DIGITECH

MARCO TEÓRICO

LibGDX:

Descripción: LibGDX es un marco de desarrollo de código abierto utilizado para la creación de juegos en Java. Proporciona una amplia variedad de herramientas y utilidades que facilitan el desarrollo de juegos multiplataforma.
Uso en el Proyecto: En este proyecto, LibGDX se utiliza para gestionar gráficos, entrada del usuario, y otras funcionalidades esenciales para el desarrollo de juegos en 2D con Java.

Descripción de Clases:

FloppyDemo:

Descripción: Clase principal que inicia la aplicación. Contiene el bucle principal del juego y maneja la creación y disposición de la ventana de la aplicación.

StateManager:

Descripción: Gestiona el cambio entre diferentes estados del juego, como Menú, Juego y Game Over. Utiliza (Stack<State>) para organizar y gestionar los diferentes estados.

State:

Descripción: Clase abstracta que define el esqueleto básico para los distintos estados del juego. Contiene métodos abstractos para manejar entrada, actualización, renderizado y liberación de recursos.

MenuState:

Descripción: Representa el estado del menú principal del juego. Permite iniciar el juego al tocar la pantalla.

PlayState:

Descripción: Representa el estado principal del juego, donde ocurre la acción. Gestiona la lógica del juego, como el movimiento del pájaro, la generación de tubos, el puntaje y las transiciones de estado.

GameOverState:

Descripción: Representa el estado de "Game Over". Se activa cuando el pájaro colisiona con un tubo o el suelo. Muestra el puntaje y el top score, y permite regresar al menú principal al tocar la pantalla.

Bird:

Descripción: Clase que modela al pájaro del juego. Gestiona su posición, animación y capacidad para saltar.

Tube:

Descripción: Representa los tubos que el pájaro debe atravesar. Gestiona su posición, colisiones y reposicionamiento para crear un flujo continuo de obstáculos en el juego.
GameStateManager:

Descripción: Gestiona el flujo de estados del juego mediante una pila (Stack<State>). Permite apilar, desapilar, establecer y actualizar estados.

GameConstants:

Descripción: Clase que almacena constantes utilizadas en el juego, como dimensiones de pantalla, separación entre tubos y factores de puntuación.

PreferencesManager:

Descripción: Clase encargada de gestionar las preferencias del juego, como el top score. Utiliza la clase Preferences de LibGDX para almacenar datos de manera persistente.

BitmapFont:

Descripción: Clase utilizada para renderizar texto en el juego. Se utiliza para mostrar el puntaje y el top score en las pantallas del juego y de Game Over.

Texture:

Descripción: Clase que representa una textura en LibGDX. Se utiliza para cargar imágenes como el fondo, el pájaro, los tubos y las pantallas de Game Over y Menú.

MARCO PRÁCTICO

Clase FloppyDemo

```
package com.mygdx.game;

import com.badlogic.gdx.ApplicationAdapter;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.audio.Music;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;

import States.GameStateManager;
import States.MenuState;



public class FloppyDemo extends ApplicationAdapter {
   public static final int WIDTH = 480;
   public static final int HEIGHT = 800;
   public static final String TITLE = "Flappy Bird";

   private GameStateManager gsm;
   private SpriteBatch batch;
   private Music music;

   @Override
   public void create() {
      // Inicialización de elementos al inicio del juego
      batch = new SpriteBatch();  // Creación del objeto SpriteBatch para
renderizar gráficos
      gsm = new GameStateManager();  // Inicialización del GameStateManager
para gestionar los estados del juego
      music = Gdx.audio.newMusic(Gdx.files.internal("music.mp3"));  //
Carga de la música de fondo
      music.setLooping(true);  // Configuración para que la música se
reproduzca en bucle
      music.setVolume(0.1f);  // Ajuste del volumen de la música
      music.play();  // Inicio de la reproducción de la música
      Gdx.gl.glClearColor(1, 0, 0, 1);  // Establece el color de fondo al
limpiar la pantalla
      gsm.push(new MenuState(gsm));  // Inicia el juego con el estado del
menú
   }

   @Override
   public void render() {
      Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
      gsm.update(Gdx.graphics.getDeltaTime());
      gsm.render(batch);
   }

   @Override
   public void dispose() {
      super.dispose();
      music.dispose();
   }
}
```

Importes:

ApplicationAdapter: Es una clase proporcionada por LibGDX que implementa la interfaz ApplicationListener. Sirve como base para crear el juego y maneja eventos del ciclo de vida del juego, como create(), render(), y dispose().

Gdx: Proporciona acceso a las funciones específicas de la plataforma, como entrada de usuario, archivos, gráficos, etc.

Music: Representa una instancia de música en el juego. En este caso, se utiliza para reproducir la música de fondo del juego.

GL20: Proporciona constantes para trabajar con OpenGL ES 2.0, que es la API gráfica utilizada por LibGDX.

Texture: Representa una imagen 2D que puede ser utilizada para texturizar objetos en el juego.

SpriteBatch: Permite renderizar múltiples objetos gráficos en un solo lote, mejorando el rendimiento al reducir la cantidad de llamadas al sistema gráfico.

ScreenUtils: Proporciona utilidades para trabajar con la pantalla, como borrar la pantalla con un color específico.

GameStateManager: Una clase personalizada que gestiona los diferentes estados del juego, como Menú, Juego y Game Over.

MenuState: Representa el estado del menú principal del juego, donde los jugadores pueden iniciar una nueva partida.

Estos imports son esenciales para el desarrollo del juego, ya que brindan acceso a las funcionalidades clave de LibGDX y a las clases personalizadas que gestionan el flujo del juego.

Declaración de variables:

public static final int WIDTH = 480; y *public static final int HEIGHT = 800*: Estas son constantes públicas y estáticas que representan el ancho y alto de la ventana del juego. Al ser public, pueden ser accedidas desde otras clases sin necesidad de crear una instancia de la clase FloppyDemo. Al ser static final, son constantes que no pueden ser modificadas una vez que se les asigna un valor inicial.

public static final String TITLE = "Flappy Bird": Similar a las variables de tamaño, esta es una constante pública y estática que representa el título del juego. Al ser String, almacena el nombre del juego y, al ser static final, no puede ser modificado.

private GameStateManager gsm: Es un objeto privado de la clase GameStateManager. Al ser privado, solo puede ser accedido internamente por métodos de la propia clase. Este objeto se encarga de gestionar los diferentes estados del juego.

private SpriteBatch batch: Es un objeto privado de la clase SpriteBatch. Al ser privado, solo puede ser accedido internamente por métodos de la propia clase. Este objeto se utiliza para renderizar gráficos eficientemente al agrupar múltiples elementos gráficos en un solo lote.

private Music music: Es un objeto privado de la clase Music. Al ser privado, solo puede ser accedido internamente por métodos de la propia clase. Este objeto representa la música de fondo del juego.

El uso de public y private controla el nivel de acceso a estas variables. Las constantes public static final aseguran que estas propiedades sean accesibles y no modificables desde cualquier parte del código. Los objetos private garantizan que solo los métodos internos de la clase tengan acceso directo a ellos, siguiendo el principio de encapsulamiento.

Métodos:

@Override public void create(): Este método se llama automáticamente cuando se crea la aplicación. Aquí se realiza la inicialización de elementos al inicio del juego.

batch = new SpriteBatch(): Se crea un nuevo objeto SpriteBatch llamado batch. Este objeto se utiliza para renderizar gráficos eficientemente al agrupar múltiples elementos gráficos en un solo lote.

gsm = new GameStateManager(): Se crea un nuevo objeto GameStateManager llamado gsm. Este objeto se encarga de gestionar los diferentes estados del juego.

music = Gdx.audio.newMusic(Gdx.files.internal("music.mp3")): Se carga la música de fondo desde el archivo "music.mp3" y se crea un objeto Music llamado music para manejar la reproducción de la música.

music.setLooping(true): Se configura la música para que se reproduzca en bucle, lo que significa que continuará reproduciéndose una y otra vez.

music.setVolume(0.1f): Se ajusta el volumen de la música a 0.1 (10%). Puedes ajustar este valor según tus preferencias.

music.play();: Se inicia la reproducción de la música.

Gdx.gl.glClearColor(1, 0, 0, 1): Se establece el color de fondo al limpiar la pantalla con el color rojo (1, 0, 0, 1 en formato RGBA).

gsm.push(new MenuState(gsm)): Se agrega un nuevo estado al GameStateManager, en este caso, el estado inicial del menú.

@Override public void render(): Este método se llama automáticamente en cada fotograma y contiene la lógica del juego y el renderizado.

Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT): Limpia la pantalla con el color de fondo configurado en glClearColor.

gsm.update(Gdx.graphics.getDeltaTime()): Actualiza la lógica del GameStateManager, pasando el tiempo transcurrido desde el último fotograma.

gsm.render(batch): Renderiza los elementos gestionados por el GameStateManager utilizando el SpriteBatch batch.

@Override public void dispose(): Este método se llama automáticamente al cerrar la aplicación y se encarga de liberar recursos.

super.dispose(): Llama al método dispose de la clase padre.

music.dispose(): Libera la memoria utilizada por el objeto Music music.

Animation:

```
package sprites;

import com.badlogic.gdx.graphics.g2d.TextureRegion;
import com.badlogic.gdx.utils.Array;

public class Animation {
    private Array<TextureRegion> frames;  // Array para almacenar las
regiones de textura de cada fotograma
    private float maxFrameTime;  // Tiempo máximo para mostrar cada
fotograma
    private float currentFrameTime;  // Tiempo actual transcurrido
    private int frameCount;  // Número total de fotogramas en la animación
    private int frame;  // Fotograma actual

    public Animation(TextureRegion region, int frameCount, float cycleTime)
{
        frames = new Array<TextureRegion>();  // Inicialización del Array
        int frameWidth = region.getRegionWidth() / frameCount;  // Ancho de
cada fotograma
        for (int i = 0; i < frameCount; i++) {
            // Creación de los fotogramas a partir de la región de textura
original
            frames.add(new TextureRegion(region, i * frameWidth, 0,
frameWidth, region.getRegionHeight()));
        }
        this.frameCount = frameCount;
        maxFrameTime = cycleTime / frameCount;  // Cálculo del tiempo
máximo por fotograma
        frame = 0;  // Inicialización del fotograma actual
    }

    public void update(float dt) {
        // Actualización del tiempo actual transcurrido
        currentFrameTime += dt;
        if (currentFrameTime > maxFrameTime) {
            frame++;  // Avance al siguiente fotograma
            currentFrameTime = 0;  // Reinicio del tiempo actual
        }
        if (frame >= frameCount)
            frame = 0;  // Reinicio al primer fotograma si se alcanza el
último
    }

    public TextureRegion getFrame() {
        // Obtención del fotograma actual
        return frames.get(frame);
    }
}
```

Importes:

TextureRegion: Clase que representa una región de textura. En este contexto, se utiliza para definir las regiones de textura de cada fotograma en la animación.

Array: Una clase de contenedor proporcionada por LibGDX que implementa un array dinámico. Se utiliza aquí para almacenar las regiones de textura de cada fotograma en la animación.

Declaración de variables:

frames: Un array que almacena las regiones de textura de cada fotograma en la animación.

maxFrameTime: El tiempo máximo que se mostrará cada fotograma antes de pasar al siguiente.

currentFrameTime: El tiempo actual transcurrido desde el último cambio de fotograma.

frameCount: El número total de fotogramas en la animación.

frame: El fotograma actual que se está mostrando.

Métodos:

El método *update* se encarga de actualizar la animación según el tiempo transcurrido (dt). Incrementa el tiempo actual y, si supera el tiempo máximo permitido para un fotograma, avanza al siguiente. Si se llega al último fotograma, reinicia al primero.

El método *getFrame* devuelve la región de textura que representa el fotograma actual de la animación. Este método se utiliza para obtener el fotograma actual y renderizarlo en la pantalla.

Animation: este generador organiza la información necesaria para la animación dividiendo el área de textura en fotogramas y estableciendo parámetros básicos. Calcule el ancho de cada cuadro dividiendo el ancho total del área de textura por el número total de cuadros. Establezca las variables necesarias, como el número total de fotogramas, el tiempo máximo de visualización por fotograma (maxFrameTime) y el fotograma actual (S). Estos datos son importantes para garantizar que las animaciones se ejecuten sin problemas a lo largo del tiempo.

Bird:

```
package sprites;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.audio.Sound;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.TextureRegion;
import com.badlogic.gdx.math.Rectangle;
import com.badlogic.gdx.math.Vector3;

import org.w3c.dom.css.Rect;

public class Bird {
    private static final int GRAVITY = -15;  // Gravedad que afecta al
pájaro
    private static final int MOVEMENT = 100;  // Velocidad de movimiento
horizontal del pájaro
    private Vector3 position;  // Posición del pájaro en el mundo
    private Vector3 velocity;  // Velocidad del pájaro
    private Rectangle bounds;  // Área rectangular para las colisiones del
pájaro
    private Animation birdAnimation;  // Animación del pájaro
    private Texture texture;  // Textura del pájaro
    private Sound flap;  // Sonido al realizar el salto

    public Bird(int x, int y) {
        position = new Vector3(x, y, 0);  // Inicialización de la posición
del pájaro
        velocity = new Vector3(0, 0, 0);  // Inicialización de la velocidad
del pájaro
        texture = new Texture("birdanimation.png");  // Carga de la textura
del pájaro
        birdAnimation = new Animation(new TextureRegion(texture), 3, 0.5f);
// Creación de la animación
        bounds = new Rectangle(x, y, texture.getWidth() / 3,
texture.getHeight());  // Inicialización de las colisiones
        flap = Gdx.audio.newSound(Gdx.files.internal("sfx_wing.ogg"));  //
Carga del sonido del salto
    }

    public void update(float dt) {
        birdAnimation.update(dt);  // Actualización de la animación del
pájaro
        if (position.y > 0)
            velocity.add(0, GRAVITY, 0);  // Aplicación de la gravedad si
el pájaro está por encima del suelo
        velocity.scl(dt);  // Escalado de la velocidad por el tiempo
        position.add(MOVEMENT * dt, velocity.y, 0);  // Actualización de la
posición del pájaro
        if (position.y < 0)
            position.y = 0;  // Asegurarse de que el pájaro no se vaya por
debajo del suelo

        velocity.scl(1 / dt);  // Deshacer el escalado de la velocidad
        bounds.setPosition(position.x, position.y);  // Actualización de
las colisiones
    }

    public Vector3 getPosition() {
        return position;  // Obtención de la posición del pájaro
```

```
    }

    public TextureRegion getTexture() {
        return birdAnimation.getFrame();  // Obtención del fotograma actual
de la animación
    }

    public void jump() {
        velocity.y = 250;  // Configuración de la velocidad vertical para
el salto
        flap.play(0.5f);  // Reproducción del sonido del salto con volumen
reducido
    }

    public Rectangle getBounds() {
        return bounds;  // Obtención del área rectangular para las
colisiones
    }

    public void dispose() {
        texture.dispose();  // Liberación de la memoria utilizada por la
textura del pájaro
        flap.dispose();  // Liberación de la memoria utilizada por el
sonido del salto
    }
}
```

Importes:

com.badlogic.gdx.Gdx: Proporciona funcionalidades de utilidad para el desarrollo de juegos en LibGDX, como la gestión de archivos y la entrada del usuario.

com.badlogic.gdx.audio.Sound: Utilizado para cargar y reproducir sonidos en el juego.

com.badlogic.gdx.graphics.Texture: Representa una imagen bidimensional que se utiliza como textura en gráficos 2D.

com.badlogic.gdx.graphics.g2d.TextureRegion: Representa una región rectangular de una textura más grande. Se utiliza para gestionar las animaciones de sprites.

com.badlogic.gdx.math.Rectangle: Define un rectángulo en un plano 2D y se utiliza para las colisiones y la delimitación de objetos.

com.badlogic.gdx.math.Vector3: Representa un vector tridimensional y se utiliza para gestionar la posición y velocidad del pájaro en el espacio 3D del juego.

Declaración de variables:

GRAVITY: Constante que representa la gravedad que afecta al pájaro. La gravedad se aplica en el eje y y hace que el pájaro caiga.

MOVEMENT: Constante que representa la velocidad de movimiento horizontal del pájaro.

position: Vector tridimensional que almacena la posición del pájaro en el mundo.

velocity: Vector tridimensional que representa la velocidad del pájaro.

bounds: Rectángulo que define el área de colisión del pájaro.

birdAnimation: Objeto de la clase Animation que gestiona la animación del pájaro.

texture: Textura del pájaro.

flap: Sonido que se reproduce cuando el pájaro realiza un salto.

Métodos:

Constructor (public Bird(int x, int y)):
Parámetros: x y y representan las coordenadas iniciales en las que se crea el pájaro.
Inicializa la posición del pájaro utilizando un nuevo objeto Vector3.
Inicializa la velocidad del pájaro con un nuevo objeto Vector3 con valores iniciales en cero.
Carga la textura del pájaro desde el archivo "birdanimation.png".
Inicializa la animación del pájaro utilizando un nuevo objeto Animation. Se le pasa la región de textura, la cantidad de fotogramas y el tiempo de ciclo.
Inicializa el área de colisiones (bounds) utilizando un nuevo rectángulo con las coordenadas y dimensiones adecuadas.
Carga el sonido del salto desde el archivo "sfx_wing.ogg".

update(float dt):dt representa el tiempo delta, la diferencia de tiempo entre dos fotogramas.
Actualiza la animación del pájaro llamando al método update de la animación y pasando dt.
Aplica la gravedad al pájaro si su posición en el eje y es mayor que cero.
Escala la velocidad del pájaro por dt para tener en cuenta la diferencia de tiempo.
Actualiza la posición del pájaro basándose en su velocidad y la constante de movimiento horizontal (MOVEMENT).
Asegura que el pájaro no caiga por debajo del suelo ajustando su posición si es necesario.
Deshace el escalado de la velocidad para evitar acumulación de cambios de velocidad.
Actualiza las coordenadas del rectángulo de colisiones (bounds).

getPosition(): Retorna un objeto Vector3 que representa la posición actual del pájaro.

getTexture(): Retorna un objeto TextureRegion que representa el fotograma actual de la animación del pájaro.

jump():Configura la velocidad vertical del pájaro para simular un salto.
Reproduce el sonido del salto con un volumen reducido.

getBounds(): Retorna un objeto Rectangle que representa el área rectangular de colisiones del pájaro.

*dispose():*Libera la memoria utilizada por la textura del pájaro.
Libera la memoria utilizada por el sonido del salto.

Tube:

```
package sprites;

import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.math.Rectangle;
import com.badlogic.gdx.math.Vector2;

import java.util.Random;

public class Tube {
    public static final int TUBE_WIDTH = 52;  // Ancho de los tubos

    private static final int FLUCTUATION = 130;  // Variación en la
posición vertical de los tubos
    private static final int TUBE_GAP = 100;  // Espacio entre los tubos
    private static final int LOWEST_OPENING = 120;  // Altura mínima de
apertura para los tubos
    private Texture topTube, bottomTube;  // Texturas para la parte
superior e inferior del tubo
    private Vector2 posTopTube, posBotTube;  // Posiciones de la parte
superior e inferior del tubo
    private Rectangle boundsTop, boundsBot;  // Áreas rectangulares para
las colisiones
    private Random rand;  // Generador de números aleatorios


    public Tube(float x) {
        topTube = new Texture("toptube.png");  // Carga de la textura del
tubo superior
        bottomTube = new Texture("bottomtube.png");  // Carga de la textura
del tubo inferior
        rand = new Random();  // Inicialización del generador de números
aleatorios

        // Inicialización de las posiciones y áreas rectangulares de los
tubos
        posTopTube = new Vector2(x, rand.nextInt(FLUCTUATION) + TUBE_GAP +
LOWEST_OPENING);
        posBotTube = new Vector2(x, posTopTube.y - TUBE_GAP -
bottomTube.getHeight());
        boundsTop = new Rectangle(posTopTube.x, posTopTube.y,
topTube.getWidth(), topTube.getHeight());
        boundsBot = new Rectangle(posBotTube.x, posBotTube.y,
bottomTube.getWidth(), bottomTube.getHeight());
    }

    public Texture getBottomTube() {
        return bottomTube;  // Obtención de la textura del tubo inferior
    }

    public Texture getTopTube() {
        return topTube;  // Obtención de la textura del tubo superior
    }

    public Vector2 getPosTopTube() {
        return posTopTube;  // Obtención de la posición del tubo superior
    }

    public Vector2 getPosBotTube() {
        return posBotTube;  // Obtención de la posición del tubo inferior
```

```
    }

    public void reposition(float x) {
        // Reposicionamiento de los tubos al pasar por la pantalla
        posTopTube.set(x, rand.nextInt(FLUCTUATION) + TUBE_GAP +
LOWEST_OPENING);
        posBotTube.set(x, posTopTube.y - TUBE_GAP -
bottomTube.getHeight());
        boundsTop.setPosition(posTopTube.x, posTopTube.y);
        boundsBot.setPosition(posBotTube.x, posBotTube.y);
    }

    public boolean collides(Rectangle player) {
        // Verificación de colisiones con el jugador
        return player.overlaps(boundsTop) || player.overlaps(boundsBot);
    }

    public void dispose() {
        // Liberación de memoria utilizada por las texturas de los tubos
        topTube.dispose();
        bottomTube.dispose();
    }
}
```

Importes:

com.badlogic.gdx.graphics.Texture;

Este import proporciona acceso a la capa de textura en LibGDX. La capa de textura se utiliza para cargar y mostrar texturas, como en este caso las imágenes de tubería superior e inferior.

com.badlogic.gdx.math.Rectangle;

Este import trae consigo la clase Rectangle de LibGDX. La clase Rectangle se utiliza para representar áreas rectangulares y es útil para manejar colisiones. En este caso, se usará para definir las áreas rectangulares de los tubos superiores e inferiores.

com.badlogic.gdx.math.Vector2;

Este import importa la clase Vector2 de LibGDX. La clase Vector2 representa un vector de dos dimensiones y se utiliza comúnmente para representar posiciones y desplazamientos en un plano bidimensional. Aquí, se utiliza para representar las posiciones de los tubos.

java.util.Random;

Este import pertenece al paquete estándar de Java y se utiliza para importar la clase Random. La clase Random se utiliza para generar números aleatorios. En el contexto de la clase Tube, se utiliza para determinar las alturas aleatorias de los tubos superiores al ser creados.

Declaración de variables:

public static final int TUBE_WIDTH = 52: Define el ancho de los tubos, establecido como una constante.
private static final int FLUCTUATION = 130: Especifica la variación máxima en la posición vertical de los tubos.
private static final int TUBE_GAP = 100: Representa el espacio entre los tubos.
private static final int LOWEST_OPENING = 120: Indica la altura mínima de apertura para los tubos.
private Texture topTube, bottomTube: Texturas para la parte superior e inferior del tubo.
private Vector2 posTopTube, posBotTube: Posiciones de la parte superior e inferior del tubo.
private Rectangle boundsTop, boundsBot: Áreas rectangulares para las colisiones.
private Random rand: Generador de números aleatorios.

Métodos:

Tube(float x): Este es el constructor de la clase Tube. Se encarga de inicializar las texturas de los tubos, el generador de números aleatorios (Random), y configurar las posiciones y áreas rectangulares de los tubos. La posición del tubo superior se establece de manera aleatoria dentro de ciertos límites, y la posición del tubo inferior se ajusta en consecuencia.

getBottomTube() y getTopTube(): Estos métodos devuelven las texturas del tubo inferior y superior, respectivamente.

getPosTopTube() y getPosBotTube(): Devuelven las posiciones del tubo superior e inferior, respectivamente.

reposition(float x): Este método se utiliza para reposicionar los tubos cuando han pasado fuera de la pantalla. Se generan nuevas alturas aleatorias y se ajustan las posiciones y áreas rectangulares en consecuencia.

collides(Rectangle player): Verifica si el jugador colisiona con alguno de los tubos. Retorna true si hay colisión, false en caso contrario.

dispose(): Libera la memoria utilizada por las texturas de los tubos al finalizar su vida útil.

GameOverState:

```
package States;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.mygdx.game.FloppyDemo;

public class GameOverState extends State {
    private Texture gameOverTexture;
    private Texture bg;

    public GameOverState(GameStateManager gsm) {
        super(gsm);
        cam.setToOrtho(false, FloppyDemo.WIDTH / 2, FloppyDemo.HEIGHT / 2);
        gameOverTexture = new Texture("gameover.png");
        bg = new Texture("bg.png");
    }

    @Override
    protected void handleInput() {
        if (Gdx.input.justTouched()) {
            gsm.set(new MenuState(gsm));  // Regresar al menú al tocar la
pantalla
        }
    }

    @Override
    public void update(float dt) {
        handleInput();
    }

    @Override
    public void render(SpriteBatch sb) {
        sb.setProjectionMatrix(cam.combined);
        sb.begin();

        // Dibuja el fondo
        sb.draw(bg, cam.position.x - (cam.viewportWidth / 2), 0);
        sb.draw(gameOverTexture, cam.position.x -
gameOverTexture.getWidth() / 2, cam.position.y);  // Renderizado centrado
        sb.end();
    }

    @Override
    public void dispose() {
        gameOverTexture.dispose();
        bg.dispose();
        System.out.println("Game Over State Disposed");
    }
}
```

Importes:

com.badlogic.gdx.Gdx: Proporciona funcionalidades básicas y utilidades relacionadas con la plataforma, como entrada de usuario, archivos, gráficos, entre otros.

com.badlogic.gdx.graphics.Texture: Representa una textura bidimensional que puede ser utilizada para renderizar imágenes.

com.badlogic.gdx.graphics.g2d.SpriteBatch: Se utiliza para renderizar texturas y otras primitivas 2D en lotes, lo que mejora el rendimiento.

com.mygdx.game.FloppyDemo: Clase principal del juego que extiende ApplicationAdapter y actúa como punto de entrada. Contiene la configuración del juego y coordina los estados del juego a través de GameStateManager.

Declaración de variables:

private Texture gameOverTexture: Representa la textura utilizada para la pantalla de Game Over. Esta textura se cargará con la imagen que se mostrará cuando el jugador pierda.

private Texture bg: Representa la textura utilizada para el fondo del juego. Esta textura se carga con la imagen que será el fondo durante el juego.

Métodos:

public GameOverState(GameStateManager gsm): Constructor de la clase. Se encarga de inicializar la cámara, cargar las texturas para la pantalla de Game Over y el fondo.

handleInput(): Método para manejar la entrada del usuario. En este caso, verifica si se tocó la pantalla y, en ese caso, cambia al estado del menú.

update(float dt): Método de actualización. Actualmente, solo llama a handleInput() para manejar la entrada del usuario.

render(SpriteBatch sb): Método de renderizado. Configura la matriz de proyección de lotes de sprites, dibuja el fondo y la textura de Game Over centrada en la pantalla.

dispose(): Método para liberar recursos cuando el estado es descartado. Libera la memoria utilizada por las texturas y muestra un mensaje en la consola.

GameStateManager:

```
package States;

import com.badlogic.gdx.graphics.g2d.SpriteBatch;

import java.util.Stack;

public class GameStateManager {
    private Stack<State> states;  // Pila para gestionar los estados del
juego

    public GameStateManager() {
        states = new Stack<State>();  // Inicialización de la pila
    }

    public void push(State state) {
        states.push(state);  // Agregar un nuevo estado a la pila
    }

    public void pop() {
        states.pop();  // Eliminar el estado superior de la pila
    }

    public void set(State state) {
        states.pop();  // Eliminar el estado actual de la pila
        states.push(state);  // Agregar un nuevo estado a la pila
    }

    public void update(float dt) {
        states.peek().update(dt);  // Llamar al método de actualización del
estado actual en la cima de la pila
    }

    public void render(SpriteBatch sb) {
        states.peek().render(sb);  // Llamar al método de renderizado del
estado actual en la cima de la pila
    }
}
```

Importes:

com.badlogic.gdx.graphics.g2d.SpriteBatch: Importa la clase SpriteBatch del paquete com.badlogic.gdx.graphics.g2d. SpriteBatch es una clase de LibGDX que se utiliza para renderizar gráficos 2D.

java.util.Stack: Importa la clase Stack del paquete java.util. Stack es una estructura de datos de tipo pila que sigue el principio de Last In, First Out (LIFO). En este contexto, se utiliza para gestionar la pila de estados del juego.

Declaración de variables:

private Stack<State> states: Se declara una variable miembro llamada states. Esta variable es de tipo Stack, una estructura de datos de tipo pila. La pila se utiliza para gestionar los diferentes estados del juego, como el estado del menú, el estado de juego, etc. La clase State es la clase base para todos los estados del juego.

Métodos:

public GameStateManager(): Constructor de la clase GameStateManager. Inicializa la pila de estados (states) como una nueva instancia de Stack<State>.

public void push(State state): Método para agregar un nuevo estado a la pila. Recibe un objeto de tipo State como parámetro y lo agrega a la pila.

public void pop(): Método para eliminar el estado superior de la pila. Utiliza el método pop de la clase Stack para quitar el elemento en la cima de la pila.

public void set(State state): Método para cambiar el estado actual de la pila. Elimina el estado actual de la cima de la pila y agrega un nuevo estado proporcionado como parámetro.

public void update(float dt): Método para actualizar el estado actual en la cima de la pila. Llama al método update del estado actual.

public void render(SpriteBatch sb): Método para renderizar el estado actual en la cima de la pila. Llama al método render del estado actual, pasando un objeto SpriteBatch como parámetro para el renderizado.

MenuState:

```
package States;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.mygdx.game.FloppyDemo;

public class MenuState extends State {
    private Texture background;  // Textura del fondo del menú
    private Texture playBtn;  // Textura del botón de reproducción
    private Texture title;

    public MenuState(GameStateManager gsm) {
        super(gsm);
        cam.setToOrtho(false, FloppyDemo.WIDTH / 2, FloppyDemo.HEIGHT / 2);
// Configuración de la cámara
        background = new Texture("bg.png");  // Carga de la textura del
fondo
        playBtn = new Texture("playbtn.png");  // Carga de la textura del
botón de reproducción
        title = new Texture("title.png"); // Carga de la textura del titulo
    }

    @Override
    public void handleInput() {
        if (Gdx.input.justTouched()) {
            gsm.set(new PlayState(gsm));  // Cambio al estado de juego al
tocar la pantalla
        }
    }

    @Override
    public void update(float dt) {
        handleInput();  // Manejo de la entrada del usuario
    }

    @Override
    public void render(SpriteBatch sb) {
        sb.setProjectionMatrix(cam.combined);  // Configuración de la
matriz de proyección
        sb.begin();
        sb.draw(background, 0, 0);  // Renderizado del fondo
        sb.draw(playBtn, cam.position.x - playBtn.getWidth() / 2,
cam.position.y);  // Renderizado del botón de reproducción centrado

        // Especifica un tamaño personalizado para el título
        float titleWidth = 200;  // Ancho deseado del título
        float titleHeight = 100;  // Altura deseada del título
        sb.draw(title, cam.position.x - titleWidth / 2, cam.position.y +
playBtn.getHeight() , titleWidth, titleHeight);

        sb.end();
    }

    @Override
    public void dispose() {
        background.dispose();  // Liberación de la memoria utilizada por la
textura del fondo
        playBtn.dispose();  // Liberación de la memoria utilizada por la
```

```
textura del botón de reproducción
        title.dispose();  // Liberación de la memoria utilizada por la
textura del titulo
        System.out.println("Menu State Disposed");
    }
}
```

Importes:

com.badlogic.gdx.Gdx: Proporciona funciones relacionadas con la entrada, gráficos, archivos, etc., en el framework LibGDX.

com.badlogic.gdx.graphics.Texture: Representa una textura 2D que puede ser utilizada para renderizar imágenes.

com.badlogic.gdx.graphics.g2d.SpriteBatch: Permite renderizar una serie de imágenes 2D usando la GPU de manera eficiente.

com.mygdx.game.FloppyDemo: Referencia a la clase principal del juego FloppyDemo. Se utiliza para acceder a las constantes como WIDTH y HEIGHT definidas en esa clase.

Declaración de variables:

background: Textura del fondo del menú.

playBtn: Textura del botón de reproducción.

title: Textura del título del juego.

Métodos:

public MenuState(GameStateManager gsm): Este es el constructor de la clase MenuState. En este método, se inicializa la cámara para que sea ortográfica (2D) con dimensiones basadas en la mitad del ancho y alto de la pantalla del juego. Además, se cargan las texturas para el fondo, el botón de reproducción y el título del menú.

public void handleInput(): Este método se encarga de manejar la entrada del usuario. En este caso, verifica si se ha tocado la pantalla (Gdx.input.justTouched()) y, en caso afirmativo, cambia al estado de juego creando una nueva instancia de PlayState y configurándola en el GameStateManager.

public void update(float dt): El método update se llama en cada ciclo de actualización del juego. Aquí, simplemente se llama al método handleInput() para manejar la entrada del usuario.

public void render(SpriteBatch sb): Este método se encarga de renderizar los elementos en la pantalla. Se configura la matriz de proyección para la cámara y se inicia el lote de sprites (sb.begin()). Luego, se dibuja el fondo, el botón de reproducción y el título. El título se dibuja con un tamaño personalizado utilizando las variables titleWidth y titleHeight. Finalmente, se termina el lote de sprites (sb.end()).

public void dispose(): Este método se llama cuando se cierra el estado del menú y se encarga de liberar los recursos utilizados por las texturas del fondo, el botón de reproducción y el título. Además, imprime un mensaje en la consola para indicar que el estado del menú ha sido liberado.

PlayState:

```
package States;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.Preferences;
import com.badlogic.gdx.graphics.Color;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.BitmapFont;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.math.Vector2;
import com.mygdx.game.FloppyDemo;
import com.badlogic.gdx.utils.Array;

import sprites.Bird;
import sprites.Tube;

public class PlayState extends State {
    private static final int TUBE_SPACING = 125;
    private static final int TUBE_COUNT = 4;
    private static final int GROUND_Y_OFFSET = -30;

    private static final int SCORE_FACTOR = 10;
    private BitmapFont font;

    private float distanceTraveled = 0;
    private int score = 0;

    private int topScore;  // Variable para almacenar el top score
    private Preferences preferences;  // Objeto para acceder a las
preferencias de almacenamiento

    private Bird bird;
    private Texture bg;
    private Texture ground;
    private Vector2 groundPos1, groundPos2;

    private Array<Tube> tubes;

    public PlayState(GameStateManager gsm) {
        super(gsm);
        bird = new Bird(50, 300);
        cam.setToOrtho(false, FloppyDemo.WIDTH / 2, FloppyDemo.HEIGHT / 2);
        bg = new Texture("bg.png");
        ground = new Texture("ground.png");
        groundPos1 = new Vector2(cam.position.x - cam.viewportWidth / 2,
GROUND_Y_OFFSET);
        groundPos2 = new Vector2((cam.position.x - cam.viewportWidth / 2) +
ground.getWidth(), GROUND_Y_OFFSET);

        tubes = new Array<Tube>();

        for (int i = 1; i <= TUBE_COUNT; i++) {
            tubes.add(new Tube(i * (TUBE_SPACING + Tube.TUBE_WIDTH)));
        }

        font = new BitmapFont();
        font.setColor(Color.WHITE);
        font.getData().setScale(2);

        // Inicializa el objeto Preferences para acceder al almacenamiento
```

```
persistente
        preferences = Gdx.app.getPreferences("FlappyBirdPreferences");

        // Recupera el top score almacenado, si no existe, establece el top
score en 0
        topScore = preferences.getInteger("topScore", 0);
    }

    @Override
    protected void handleInput() {
        if (Gdx.input.justTouched())
            bird.jump();
    }

    @Override
    public void update(float dt) {
        handleInput();
        updateGround();
        bird.update(dt);
        cam.position.x = bird.getPosition().x + 80;

        for (int i = 0; i < tubes.size; i++) {
            Tube tube = tubes.get(i);

            if (cam.position.x - (cam.viewportWidth / 2) >
tube.getPosTopTube().x + tube.getTopTube().getWidth()) {
                tube.reposition(tube.getPosTopTube().x + ((Tube.TUBE_WIDTH
+ TUBE_SPACING) * TUBE_COUNT));
            }

            if (tube.collides(bird.getBounds())) {
                // Cambiar al estado de Game Over si hay una colisión
                gsm.set(new GameOverState(gsm));
            }
        }

        if (bird.getPosition().y <= ground.getHeight() + GROUND_Y_OFFSET) {
            // Cambiar al estado de Game Over si el pájaro toca el suelo
            gsm.set(new GameOverState(gsm));
        }

        cam.update();
        cam.position.x = bird.getPosition().x + 80;

        distanceTraveled = bird.getPosition().x;
        score = (int) (distanceTraveled / SCORE_FACTOR);

        if (score > topScore) {
            topScore = score;
            preferences.putInteger("topScore", topScore);
            preferences.flush();
        }
    }

    @Override
    public void render(SpriteBatch sb) {
        sb.setProjectionMatrix(cam.combined);
        sb.begin();
        sb.draw(bg, cam.position.x - (cam.viewportWidth / 2), 0);
        sb.draw(bird.getTexture(), bird.getPosition().x,
bird.getPosition().y);
```

```
        for (Tube tube : tubes) {
            sb.draw(tube.getTopTube(), tube.getPosTopTube().x,
tube.getPosTopTube().y);
            sb.draw(tube.getBottomTube(), tube.getPosBotTube().x,
tube.getPosBotTube().y);
        }

        sb.draw(ground, groundPos1.x, groundPos1.y);
        sb.draw(ground, groundPos2.x, groundPos2.y);

        font.draw(sb, "Puntos: " + score, cam.position.x -
cam.viewportWidth / 2 + 20, cam.position.y + cam.viewportHeight / 2 - 20);
        font.draw(sb, "Top Score: " + topScore, cam.position.x -
cam.viewportWidth / 2 + 20, cam.position.y + cam.viewportHeight / 2 - 60);

        sb.end();
    }

    @Override
    public void dispose() {
        bg.dispose();
        bird.dispose();
        ground.dispose();
        for (Tube tube : tubes)
            tube.dispose();
        font.dispose();
        System.out.println("Play State Disposed");
    }

    private void updateGround() {
        if (cam.position.x - (cam.viewportWidth / 2) > groundPos1.x +
ground.getWidth())
            groundPos1.add(ground.getWidth() * 2, 0);
        if (cam.position.x - (cam.viewportWidth / 2) > groundPos2.x +
ground.getWidth())
            groundPos2.add(ground.getWidth() * 2, 0);
    }
}
```

Importes:

Gdx: Es una clase principal de LibGDX y proporciona funciones relacionadas con la entrada, gráficos, audio, archivos, etc.

Texture: Representa una imagen que se puede cargar en la GPU y utilizar para gráficos.

SpriteBatch: Permite renderizar una secuencia de sprites en una sola llamada, lo que mejora el rendimiento.

Vector2: Representa un vector bidimensional y es utilizado para representar posiciones y direcciones en el juego.

FloppyDemo: La clase principal del juego, que extiende ApplicationAdapter y proporciona los métodos principales de creación, renderizado y actualización.

Array: Una clase de contenedor dinámico proporcionada por LibGDX que se utiliza para almacenar objetos de manera eficiente.

Bird: La clase que modela al pájaro en el juego, con métodos y atributos para gestionar su comportamiento.

Tube: La clase que modela los tubos en el juego, con métodos y atributos para gestionar su posición y colisiones.

GameStateManager: Gestiona los diferentes estados del juego y proporciona métodos para cambiar entre ellos.

MenuState: Representa el estado del menú principal del juego, con métodos para manejar la entrada del usuario, actualizar y renderizar.

Declaración de variables:

TUBE_SPACING: Espacio entre los tubos en el juego.

TUBE_COUNT: Número total de tubos en el juego.

GROUND_Y_OFFSET: Desplazamiento vertical del suelo en relación con la parte inferior de la pantalla.

SCORE_FACTOR: Factor utilizado para calcular la puntuación.

BitmapFont font: Objeto que almacena la fuente utilizada para mostrar la puntuación.

float distanceTraveled: Distancia total recorrida en el juego.

int score: Puntuación actual del jugador.

int topScore: Puntuación máxima alcanzada.

Preferences preferences: Objeto utilizado para acceder a las preferencias de almacenamiento, como el top score.

Bird bird: Instancia de la clase Bird que representa al pájaro en el juego.

Texture bg: Textura utilizada para el fondo del juego.

Texture ground: Textura utilizada para representar el suelo.

Vector2 groundPos1, groundPos2: Posiciones del suelo, ya que el suelo se desplaza horizontalmente para dar la ilusión de movimiento.

Array<Tube> tubes: Un contenedor dinámico para almacenar instancias de la clase Tube, que representan los tubos en el juego.

Métodos:

En la clase PlayState, el constructor se encarga de inicializar las diferentes variables necesarias para el estado de juego. Se crea una instancia del objeto Bird, se configura la cámara, se cargan las texturas del fondo y suelo, se generan los tubos y se inicializan las variables relacionadas con la puntuación y el top score almacenado en preferencias.

El método handleInput gestiona la entrada del jugador, detectando si se ha tocado la pantalla y realizando un salto del pájaro en consecuencia.

En el método update, se maneja la entrada del jugador, se actualiza la posición del suelo, se actualiza la posición del pájaro, y se maneja la lógica de generación y colisión de los tubos. Además, se verifica si el pájaro ha tocado el suelo o un tubo, lo que lleva a un cambio al estado de Game Over. La posición de la cámara se actualiza para seguir al pájaro, y se calcula la puntuación en función de la distancia recorrida. Si la puntuación supera el top score almacenado, se actualiza el top score en las preferencias.

El método render se encarga de dibujar los elementos del juego, incluyendo el fondo, el pájaro, los tubos, el suelo y la puntuación. La posición de la cámara determina la posición de estos elementos en la pantalla.

Finalmente, el método dispose se encarga de liberar los recursos utilizados, como las texturas y la fuente, al cambiar de estado o cerrar la aplicación. Además, se imprime un mensaje en la consola para indicar que el estado de juego ha sido liberado.

PreferencesManager:

```java
package States;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.Preferences;

public class PreferencesManager {
    private static final String PREFERENCES_NAME = "FlappyBirdPreferences";
    private static final String TOP_SCORE_KEY = "topScore";

    private Preferences preferences;

    public PreferencesManager() {
        preferences = Gdx.app.getPreferences(PREFERENCES_NAME);
    }

    public int getTopScore() {
        return preferences.getInteger(TOP_SCORE_KEY, 0);
    }

    public void setTopScore(int topScore) {
        preferences.putInteger(TOP_SCORE_KEY, topScore);
        preferences.flush(); // Guarda los cambios inmediatamente
    }
}
```

Importes:

com.badlogic.gdx.Gdx: Proporciona acceso a funciones del entorno de ejecución de la aplicación, como entrada del usuario y acceso a archivos.

com.badlogic.gdx.Preferences: Utilizado para manejar preferencias de la aplicación, como almacenar y recuperar datos persistentes, por ejemplo, puntuaciones máximas.

Declaración de variables:

private static final String PREFERENCES_NAME = "FlappyBirdPreferences";: Establece el nombre de las preferencias, que actúa como un identificador único. En este caso, se llama "FlappyBirdPreferences".

private static final String TOP_SCORE_KEY = "topScore";: Define la clave para acceder al valor del puntaje máximo en las preferencias. En este caso, la clave es "topScore".

private Preferences preferences;: Crea un objeto de la clase Preferences para interactuar con el sistema de preferencias y acceder a los datos almacenados.

Métodos:

public PreferencesManager() { ... }: Constructor de la clase PreferencesManager. Se inicializa creando un objeto Preferences asociado al nombre "FlappyBirdPreferences". Este objeto es esencial para interactuar con el sistema de preferencias y almacenar/recuperar datos persistentes.

public int getTopScore() { ... }: Método que devuelve el puntaje máximo almacenado en las preferencias. Utiliza la clave "topScore" para acceder al valor almacenado. Si no hay un valor almacenado, devuelve 0 como valor predeterminado.

public void setTopScore(int topScore) { ... }: Método que establece el puntaje máximo en las preferencias. Toma como parámetro el nuevo puntaje máximo y lo guarda utilizando la clave "topScore". El método preferences.flush() se utiliza para guardar los cambios de manera inmediata.

State:

```
package States;

import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.math.Vector3;

public abstract class State {
    protected OrthographicCamera cam;  // Cámara para la vista del juego
    protected Vector3 mouse;  // Vector para almacenar las coordenadas del
puntero del mouse
    protected GameStateManager gsm;  // Administrador de estados del juego

    public State(GameStateManager gsm) {
        this.gsm = gsm;
        cam = new OrthographicCamera();  // Inicialización de la cámara
        mouse = new Vector3();  // Inicialización del vector del mouse
    }

    protected abstract void handleInput();  // Método para manejar la
entrada del usuario
    public abstract void update(float dt);  // Método para actualizar la
lógica del estado
    public abstract void render(SpriteBatch sb);  // Método para renderizar
los elementos del estado
    public abstract void dispose();  // Método para liberar los recursos y
la memoria utilizados por el estado
}
```

Importes:

com.badlogic.gdx.graphics.OrthographicCamera: Importa la clase OrthographicCamera del paquete com.badlogic.gdx.graphics. Esta clase representa una cámara ortográfica que proyecta en 2D y se utiliza comúnmente en juegos 2D.

com.badlogic.gdx.graphics.g2d.SpriteBatch: Importa la clase SpriteBatch del paquete com.badlogic.gdx.graphics.g2d. SpriteBatch es una clase que se utiliza para renderizar gráficos 2D, especialmente útil para renderizar texturas y sprites.

com.badlogic.gdx.math.Vector3: Importa la clase Vector3 del paquete com.badlogic.gdx.math. Vector3 es una clase que representa un vector tridimensional y se utiliza para representar puntos en el espacio tridimensional. En este contexto, se puede utilizar para manejar la posición de la cámara en un juego 2D.

Declaración de variables:

En Java, el modificador protected se utiliza para especificar que un miembro de una clase (ya sea un campo o un método) es accesible dentro del mismo paquete y también por las subclases, incluso si estas están en paquetes diferentes. Veamos por qué se ha utilizado protected en los campos que mencionas:

protected OrthographicCamera cam;: La cámara (OrthographicCamera) se declara como protected porque es probable que otras clases que hereden de esta también necesiten acceder a la cámara para realizar operaciones específicas de la vista del juego. Al hacerlo protected, las subclases pueden acceder a esta cámara.

protected Vector3 mouse;: El vector mouse se declara como protected para permitir que las subclases accedan a él si necesitan realizar operaciones relacionadas con las coordenadas del puntero del mouse. Esto podría ser útil en juegos interactivos donde se requiere interacción del usuario.

protected GameStateManager gsm;: El GameStateManager se declara como protected para permitir que las subclases tengan acceso al administrador de estados del juego. Esto es útil porque las subclases pueden necesitar cambiar o manipular estados del juego.

Métodos:

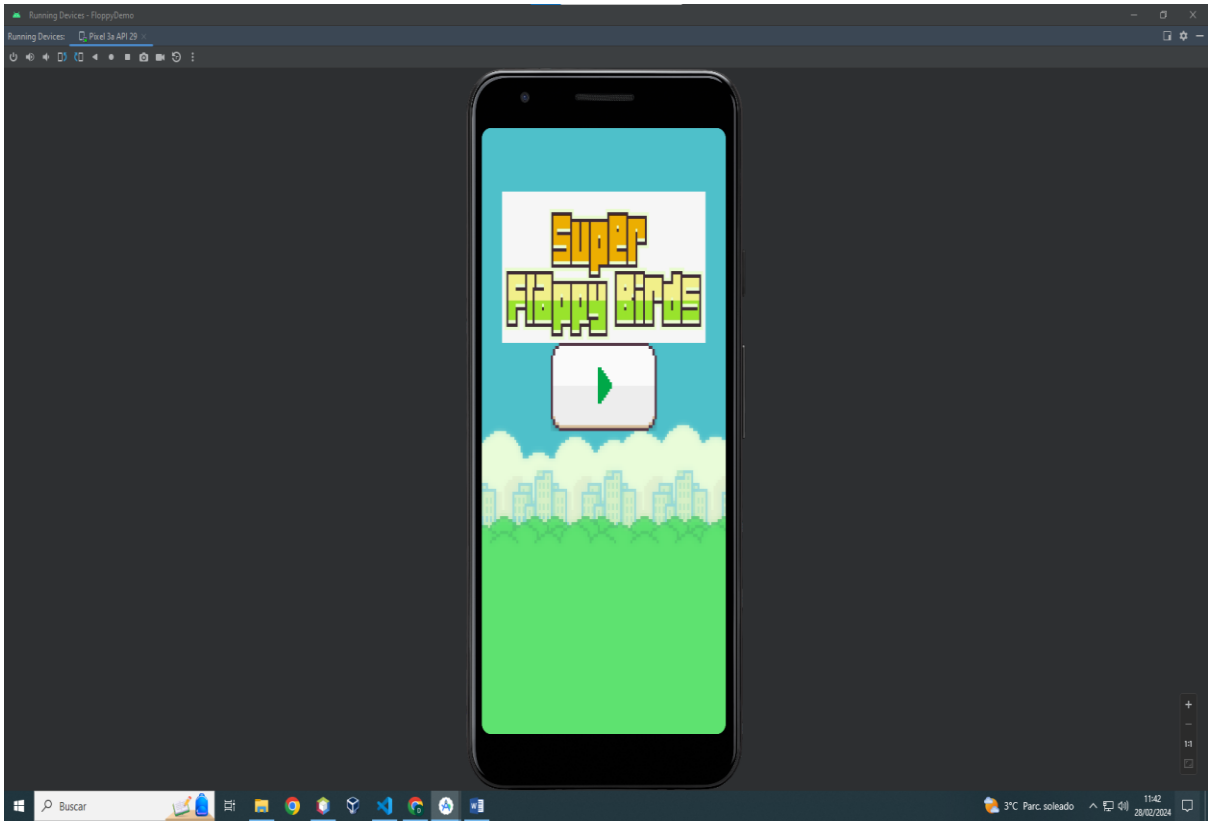
Constructor (public State(GameStateManager gsm)): El constructor de la clase toma un parámetro GameStateManager (gsm) que representa el administrador de estados del juego. La cámara (cam) y el vector del mouse (mouse) se inicializan en este constructor.

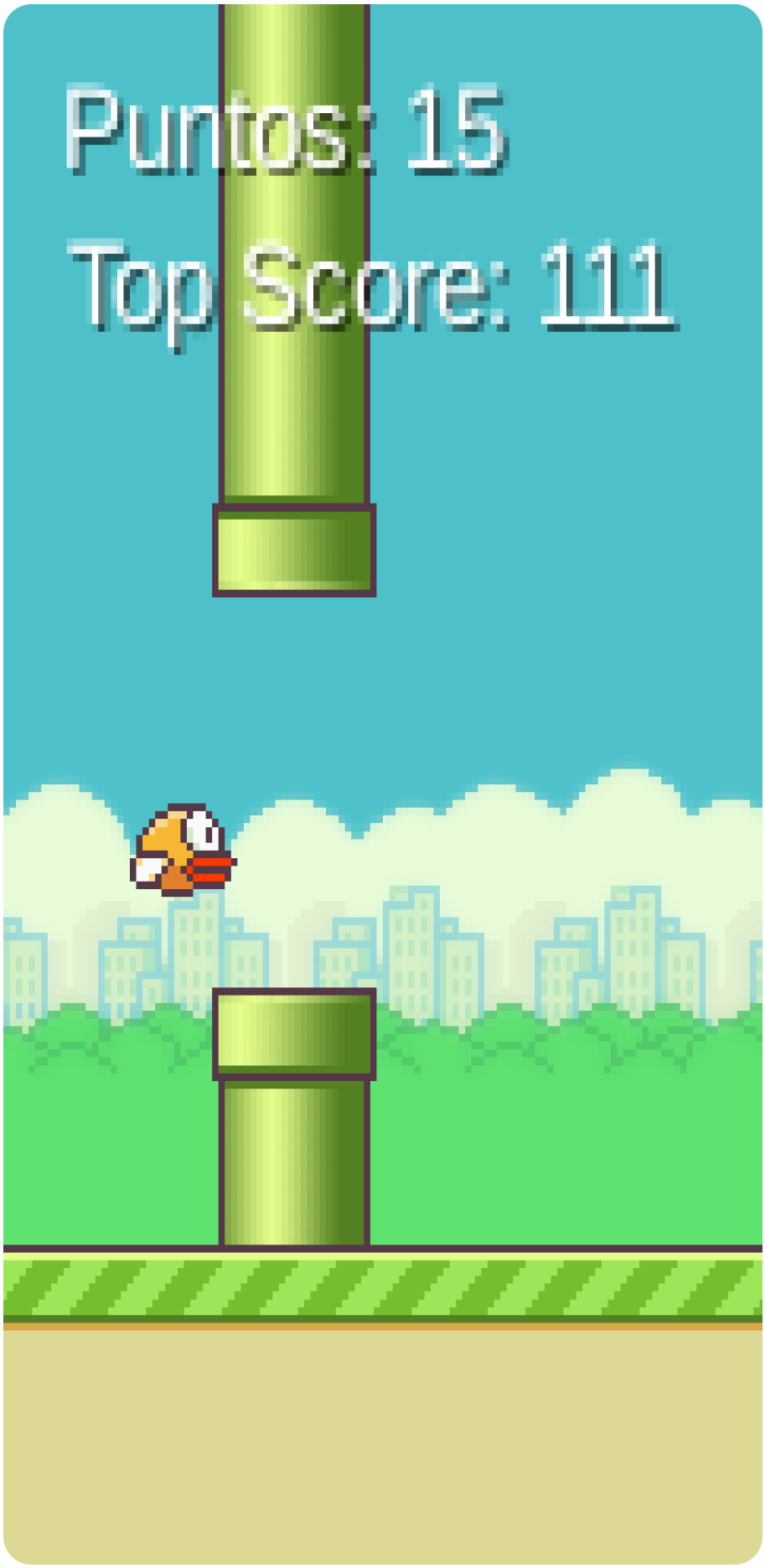
Campos Protegidos (protected OrthographicCamera cam; y protected Vector3 mouse;): Estos campos son protected, lo que significa que están accesibles para las subclases. La cámara (OrthographicCamera) se utiliza para manejar la vista del juego, y el vector mouse almacena las coordenadas del puntero del mouse.

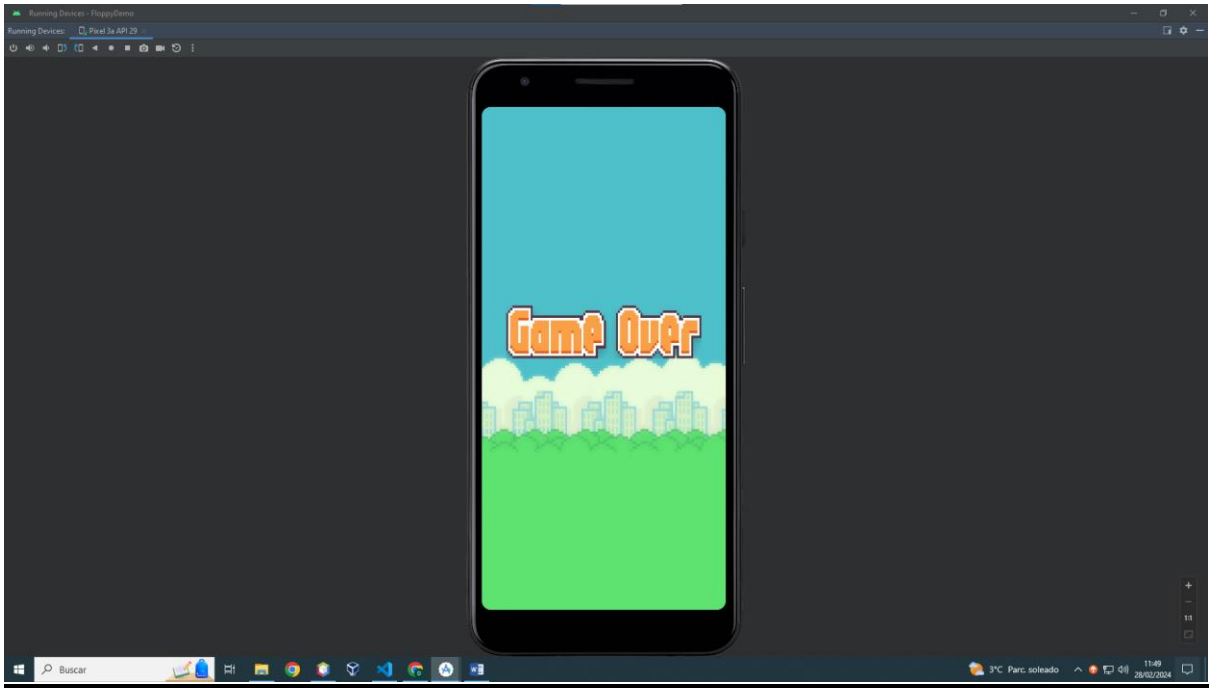
Métodos Abstractos (protected abstract void handleInput();, public abstract void update(float dt);, public abstract void render(SpriteBatch sb);, public abstract void dispose();): Estos métodos son abstractos y deben ser implementados por las subclases de State. Cada método tiene un propósito específico:

handleInput(): Maneja la entrada del usuario.
update(float dt): Actualiza la lógica del estado, donde dt representa el tiempo transcurrido.
render(SpriteBatch sb): Renderiza los elementos visuales del estado utilizando un SpriteBatch.
dispose(): Libera los recursos y la memoria utilizados por el estado cuando ya no es necesario.

Vista del proyecto:


Running Devices - FloppyDemo
Running Devices:
Pixel 3a API 29
Super
Flappy Birds
Buscar
3°C Parc. soleado
11:42
28/02/2024


Puntos: 15
Top Score: 111


Running Devices - FloppyDemo
Running Devices:
Pixel 3a API 29
Game Over
Buscar
3°C Parc. soleado
11:49
28/02/2024

CONCLUSIÓN

Antes de decidirnos por el proyecto, evaluamos diversas opciones y observamos que Flappy Bird se presentaba como una elección apropiada. Dada su antigüedad y relativa simplicidad, encontramos numerosas referencias que podríamos aprovechar en caso de tener dudas o enfrentarnos a desafíos. La disponibilidad de texturas del juego en formato PNG y con una calidad excepcional también influyó en nuestra decisión, ya que nos permitiría ahorrar tiempo en la creación de gráficos.

En última instancia, optamos por mantener el juego lo más fiel posible al original. Inicialmente, consideramos la posibilidad de agregar las imágenes de los profesores y permitir que los usuarios seleccionaran a su personaje en la pantalla de MenuState. Sin embargo, abandonamos esta idea por dos razones fundamentales. En primer lugar, la implementación de una animación similar a la que utilizamos, donde una imagen PNG se divide en tres y se muestra según el tiempo transcurrido, se reveló como un desafío considerable. Además, la dificultad para obtener fotografías ideales de los profesores para realizar dicha animación complicaba aún más el proyecto.

Esta decisión nos permitió centrarnos en aspectos cruciales del desarrollo, como la mecánica del juego y la eficiencia del código, evitando complicaciones innecesarias y asegurando un enfoque más pragmático en el desarrollo del Flappy Bird que presentamos en la actualidad.

Alguna implementación de mejoras específicas que podría enriquecer aún más la experiencia de juego de Flappy Bird incluyen la introducción de elementos interactivos como power-ups y obstáculos móviles con el aumento de dificultad según sube la puntuación, la incorporación de personajes desbloqueables con habilidades únicas, la posibilidad de personalizar la apariencia del personaje como hemos mencionado anteriormente, y la expansión de los entornos gráficos, es decir, nuevos mapas.

Si queremos darle un enfoque un poco más empresarial al juego habría que buscar algún tipo de forma de obtención económica basada en por ejemplo mundos, o skins únicas basado en modelos actuales en grandes videojuegos como “call of duty” o “Apex Legends” donde pagas por lotes aleatorios de este tipo de items. Incluir las redes sociales podría ser clave para asegurar mantener a la gente jugando con el pase del tiempo a un juego relativamente “simple y fácil” mediante retos entre jugadores o amigos.

Además, la adición de desafíos diarios o semanales y la implementación de un sistema de logros podrían mantener a los jugadores comprometidos y proporcionar una mayor sensación de progresión. Estas mejoras, combinadas con la base sólida de Flappy Bird, podrían ofrecer una experiencia de juego más completa y atractiva para los usuarios.

BIBLIOGRAFÍA

https://libgdx.com/wiki/
https://libgdx.com/dev/
https://docs.oracle.com/en/java/
https://www.java.com/en/
https://libgdx.com/wiki/graphics/2d/2d-animation
https://libgdx.com/wiki/graphics/2d/spritebatch-textureregions-and-sprites
https://libgdx.com/wiki/graphics/2d/orthographic-camera
https://es.stackoverflow.com/