Mapusky

TRV: Tecnología de videojuegos

Memoria

Jorge González Rodrigo (Jefe de equipo)

Guillermo Almagro Alonso

Ana Mª de la Fuente Aguilar

Álvaro Sáez contreras

Francisco Sánchez Jabonero

Pedro Barquín Ayuso

Mapusky, el videojuego

1. Introducción

Ésta es la memoria del código de nuestro videojuego. En ella se realiza un análisis del código, diferenciando los distintos paquetes utilizados para implementar el juego. Dentro del análisis de los paquetes, se documentan las clases que los forman, así como los métodos implementados en cada una de ellas.

2. Análisis del código

En este apartado se va a realizar un análisis del código que hemos utilizado para el desarrollo del nuestro videojuego. Hemos separado las funcionalidades mediante el uso de paquetes para que resulte más fácil seguir todo el código y otorgarle cierta modularidad.

Nuestro proyecto consta de nueve paquetes:

- game.state
- game.level
- game.character
- game.controller
- game.enums
- game.level.tile
- game.physics
- game
- sound

A continuación entraremos a explicar cada clase contenida en cada paquete.

2.1 game.state:

En este contenedor se encuentra la clase LevelState.java. Esta clase es la encargada de inicializar el juego en el contenedor de juego del tipo GameContainer. El contenedor del juego seleccionado es un tipo definido en las librerías de Slick2D.

La clase implementa el metodo init(Gamecontainer,StateBasedGame), que se encarga de inicializar el juego con el nivel que recibe mediante su constructor, inicializando un nuevo player e introduciéndolo en el nivel.

El método update(GameContainer, StateBasedGame, int), se encarga de la actualización de los datos.

Se implementa el método render(GameContainer, StateBasedGame, Graphics), que se encarga de llamar a OpenGL que renderiza el nivel con la escala proporcionada.

Por último se implementa un método de escape que se activa al pulsar la tecla escape y que desencadena la salida del juego. El método se denomina keyPressed(int, char)

2.2. game.enum

Se crea una enumeración para definir los movimientos del personaje.

```
package game.enums;

public enum Facing
{
    LEFT, RIGHT, STAND
}
```

Definiendo de esta manera las direcciones permitidas hacia las que se mueve el personaje o si este por el contrario se encuentra parado

2.3. game.character.

En este paquete se implementa todo lo necesario para la creación del personaje que vamos a utilizar, y de todas sus características.
Todo este se implementa en dos clases, la clase abstracta "Character", y la clase "Player" que se extiende de la clase abstracta, heredando de esta manera todos sus métodos.

2.3.1. class Character:

Se trata de una clase abstracta, en la que se definen las características principales que debe tener un personaje básico. De esta forma se podrán crear todos los personajes que se deseen con unas características básicas, simplemente extendiendo de esta clase abstracta.
Los atributos de esta clase se declaran de forma protegida para que de esta manera solo sean accesibles a las clases pertenecientes al mismo paquete. A continuación definiremos los atributos de esta clase:

- `protected HashMap<Facing,Image> sprites:` Se define un HashMap que almacenara las direcciones que puede tomar el personaje. De tal forma que cada dirección lleva asociada una imagen del personaje.
- `protected HashMap<Facing,Animation> movingAnimations:` Igual que el atributo anterior almacenaba direcciones asociadas a imágenes, este HasMap almacena las animaciones de movimiento asociadas a las direcciones.
- `protected HashMap<Facing,Animation> standingAnimations:` Este atributo almacena la animación de movimiento de cuando el personaje se encuentra parado, y la asocia al movimiento que corresponde a cuando el personaje no se mueve.

- `protected Facing facing`: Atributo del tipo Facing q contienen las enumeraciones de las direcciones hacia donde se puede mover el personaje y de cuando este está parado.
- `protected boolean moving = false`: Booleano que servirá para devolver si el personaje se mueve o no.
- `protected float accelerationSpeed = 1`: Atributo que define la aceleración del personaje.
- `protected float decelerationSpeed = 1`: Atributo que define la deceleración del personaje.
- `protected float maximumSpeed = 1`: define la velocidad máxima del personaje.
- `protected long lastTimeMoved`: este atributo se utiliza para detectar si se está moviendo el personaje, y si es así cargar la animación.

A continuación se definen los métodos de la clase:

- `public Character(float x, float y)`: Método constructor, recibe la posición inicial donde aparecerá el personaje. En él se define que la posición inicial del personaje será parado y se llama al método setSprite para introducirle la imagen inicial correspondiente al personaje sin moverse.
- `protected void setStandingAnimation(Image[]images, int frameDuration)`: Método que inicializa la animación del personaje cuando este se encuentra parado.
- `protected void setMovingAnimation(Image[]images, int frameDuration)`: Método similar al anterior, salvo que este inicia la animación de movimiento, teniendo en cuenta la dirección del personaje.
- `protected void setSprite(Image i)` Mediante este método inicializa el HashMap sprites y le asocia las imágenes de las direcciones.
- `public boolean isMoving()`: Este método devuelve el atributo moving que se utiliza para determinar si el personaje se mueve o no.
- `public void setMoving(boolean b)`: Este método modifica el atributo moving con el parámetro que recibe.
- `public void decelerate(int delta)`: Se define el comportamiento que debe tener el personaje al decelerar.
- `public void jump()`: Este método se encarga de definir el salto del personaje.
- `public void moveRight(int delta)`: Método que define el movimiento del personaje hacia la derecha. Modifica la variable moving a true.
- `public void render(float offsetX, float offsetY)`: el método render es el encargado de dibujar el personaje. Se ha implementado de forma que si detecta que el personaje está parado

salta la animación de cuando está parado y si no saltan las animaciones de movimiento. Otorgando de esta manera mas vida al personaje.

Mediante estos métodos se crea toda la lógica de juego que debe tener un personaje, definiendo como debe moverse, a qué velocidad, como saltar. También se define como debe mostrarse por pantalla mediante las animaciones e imágenes.

2.3.2. class Player:

La clase Player extiende de la clase abstracta contenida en este mismo paquete llamada Character. De esta forma hereda todas la características básicas que debe tener un personaje. En esta clase definiremos nuestro personaje o jugador. A continuación se muestran los atributos de esta clase, que enriquecen a los de la clase abstracta de la que hereda:

- `Image [] movRight = {...}`: Este atributo es un array que contiene las imagenes de nuestro personaje, Mapusky, andando.
- `Image []stand = {...}`: Este atributo es un array que contiene las imagenes de nuestro personaje, Mapusky, cuando está parado.

Esta clase solo contiene dos métodos:

- `public Player(float x, float y)` : El constructor recibe la posición y llama a los métodos de la clase abstracta. Se pasa por parámetro los atributos de esta clase a los metodos setMovingAnimation y SetStandingAnimation. Se definen la velocidad de aceleración, la velocidad máxima, la velocidad de deceleración y la máxima velocidad de caída que tendrá nuestro personaje.
- `public void updateBoundingShape()`: Este método sirve para ajustar el delimitador y poder calcular bien las colisiones respecto al tamaño de nuestro personaje. en este caso se ajustan 5 pixeles a la derecha y 2 pixeles hacia arriba.

Esta clase define nuestro personaje y sus características especificas para nuestro juego.

2.4. game.controller.

En el paquete game.controller se almacenan la clases necesarias para definir los controles que tendrá nuestro juego. Consta de dos clases, una abstracta y otra con la lógica de control elegida.

2.4.1. class PlayerController:

Esta clase es una clase abstracta creada para ofrecer mayor portabilidad a nuestro juego, es decir mediante esta clase podremos definir nuevas clases con distintos tipos de controles. Esto sirve por si se quiere adaptar a los controles de un mando de consola, a un teléfono móvil.

Solo consta de un atributo y dos métodos:

- `protected Player player:` Mediante este atributo creamos un objeto del tipo jugador.
- `public PlayerController(Player player):` El método constructor recibe un objeto player y lo inicializa.
- `public abstract void handleInput(Input i, int delta):` Método abstracto, se trata de un manejador para definir los tipos de controles que queremos. Mediante este método podremos generar cualquier control como decíamos antes, ofreciendo de esta manera la posibilidad de portar este juego a plataformas móviles como pueden ser teléfonos, tabletas, etc.

2.4.2. class MouseAndKeyBoardPlayerController:

La clase MouseAndKeyBoardPlayerController extiende de la clase abstracta PlayerControler. De esta forma hereda todos sus métodos de la clase abstracta, pero pudiendo redefinir el método abstracto handleInput.

Esta clase está formada por tres métodos:

- `public MouseAndKeyBoardPlayerController(Player player):` Constructor de la clase.
- `private void handleKeyboardInput(Input i, int delta):` Este método define la lógica de controles que seguirá nuestro juego. Las teclas que permitirán mover al personaje son las siguientes: tecla de dirección derecha y tecla "D" para mover a Mapusky hacia la derecha, tecla de dirección izquierda y tecla "A" para mover a Mapusky hacia la izquierda y por último la teclas espacio para que Mapusky salte.
- `public void handleInput(Input i, int delta):` Método heredado de la clase abstracta. Se redefine este método introduciendo una llamada al método anterior que contiene nuestra lógica de juego.

2.4.3. class XBoxController:

Esta clase extiende también de la clase abstracta PlayerControler. En este caso se implementa la utilización de un mando de XBOX para poder jugar al juego.

Se implementan los siguientes métodos:

- `XBoxController(Player player)`: Constructor de la clase.
- `private void handleKeyboardInput(Input i, int delta)`: Este método define los controles básico en los botones del mando de XBOX.
- `public void handleInput(Input i, int delta)`: Método heredado de la clase abstracta. Se redefine este método introduciendo una llamada al método anterior que contiene nuestra lógica de juego.

2.5. game.physics.

Este paquete contiene las clases con las que se definirá las colisiones y como interactuara el personaje por los mapas. Se definirá la fuerza gravitatoria que se simulara cuando el personaje este en el aire.

2.5.1. class BoundigShape:

Se trata de una clase abstracta en la que se declaran lo métodos abstractos que heredaran las clases que extiendan de ella.

Se implementan los siguientes métodos:

- `public abstract boolean checkCollision(AABoundingRect box)`: Este método se utiliza para ver el tipo de colisión que se ha detectado y llamar así al método adecuado para hacer frente a la colisión.
- `public abstract void updatePosition(float newX, float newY)`: Método que actualiza la posición con las nuevas posiciones que se le pasan como parámetro.
- `public abstract void movePosition(float x, float y)`: Se encarga de comprobar las colisiones cercanas cuando se realiza un movimiento.
- `public abstract ArrayList<Tile> getTilesOccupying(Tile[][] tiles)`: Almacena los tiles que ocupa nuestro personaje para poder calcular cuando se producirá una colisión.
- `public abstract ArrayList<Tile> getGroundTiles(Tile[][] tiles)`: Almacena los tiles declarados como suelo.

2.5.2. class AABoundingRect:

En esta clase se implementa el rectángulo de colisiones, es decir la forma en que detectaremos si nuestro personaje ha encontrado una colisión. Este rectángulo por decirlo de manera sencilla contenderá nuestro personaje para así poder detectar cuando se va a chocar y no puede seguir.

Sus atributos son los siguientes:

- `public float x`: Posición en el eje X en la que está el rectángulo.
- `public float y`: Posición en el eje Y en la que está el rectángulo.
- `public float width`: Anchura del rectángulo.
- `public float height`: Altura del rectángulo.

Se implementan los siguientes métodos:

- `public AABoundingRect(float x, float y, float width, float height)`: Constructor de la clase.
- `public void updatePosition(float newX, float newY)`: Actualiza la posición con los parámetros que recibe.
- `public void movePosition(float x, float y)`: Método que mueve el rectángulo sumando los valores x e y a los atributos de posición.
- `public boolean checkCollision(AABoundingRect rect)`: Este método comprueba los límites del rectángulo para comprobar si se a encontrado una colisión. en caso de que se cumpla devuelve true si no devuelve false.
- `public ArrayList<Tile> getTilesOccupying(Tile[][] tiles)`: Este método almacena las tiles que está ocupando nuestro personaje o mejor dicho el rectángulo de colisiones.
- `public ArrayList<Tile> getGroundTiles(Tile[][] tiles)`: Este método reconoce las tiles del suelo que se están ocupando y las almacena.

2.5.3. class Physics:

La clase Physics implementa lo necesario para que nuestro personaje se mueva correctamente con la gravedad asignada y detectando las colisiones. Se encarga de definir toda la física necesaria para el juego.

Tiene los siguientes atributos:

- `private final float gravity = 0.0015f`: Define la gravedad que tendra nuestro juego.
- `private static Sound mcomer`: En el se carga el sonido que hará Mapusky al comer.
- `public boolean sueloCierto=true`: Para detectar si es el suelo.

Se han creado los siguientes métodos:

- `public void phsysics()`: Este método carga la música que hace al comer nuestro personaje.
- `public void handlePhysics(Level level, int delta)`: Se encarga de toda la física dentro de un nivel.
- `private boolean checkCollision(LevelObject obj, Tile[][] mapTiles)`: Obtiene todas las tiles que ocupa el

personaje y comprueba las colisiones que pueda tener, devolviendo true si encuentra y false si no lo hace.

- `private boolean isOnGround(LevelObject obj, Tile[][] mapTiles)`: Este método comprueba si de verdad el personaje esta sobre el suelo. Para ello se baja un poco la posición y se comprueba si choca o se superpone. si se da ese caso estaría en el suelo, por lo que se restauraría posición y se devolvería true, en caso contrario se restauraría la posición y devolvería false.
- `private void handleCharacters(Level level, int delta)`: Este metodo maneja toda la fisica correspondiente al personaje, detectando las colisiones y definiendo el objetivo del personaje, que en este caso es comer frutas, llevando un contador de las que come y sonando una musica cada vez que come.
- `private void handleGameObject(LevelObject obj, Level level, int delta)`: Este método implementa toda la física del nivel y como interactúa con el nuestro personaje.
- `private void handleLevelObjects(Level level, int delta)`: este manejador carga el manejador handleGameObject, por cada objeto que hay en el nivel.

2.6. game.level.tile.

En este paquete se define como se trataran las tiles diferenciando entre dos tipos, la tile solida o la tile de aire. De esta manera podremos cargar en la solida el suelo y las colisiones. El paquete se compone de tres clase que definen los tiles.

2.6.1. class Tile:

Esta clase define el comportamiento y propiedades básicas que debe tener un Tile. A continuación se muestran los atributos de la clase:

- `protected int x`: Posición en el eje X.
- `protected int y`: Posición en el eje Y.
- `protected BoundingShape boundingShape`: Atributo para añadir las colisiones.

Sus métodos son:

- `public Tile(int x, int y)`: Constructor de la clase.
- `public int getX()`: Devuelve la posición del eje X.
- `public int getY()`: Devuelve la posición del eje Y.
- `public BoundingShape getBoundingShape()`: Devuelve el atributo de colisiones.

2.6.2. class SolidTile:

Esta clase extiende de la clase Tile, heredando todos sus métodos. En esta clase se le asigna el rectángulo de colisiones con un tamaño especifico. Tiene un único método, y es el constructor:

- `public SolidTile(int x, int y)`: Define un nuevo rectángulo de colisiones con las características que necesitamos para nuestro juego.

2.6.3. class AirTile:

Esta clase extiende de la clase Tile heredando todos sus métodos. No implementa ningún método, solo sirve para crear un nuevo tipo de Tile llamado AirTile, para definir una tile sin colisione. Su constructor es el siguiente:

- `public AirTile(int x, int y)`: Constructor de la clase.

2.7. game.level:

En el paquete game.level se implementa todo lo necesario para la gestión de los niveles. Consta de dos clases.

2.7.1. class LevelObject:

Esta clase es la superclase de todos los objetos dinámicos de nuestro juego, incluyendo personajes y objetos. Se trata de una clase abstracta y consta de los siguientes atributos:

- `protected float x`: Coordenada del eje X.
- `protected float y`: Coordenada del eje Y.
- `protected BoundingShape boundingShape`: Atributo para definir las colisiones.
- `protected float xVelocity = 0`: Velocidad en el eje X.
- `protected float yVelocity = 0`: Velocidad en el eje Y.
- `protected float maximumFallSpeed = 1`: Máxima velocidad de caída.
- `protected boolean onGround = true`: Booleano que indica si esta sobre el suelo.

Se definen los siguientes métodos:

- `public LevelObject(float x, float y)`: Constructor de la clase. Se declara un rectangulo de colisiones de 70x70.
- `public void applyGravity(float gravity)`: Este método aplica la gravedad sobre los objetos, controlando la velocidad de caída, y solo permitiendo una velocidad máxima.
- `public float getYVelocity()`: Devuelve la velocidad en el eje Y.

- `public void setYVelocity(float f):` Cambia la velocidad el eje Y, por la que se le pasa por parámetro.
- `public float getXVelocity() x:` Devuelve la velocidad en el eje X.
- `public void setXVelocity(float f):` Cambia la velocidad el eje X, por la que se le pasa por parámetro.
- `public float getX():`Devuelve la coordenada X de la posición.
- `public float getY():` Devuelve la coordenada Y de la posición.
- `public void updateBoundingShape():` Se actualiza la posición del rectángulo de colisiones para poder detectar nuevas colisiones.
- `public void setX(float f):` Se cambia la posición de la coordenada X y se llama al método updateBoundingShape() para actualizar la detección de colisiones en la nueva posición.
- `public void setY(float f):` Se cambia la posición de la coordenada Y y se llama al método updateBoundingShape() para actualizar la detección de colisiones en la nueva posición.
- `public boolean isOnGround():` Devuelve el valor del atributo onGround
- `public void setOnGround(boolean b:` Cambia el valor del atributo onGround por el valor que se le pasa por parámetro.
- `public BoundingShape getBoundingShape():` Devuelve el atributo boundingShape.
- `public abstract void render(float offsetX, float offsetY):` Método abstracto que sirve para renderizar el nivel.

2.7.2. class Level

Esta clase es la encargada de gestionar e inicializar todos los objetos dinámicos que aparecerán en el nivel. Se encarga de cargar el mapa, gestionar los personajes, gestionar los objetos del nivel y llevar la cuenta de las frutas recogidas.

Consta de los siguientes atributos:

- `private TiledMap map:` mapa del nivel a cargar.
- `private ArrayList<Character> characters:` array de personajes que aparecerán en el nivel.
- `private ArrayList<LevelObject> levelObjects:` array de objetos para cargar en el nivel.
- `private Tile[][] tiles:` tiles utilizados.
- `private Player player:` define nuestro jugador.
- `private int itemCount:` conteo de objetos (frutas) del nivel.
- `private int posY;`

Se definen los siguientes métodos:

- `public Level(String level, Player player)`: Constructor de la clase. Se define el mapa, los arrays de personajes y objetos, se define y carga el jugador, se cargan los tiles, se inicializan los contadores `itemCount` y `posY` a 0 y se llama al método `cargaObjetos()` para cargar los objetos en el nivel.
- `public float getPosY()`: devuelve el valor de la variable `posY`.
- `private void cargaObjetos()`: método encargado de cargar los objetos (frutas) en el mapa. Para ello, primero selecciona la capa "`FruitLayer`" de las capas del mapa, que será la capa dónde cargará las distintas frutas. Se recorre todo el mapa mediante dos bucles, y con un switch se generan las frutas que se cargan en el ArrayList `LevelObject` mediante el método `addLevelObject()` creando objetos del tipo `Objective` y se aumenta en 1 el contador `itemCount` por cada una de ellas.
- `public void render()`: Método de renderizado, tanto del mapa como del personaje.
- `private void loadTileMap()`: este método lleva a cabo la carga de los tiles que forman el mapa en la capa "`ColisionLayer`". Se crea un array con todos los tiles que forman el mapa y se recorre entero cargando en cada posición el tile correspondiente, ya sea de aire, como de terreno.
- `public ArrayList<LevelObject> getLevelObjects()`: devuelve todos los objetos dentro del ArrayList `LevelObject`.
- `public void addLevelObject(LevelObject obj)`: añade objetos al ArrayList `LevelObject`.
- `public ArrayList<Character> getCharacters()`: devuelve todos los personajes dentro del ArrayList `Character`.
- `public void addCharacter(Character c)`: añade personajes al ArrayList `Character`.
- `public Tile[][] getTiles()`: devuelve el tile de la posición indicada.
- `public int getXOffset()`: calcula el offset en el eje X.
- `public int getYOffset()`: calcula el offset en el eje Y.
- `public void removeObject(LevelObject obj)`: elimina un objeto del ArrayList `LevelObject`.
- `public void removeObjects(ArrayList<LevelObject> objects)`: vacía por complete el ArrayList `LevelObject`.
- `public int getItemCount()`: devuelve el valor de `itemCount`.
- `public void resetItemCount()`: pone a cero el contador `itemCount`.

2.7.3. class Objective

Clase que extiende de la clase `Level` y en la cual se definen las distintas imágenes que formarán las animaciones de las frutas, indicando su ruta.

Mediante el constructor `public Objective(float x, float y, String fruit)` y dependiendo de la fruta escogida, se crea la animación correspondiente y se carga en la posición obtenida mediante `super(x,y)`. Con el método `public void render(float offsetX, float offsetY)` se renderizan las animaciones.

2.8. game.sound

Paquete que engloba las clases encargadas de la gestión de la música del juego. Está formado por dos clases, iguales, pero en las cuales se carga una pista distinta.

2.8.1. class Musica y class Musica1

Estas dos clases están formadas por un único atributo `private Music music`. Poseen, además del constructor `public Musica()`, en el cual se indica la ruta de la pista que queremos reproducir, los siguientes métodos:

- `public boolean sondando()`: comprueba si se ha llamado al método `music.playing()` y devuelve un booleano en función de si la música está reproduciéndose o no.
- `public void repoducir()`: llamada al método `music.play()` que hace que comience la reproducción.
- `public void parada()`: método que detiene la reproducción.

2.9. game

Main class del proyecto. Nuestro videojuego es basado en estados, es decir, su funcionamiento radica en la creación y gestión (transiciones) de estados y por tanto extiende de la clase `StateBasedGame`.

En esta clase se definen los valores de:

- La resolución de pantalla:
```
public static final int     WINDOW_WIDTH  = 1366;
public static final int     WINDOW_HEIGTH = 768;
```

- La posibilidad de ejecutar el juego en pantalla completa:
```
public static final boolean FULLSCREEN = true;
```

- El tamaño de los tiles:
```
private static final int SIZE = 70;
```

- La escala (si es necesaria para ajustar):

```
public static final float  SCALE         = (float)
(0.6122448979591837*((double)WINDOW_WIDTH/840));
```

- El nombre del juego:
  ```
  public static final String GAME_NAME     = "Mapuski";
  ```

Se declaran también dos atributos, uno para llevar la cuenta de las frutas recogidas, y otro para conocer el nivel en el que estamos:

- `public static int FRUITS_COLLECTED;`
- `public static int ULTIMO_NIVEL;`

Para teminar encontramos los siguientes métodos:

- `public Game()`
- `public void initStatesList(GameContainer gc)`: crea los distintos estados, que se encargan de toda la lógica y el renderizado de cada uno de los niveles. Aquí es donde se cargan los niveles que serán jugables en el juego.
- `public static void main(String[] args)`: se crea un `AppGameContainer` el cual contendrá los estados de nuestro juego. Se definen los valores para el tamaño de visualización y si se ejecutará en pantalla completa o no. Se indica si queremos que se muestren los FPS por pantalla y se intenta crear un framerate de, en este caso, 60 frames por segundo. Para terminar, se inicializa el juego mediante `app.start();`