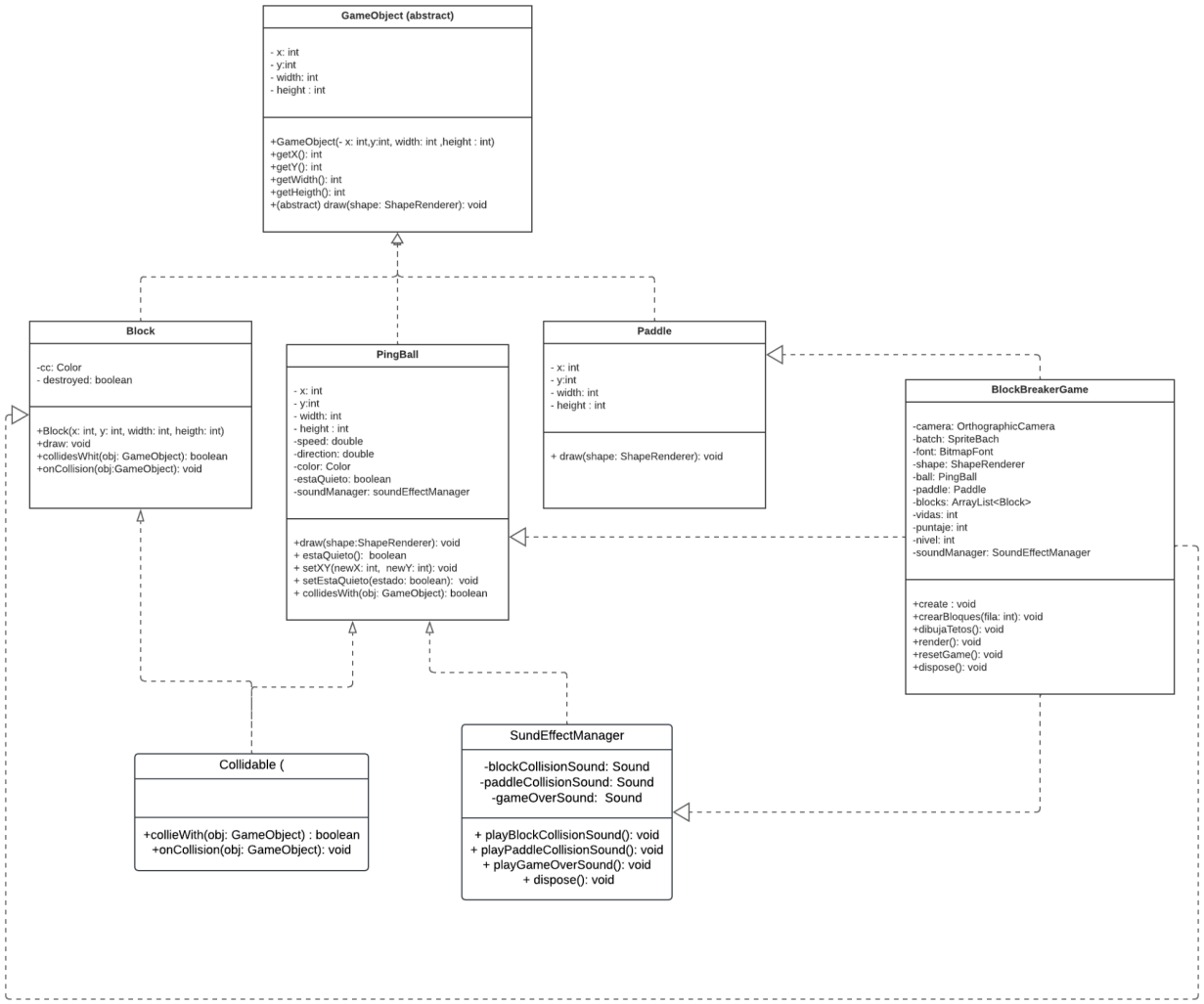
Ahora enfocándonos a nivel de código en el ámbito jugable tenemos que:

1. El primer cambio es que originalmente la pelota usaba la misma variable para la velocidad y la dirección, con esto si por ejemplo se quería aumentar solo la velocidad de la pelota en ciertas circunstancias como niveles o modificadores, resultaba imposible de realizar sin tocar el código de la trayectoria que seguía la pelota. Así que se separó en 2; las mencionadas velocidad y dirección, pudiendo ahora trabajar de manera más flexible y personalizada respecto a el movimiento de la pelota.
2. Relacionado con el primer cambio, ahora separadas las variables de velocidad y dirección se pudo implementar que la pelota rebotara de forma aleatoria al chocar contra la plataforma móvil, modificando solo la dirección, solucionando algunos problemas observados en el código original a nivel jugable.

Ahora metiéndonos en la estructura de las clases:

1. Se creó la clase abstracta GameObject con el objetivo de unificar clases que tenían cosas en común, las clases: Block, Paddle y PingBall ahora se extienden de la clase GameObject, y usan su constructor super, getters y setters para asignar sus características y usan el renderizado de también la clase abstracta. El uso de esto nos permitió ahorrar código y mejorar el entendimiento y características de los objetos.
2. También se agregó la interfaz Collidable, la cual es implementada por las clases Block y PingBall al igual que la clase abstracta, la idea es que esta interfaz se implementa para detectar las colisiones de los objetos que la usan sin importar contra qué objeto choquen. Una pequeña implementación es que se pueden ver en las colisiones cuando la pelota choca con los bloques se pone roja, y cuando choca con la plataforma se pone verde.
3. Se creó la clase SoundEffectManager la cual es implementada por la clase PingBall para agregar sonido cada vez que rebota en una superficie. Posee un efecto de sonido al rebotar con Paddle y uno diferente al romper un bloque. También es utilizada por la clase BlockBreakerGame, en la cual se aplica un efecto de sonido cuando se detecta que el contador de vidas llegó a 0.

Diseño diagrama UML



Descripción de diagrama UML

Este diagrama UML busca describir un sistema de juego de bloques, representado por varias clases y sus relaciones, a continuación se explicara cada una de las clases, sus atributos y métodos principales:

1. GameObject (abstracto):

    - Representa la clase base de todos los objetos del juego que tienen posición y tamaño
    - **Atributos :** x, y (posiciones), width, height (dimensiones)
    - **Métodos:**

        - Métodos para obtener las coordenadas y dimensiones : getX(), getY(), getWidth(), getHeight();

        - Método abstracto para dibujar el objeto: draw(ShapeRenderer)

2. Block:
    - Hereda de GameObject y representa un bloque destructible
    - **Atributos :**
        - Color del bloque: cc: Color
        - Estado de destrucción del bloque: destroyed: boolean
    - **Métodos:**
      -Dibuja el bloque: draw(ShapeRenderer)
      -Verifica colisiones con otros objetos: collie With(GameObject)
      -Maneja las acciones al colisionar, cambiando el estado del bloque: onCollision(GameObject)
3. Paddle:
    - Representa la paleta controlada por el jugador
    - **Atributos :** Herenda x, y (posiciones), width, height (dimensiones)
    - **Métodos:**
      -Dibuja la paleta en la pantalla
4. PingBall:
    - Representa la pelota del juego
    - **Atributos :**
      -Velocidad de la pelota: speed: double
      -Dirección en grados: direction: double
      -Color de la pelota: color: Color
      -Estado inicial (quieta o en movimiento): estadoQuieto: boolean
      -Administra los efectos de sonido entre las colisiones: soundManager: SoundEffectManger

    - **Métodos:**
      -Dibuja la pelota: draw(ShapeRenderer)
      -Retorna si la pelota está quieta: estaQuieto()
      -Actualiza la posición:setXY(int, int)
      -Cambia el estado de movimiento de la pelota: setEstaQuieto (boolean)

-Verifica colisiones: collidesWith(GameObject)
-Cambia el color y dirección en función del tipo de colisión: onCollision(GameObject)

5. Collidable (Interfaz):
    - Define métodos para detectar y manejar colisiones
    - **Métodos:**
      -Verifica si colisiona con otro objeto: collidesWith(GameObject)
      -Define la acción a tomar en caso de colisión: onCollision(GameObject)

6. SoundEffectManager:
    - Gestiona los efectos de sonido
    - **Atributos:**
      -Sonidos para colisiones y fin de juego: blockCollisionSound: Sound, paddleCollisionSound: Sound, gameOverSound: Sound
    - **Métodos:**
      -Reproduce los sonidos correspondientes: playBlockCollisionSound(), playPaddleCollisionSound(), playGameOverSound()
      -Libera recursos de sonido: dispose()

7. BlockBreakerGame:
    - Controla la lógica del juego principal
    - **Atributos:**
      -Herramientas para el renderizado: camera: OrthographicCamera, batch: SpriteBatch, font: BitmapFont, shape: ShapeRenderer
      -Elementos del juego: ball: PingBall, paddle: Paddle, blocks: ArrayList<Block>:
      -Variables de estado del juego: vidas: int, puntaje: int, nivel: int
      -Gestiona los sonidos del juego: soundManager: SoundEffectManage
    - **Metodos:**
      -Inicializa el juego: create()
      -Crea una fila de bloques: crearBloques(int fila)
      -Dibuja el texto en pantalla:dibujaTetos()
      - Renderiza el juego: render()
      - Reinicia el juego: resetGame()
      - Libera recursos del juego: dispose()

Este diagrama UML busca ofrecer una vista integral del juego destacando las relaciones, responsabilidades y comportamiento de cada clase necesarios para este.

Descripción del usuario

El juego consiste en lanzar una pelota, la cual debe colisionar con ciertos bloques de colores que se encuentran en la parte superior de la pantalla, al impactar con los

mencionados bloques o con los bordes de la pantalla la pelota rebotará en sentido opuesto, en caso de impactar con los bloques estos se romperán y otorgarán puntaje, también se buscará que la pelota no llegue al límite inferior del área de juego o se perderá una vida. Para evitar perder hay una plataforma que el jugador mueve de derecha a izquierda buscando interceptar la bola y evitar que toque el borde inferior, el movimiento se realiza por teclado por parte del usuario, se busca que las teclas sean intuitivas y simples en su uso. Los bloques te dan puntaje cada vez que los rompes y el objetivo final del juego es conseguir la mayor cantidad de puntaje sin que se “caiga” la bola.

Instrucciones de uso para el usuario:

El juego comienza con una plataforma con la bola encima, en la parte superior aparecerán una serie de bloques de colores, en la esquina inferior izquierda aparece un contador de puntos y en la esquina inferior derecha está el número de vidas disponibles.

Paso 1: El jugador puede mover la plataforma con las flechas direccionales de izquierda y derecha, luego para lanzar la pelota debe presionar la barra espaciadora del teclado, luego de eso la pelota comenzará a desplazarse y chocará con los bloques y límites de la pantalla.
Paso 2: Cada vez que la bola se acerque a la parte inferior de la pantalla el jugador debe interceptarla con la plataforma, moviéndola de izquierda a derecha con las flechas del teclado respectivas. En el caso de no poder interceptar la pelota “caerá”, se perderá una vida y se volverá al estado del paso 1.

Si las vidas llegan a 0, el juego se habrá acabado, el puntaje se reiniciará y podrás volver a jugar para buscar tener un mayor puntaje.

Descripción de programador y función de cada clase

El código usado usa 5 clases y una interfaz, a continuación se mencionan y se mencionan las clases, su funcionalidad en el código.

1. Interface Collidable: Esta interfaz es usada por los objetos que pueden colisionar y usa 2 métodos que se usan para detectar colisiones.

2. Clase GameObject: Esta es una clase abstracta que conserva 4 atributos usados por todos los demás objetos y su constructor, getters y setters, son; posición en “x”, posición en “y”, ancho y alto, además renderiza a el objeto en cuestión. uwu

3. Clase SoundEffectManager: Esta clase posee 3 métodos los cuales otorgan efectos de sonido diferentes según sea necesario, actualmente se usa para poner efectos de sonidos en las colisiones de la pelota con la plataforma y con los bloques, y también cuando se termina la partida.

4. Clase Block: Esta clase representa a los bloques que se cargan en la parte superior de la pantalla, se extiende de GameObject e implementa Collidable, se encarga de renderizar los rectángulos y sus colores aleatorios, gracias a la interfaz Collidable, detecta si choca contra la pelota y que se destruya si es el caso.

5. Clase Paddle: Esta clase se extiende de GameObject, se encarga de renderizar la plataforma y de su movimiento con las flechas del teclado.

6. Clase PingBall: Esta clase se extiende de GameObject e implementa Collidable, representa la pelota con la que se rompen los bloques, usa el constructor de GameObject y usa los atributos de velocidad, dirección, color y su estado de movimiento, este último muy útil para saber cómo se comportan las funciones del juego. Renderiza la pelota, ve las colisiones con los bloques y con la plataforma usando la interfaz Collidable y ve las colisiones con los bordes de la pantalla.

7. Por último tenemos a la clase BlockBreakerGame: Esta clase se extiende de AplicationAdapter de la librería de LibGDX para hacer más simple el diseño de comportamiento del juego, desde BlockBreakerGame se ve lo que es la pantalla, el almacenamiento de los bloques y su posición de creación, se dibujan los textos, si la pelota está quieta ve el comienzo del lanzamiento, verifica si la pelota toca el inferior de la pantalla, ve el decremento de vidas e incremento de nivel, remueve bloques destruidos y resetea el juego en caso de fin de la partida.

Como se pudo observar, el diseño de las clases consideramos que es coherente y tiene sus funcionalidades bien definidas, las tareas son específicas para el objeto o funcionalidad que representa en cada definición de clase. La clase abstracta GameObject nos permite representar la entidad general de objeto y permite heredar sus características y funcionalidades a el resto de objetos, en este caso; la pelota, la plataforma y los bloques, centralizando un código común para los objetos, también se ahorra código en constructores y métodos, permitiendo que los objetos se enfoquen en sus funcionalidades específicas. En cuanto a la interfaz podemos decir que ahora nos permite implementar colisiones con cualquier objeto que implemente la interfaz, sin importar la clase específica que tenga, haciendo el código más versátil y fácilmente escalable en este aspecto.

Escalabilidad y mantenibilidad

A continuación se hablará de buenas prácticas en el desarrollo del código, para su ordenamiento, legibilidad, mantenibilidad y escalabilidad futura.

Encapsulamiento y modularización:

En cuanto a el encapsulamiento del código, se cuidó que todos los atributos de las clases sean “private”, solo accesibles desde afuera con métodos getters y setters, los métodos son

públicos, y se implementó una modularización que nos permita modificar comportamiento clases sin afectar al resto, un ejemplo de esto es que como se mencionó en otro apartado se quiso modificar el comportamiento de la colisión de la plataforma con la pelota, para hacer este cambio solo fue necesario modificar la clase de la pelota, o si se quisiera cambiar la velocidad de la plataforma solo hay que modificar esa clase específica.

Herencia:

La herencia la podemos observar en clase GameObject, de la cual derivan características, constructores, getters y setters, y funcionalidades a los objetos hijos, permitiendo ahorrando código y facilitando la extensión de funcionalidades comunes para los objetos o nuevos objetos con un comportamiento o características en común.

Polimorfismo:

Con la integración de la interfaz de colisiones podemos observar el uso de polimorfismo, con una detección de colisiones que puede utilizar cualquier clase en la que colisione un objeto con cualquier otro objeto, y si se agrega una nueva clase si uno quiere solo hay que implementar la interfaz o la clase abstracta y ya estará dado su comportamiento y características, facilitando la escalabilidad y mantenimiento.

Resumen

En este informe de avance explicamos el análisis del juego original, viendo su estructura de código y cambios jugables que se vieron necesarios, y con eso determinamos las mejoras necesarias que se necesitaban, para luego ver como que se implementaron en el juego, cambiando mecánicas jugables y la organización y ordenamiento del código mismo, También vimos en más detalle esto último, mencionando y explicando las clases que se usaron con sus motivos y funcionalidades, desde un diseño UML a una explicación más detallada de su comportamiento. Se explicó el funcionamiento del juego para que el usuario que lo pruebe no tenga problemas en usarlo, y por último explicamos sus características en el contexto de la escalabilidad y la mantenibilidad del juego.

Próximas ideas mejoras

Para la entrega final del juego, inicialmente tenemos pensado agregar:

- Terminar de agregar la mecánica de niveles que aumentan en dificultad de forma progresiva.
- Agregar una pantalla de inicio y una pantalla de game over con el recuento de puntaje y nivel alcanzado.
- Agregar pantallas y música exclusivos por nivel.

Link de GitHub:

https://github.com/BastianJorquera/BlockBreaker