MEMORIA VIDEOJUEGO DANGERFLASK

Jorge Berbel Ruiz

Brianda Cabrera Riaza

Rafael Maderuelo Escolar

Sergio Sanz Sacristán

Javier

Índice de contenidos:

1. Introducción

Este proyecto el cual ha sido la construcción de un videojuego ha sido creado desde Netbeans, utilizando Slick2d, para la asignatura de Tecnología de videojuegos de la universidad de Alcalá.

Hemos disfrutado mucho creando algo parecido a esos juegos que disfrutábamos y disfrutamos actualmente, pero claro todo dentro de las posibilidades y el tiempo.

2. Equipo de desarrollo

El equipo encargado en el desarrollo del videojuego está compuesto por cuatro miembros. Cada miembro del equipo tendrá un rol determinado dentro del equipo:

- Sergio Sanz Sacristán: programación
- Brianda Cabrera Riaza: historia y jefa de proyecto
- Rafael Maderuelo Escolar: creación mapas
- Jorge Berbel Ruiz: sprites personajes e historia
- Javier

3. Ficha técnica

- Desarrollo: Grupo 3
- Género: Árcade, Laberinto/Puzzle.
- Número de jugadores: 1 Jugador.
- Plataforma: PC.
- Formato: Digital.
- Textos: Español.
- Voces: No.
- Online: No.
- Clasificación: PEGI 3.

4. Historia

Nos encontramos en el siglo XXIV. La humanidad se encuentra sobrecogida por una nueva y mortal enfermedad: la Fiebre Gorrina. Se ha llevado ya a decenas de miles de víctimas y está extendiéndose a una velocidad preocupante. Según las investigaciones, la Fiebre G brotó en Europa, tal vez en el sur. Muchos son los científicos que la intentan contener y en el mejor de los casos erradicarla, pero, por el momento, solo uno ha conseguido avances reales.

En un lugar sin nombre, escondido bajo tierra, se encuentra un enorme laboratorio, habitado por el exitoso científico y doctor Zacarías Lacura. Junto a él, un sobresaliente estudiante busca destacar en este mundo de la mano de un grande, el joven Julián Ponzoña. Muchos piensan que rozan la locura, otros piensan que la incondicional dedicación y sus avances son causa de sus mentes brillantes y su pasión por la ciencia, nada más lejos de la realidad. Sin embargo…

Leopoldo, hijo de Zacarías, sobrevive día a día a esta terrible enfermedad. También lo hace Brianda, madre de Julián.

Entre sus diversas investigaciones, observan las diferentes reacciones a las primeras curas de la enfermedad en otras formas de vida, llegando a usar animales, todos ellos acomodados en vitrinas, con viales donde se les aplican las diferentes soluciones. Al fin llegó el día en que parece que dieron con la cura, pero ente saltos de alegría y un pequeño despiste, a nuestros protagonistas les surgió un nuevo y último problema.

5. Objetivos del juego

El juego consta de 4 niveles en los cuales el jugador tendrá que superar los laberintos y obtener una serie de llaves ``escondidas´´ por el mapa sin perder las seis vidas que se tienen al empezar.

El nivel se visualizará por completo al jugador, por lo que no hay ningún tipo de scroll de la pantalla.

La puerta de salida se abrirá al encontrar las llaves necesarias, indicadas en el HUD.

6. Ventana inicial del juego

La ventana principal del juego consta de las siguientes opciones: (ver imagen adjunta)

- Jugar: Comenzar nueva partida
- Controles: Permite ver como se juega, las teclas.
- Opciones: Permite configurar las opciones del juego
- Cerrar(``X´´): Cierra la ventana del juego

7. HUD

En la parte superior de la pantalla de juego se visualiza un apartado en el que se aprecia claramente las vidas restantes y las llaves necesarias para pasar el nivel.

8. Enemigos

Babosas: velocidad reducida.
Movimiento lineal.

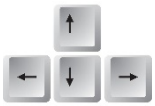
Monos: más rápidos que las babosas.
Movimiento lineal.

En nuestro juego hay dos tipos de enemigos, considerados por su velocidad de movimiento, el más ``conflictivo´´ seria el mono debido a su mayor velocidad con respecto a la babosa. El movimiento de los enemigos esta implementado en base a puntos del mapa, utilizando los sprites adecuados para cada dirección del movimiento. Para la detección de la colisión, se ha ``encapsulado´´ cada enemigo en un rectángulo al igual que los personajes, cuando los rectángulos se tocan (intersectan) la colisión es detectada y se produce la muerte del personaje.

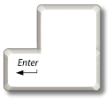
9. Controles

Los controles del juego son muy simples y son los siguientes:

- Flechas: moverse por los niveles del juego.



- Enter: para pasar los diálogos.



- Esc: volver atrás en el menú.

10. Análisis del código

El código de este juego está formado por 17 clases, vamos a comenzar por la clase mas importante, la clase Principal.

Principal:

Esta clase es una clase extensa de StateBasedGame. Es la que inicializará todos los estados del juego, el tamaño del juego y algunos atributos necesarios para la creación de los estados. Por ejemplo se inicializa la clase Vidas a 6, ya que esta clase contiene el numero de vidas del personaje del juego y tiene que ser una clase compartida entre los distintos estados para mantener el número de vidas. También nos encontramos la clase Sonido y la música del menú que la pasamos como atributo en el constructor de los estados Menú y Opciones para si en las opciones silenciamos la música esta música automáticamente deje de sonar.

Entonces, los estados inicializados, y por lo tanto, los estados del juego serán, Menu, Dialogo, Mapa1, Mapa2, Mapa3, Mapa4, DialogoFinal, GameOver, Controles y Opciones.

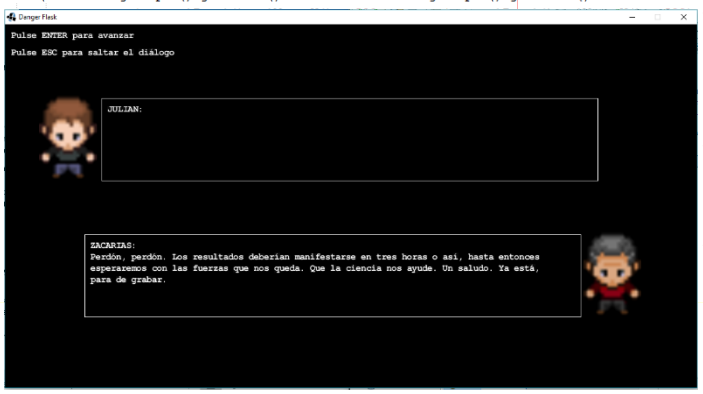
```
//Iniciamos los estados
@Override
public void initStatesList(GameContainer arg0) throws SlickException {
    this.addState(new Menu(sonido, menutheme));      //0
    this.addState(new Dialogo());                    //1
    this.addState(new Mapa1(vidas, sonido));         //2
    this.addState(new Mapa2(vidas, sonido));         //3
    this.addState(new Mapa3(vidas, sonido));         //4
    this.addState(new Mapa4(vidas, sonido));         //5
    this.addState(new DialogoFinal());               //6
    this.addState(new GameOver());                   //7
    this.addState(new Controles());                  //8
    this.addState(new Opciones(sonido, menutheme)); //9
}
```

Menu:

Esta clase es el estado 0 del juego y es de tipo BasicGameState. Este estado es el que contiene el menú del juego, el cual esta formado por una imagen de fondo y tres botones, jugar, controles y opciones, según se clickee a un botón u a otro accede a un estado u a otro.

En cuanto al código no hay mucho que contar, ya que, inicializamos las imágenes y la única "dificultad" es configurar que cuando haga click detecte a que botón ha dado y por lo tanto realizar la acción correspondiente. Además, si la música esta activada ponemos el loop la música y la paramos cuando demos al botón jugar.

```
//Detectamos cuando clickamos en el menú
if (container.getInput().isMouseButtonDown(0)) {
    //Si damos al botón JUGAR para la música pone el b a falso y entra en el estado 1
    if (container.getInput().getMouseX() < 388 && container.getInput().getMouseX() > 105
```



Dialogo y DialogoFinal

Estas clases son los estados 1 y 6. Son de tipo BasicGameState. En estas clases lo que hacemos es cargar las imágenes de los personajes y crear un rectángulo donde se va a encontrar el texto del dialog. Para que este dialogo avance creamos una variable de tipo int i, que se inicializa a 0 y que según pulsamos la tecla ENTER vamos aumentando en uno y dependiendo del valor de la variable i ponemos un texto u otro del dialogo.

Si pulsamos la tecla ESCAPE en el dialogo inicial nos saltamos el dialogo y avanzamos directamente al juego. Cuando termina el dialogo continua al estado que le corresponde.

Antes de continuar con el resto de los estados, vamos a explicar los métodos utilizados en las clases adicionales para así conocer el funcionamiento de estos cuando son utilizados en los estados.

LimitesMapa:

Esta clase consiste en crear una matriz de booleanos la cual esta creada a partir del TiledMap correspondiente. En esta clase tenemos cuatro métodos uno para cada mapa. Como los TiledMap están formados por capas la creación de esta matriz consiste en si en una parte del mapa, está la pared, en las coordenadas correspondientes en la matriz se ponga un true y un false cuando esta dentro de los limites del mapa. Así si las coordenadas del personaje están donde hay una pared en la matriz haya un true. (Para que esto fuese mas preciso tuvimos que realizar ciertas cuentas matemáticas para ajustarlo)

```
public boolean[][] crearLimite1(TiledMap mapa) {
    int totalTilesWidth = mapa.getWidth() * 2;
    int totalTilesHeight = mapa.getHeight() * 2;
    obstaculo = new boolean[totalTilesWidth][totalTilesHeight];
    for (int i = 0; i < totalTilesWidth; i++) {
        for (int j = 0; j < totalTilesHeight; j++) {
            obstaculo[i][j] = ((mapa.getTileId(i / 2, j / 2, mapa.getLayerIndex("Capa de patrones 4")) != 0)
                    || (mapa.getTileId(i / 2, j / 2, mapa.getLayerIndex("Capa de patrones 3")) != 0)
                    || (mapa.getTileId(i / 2, j / 2, mapa.getLayerIndex("Capa de patrones 2")) != 0));
        }
    }
    return obstaculo;
}
```

Colisiones:

Esta clase consiste en gestionar las colisiones del personaje con los límites del mapa. Y cuando poder atravesar las puertas y pasar al mapa siguiente.

Para ello creamos las coordenadas x_ e y_ que nos servirán para registrar las coordenadas del personaje y las variables int de las tarjetas de cada mapa, para asi saber cuando podemos atravesar las puertas.

También creamos rectángulos para gestionar las colisiones. Estos se encuentran en las puertas de salida de cada mapa y los otros en el limite del mapa para saber cuando cambiar al siguiente estado. Asi como un rectángulo que rodea al personaje para saber cuando colisiona con otros objetos.

```
private final Rectangle rectSalida1 = new Rectangle(1150, 495, 55, 30);
private final Rectangle rectSalida2 = new Rectangle(1167, 270, 50, 25);
private final Rectangle rectSalida3 = new Rectangle(335, 592, 50, 50);
private final Rectangle rectSalida4 = new Rectangle(478, 592, 50, 50);
private final Rectangle cambiarMapa1 = new Rectangle(1213, 499, 1, 45);
private final Rectangle cambiarMapa2 = new Rectangle(1213, 275, 1, 45);
private final Rectangle cambiarMapa3 = new Rectangle(336, 638, 45, 1);
private final Rectangle cambiarMapa4 = new Rectangle(480, 638, 45, 1);
private final Rectangle rectAnim = new Rectangle(x_ + 2, y_ + 2, 12, 23);
```

Tenemos los métodos animDentro, los cuales tenemos 4, uno para cada mapa. Este método consiste en detectar si el personaje colisiona con las paredes y si tiene todas las tarjetas le deja avanzar por las puertas gracias a los rectángulos

```
public boolean animDentro1(boolean[][] paredes, float x, float y) {
    if (((paredes[(int) ((x * 2) / 16)][(int) ((y * 2) / 16) + 3] == true)
            || (paredes[(int) ((x * 2) / 16) + 2][(int) ((y * 2) / 16) + 3] == true))
            && rectAnim.intersects(rectSalida1)
            && tarjeta1 == 2) {
        return false;
    } else if ((paredes[(int) ((x * 2) / 16)][(int) ((y * 2) / 16) + 3] == true)
            || (paredes[(int) ((x * 2) / 16) + 2][(int) ((y * 2) / 16) + 3] == true)) {
        return true;
    } else {
        return false;
    }
}
```

creados anteriormente. Este método devuelve un true cuando colisiona con alguna pared.

Luego esta el método actualizar, el cual actualiza las coordenadas del rectángulo rectAnim correspondiente al personaje, para que así el rectángulo siempre este rodeando al personaje.

```
public void actualizar(float x, float y) {
    rectAnim.setX(x + 2);
    rectAnim.setY(y + 2);
}
```

También tenemos los métodos cambiarMapa, también hay 4 uno para cada mapa. Estos métodos consisten en detectar cuando el personaje atraviesa la puerta para pasar al mapa siguiente. Cuando esto sucede, devuelve un true.

```
public boolean cambiarMapa1() {
    if (rectAnim.intersects(cambiarMapa1)) {
        return true;
    } else {
        return false;
    }
}
```

Personajes:

En esta clase vamos a gestionar la creación y movimiento de los personajes y sus animaciones, además de las colisiones entre estos.

En primer lugar iniciamos todos los sprites, animaciones y demás atributos necesarios. Por cada dirección de movimiento tanto del personaje como de los enemigos vamos a necesitar un Sprite y animación.

El constructor de la clase necesita tanto la clase Colisiones como varios atributos int, que corresponden a las coordenadas variables en su posición inicial de los enemigos.

Empezamos con los métodos necesarios para iniciar a los personajes y los enemigos, estos serán iniciarPersJulian(), iniciarPersZacarias(), iniciarEnemBab() e iniciarEnemMono(). Estos métodos inician los sprites y las animaciones correspondientes a cada personaje o enemigo. Ponemos un ejemplo.

```
//Método que inicia los sprites y las animaciones del personaje Julian
public void iniciarPersJulian() throws SlickException {
    spriteD = new SpriteSheet("./Personajes/spr_julian_derecha.png", 16, 26);
    animD = new Animation(spriteD, 100);
    spriteI = new SpriteSheet("./Personajes/spr_julian_izquierda.png", 16, 26);
    animI = new Animation(spriteI, 100);
    spriteAr = new SpriteSheet("./Personajes/spr_julian_arriba.png", 17, 27);
    animAr = new Animation(spriteAr, 100);
    spriteAb = new SpriteSheet("./Personajes/spr_julian_abajo.png", 17, 26);
    animAb = new Animation(spriteAb, 100);
}
```

Luego tenemos el método dibujarPers, el cual dependiendo la dirección a la que se esta moviendo el personaje dibuja una animación u otra. Esto lo consigue gracias a unos booleanos por cada dirección, los cuales solo puede estar a true uno de ellos. A este método hay que pasarlo los atributos x e y correspondientes al personaje

```
public void dibujarPers(float x, float y) {
    if (derecha) {
        animD.draw(x, y);
    }
    if (izquierda) {
        animI.draw(x, y);
    }
    if (arriba) {
        animAr.draw(x, y);
    }
    if (abajo) {
        animAb.draw(x, y);
    }
}
```

También tenemos el método dibujarEnem al cual hay que pasarle las coordenadas estaticas de los enemigos, ya que, estos solo se pueden mover en horizontal o vertical. Por cada enemigo tenemos un booleano, si este esta a true es que se mueve en la dirección inicial y pone la animación correspondiente, y si esta a false se mueve en la dirección contraria, entonces pone la animación correspondiente. Por ejemplo el enemigo 1.

```
public void dibujarEnem(int yl,
    if (enem1) {
        animEnemD.draw(a, yl);
    }
    if (!enem1) {
        animEnemI.draw(a, yl);
    }
```

Luego tenemos el método movimiento, cuya función es gestionar el movimiento del personaje. Este método devuelve el int i que tiene el valor correspondiente a la dirección a la que se mueve. También este método gestiona el movimiento del personaje según se pulse una tecla u otra, siempre que este dentro de los limites del mapa. Si pulsa la derecha pone todos los booleanos a false excepto el de la derecha, iniciamos la animación correspondiente a la dirección y movemos la x en este caso mientras este pulsado.

```
//Método que según pulses una dirección con los controles mueve el personaje en una dirección y otra
public int movimiento(boolean dentro, float x, float y, GameContainer container, int delta) {
    x_ = x;
    y_ = y;

    //Vamos a explicar este caso, si pulsa la flecha derecha y no ha colisionado con ninguna pared
    if (container.getInput().isKeyDown(Input.KEY_RIGHT) && dentro) {
        //pone las variables a false menos la direccion a la que va, en este caso, la derecha
        derecha = true;
        arriba = false;
        izquierda = false;
        abajo = false;
        //iniamos la animacion correspondiente a la direccion a la que va
        animD.start();
        //desplazamos el personaje en este caso hacia la derecha, aumentamos la x
        x_ += 100 * (float) delta / 1000;
        //Finalmente devolvemos la i que es un número que corresponde a una dirección, en este caso...
        //...la derecha, el 6
        i = 6;
        return i;
```

Cuando dejamos de mover el personaje paramos todas las animaciones y dejamos el Sprite en una determinada posición.

```
} else {
    //Esto significa que el personaje no se mueve entonces paramos las animaciones...
    //...y las dejamos congeladas en un sprite determinado
    animD.stop();
    animI.stop();
    animAr.stop();
    animAb.stop();
    animD.setCurrentFrame(1);
    animI.setCurrentFrame(1);
    animAr.setCurrentFrame(1);
    animAb.setCurrentFrame(1);
    return i;
}
```

```
public void movimientoEnem1(int delta) {
    //Por ejemplo aqui el enemigo 1 se mueve hacia la derecha hasta llegar...
    //...a la x=288, entonces se pone a false su variable enem1 y cambia de sentido...
    //...hasta la coordenada x=99. Y vuelve a comenzar el movimiento.
    if (enem1) {
        animEnemD.start();
        a += 10 * (float) delta / 1000;
        if (a > 288) {
            enem1 = false;
        }
    } else {
        animEnemI.start();
        a -= 10 * (float) delta / 1000;
        if (a < 96) {
            enem1 = true;
        }
    }
```

Para cada mapa tenemos un método distinto para el movimiento de los enemigos, se podría llamar inteligencia artificial a estos métodos movimientoEnem. Esto consiste en mover un enemigo en su dirección inicial hasta que llega a una cierta coordenada, donde entonces pasara a moverse en la dirección contraria, y cuando se llegue a la coordenada inicial vuelve a moverse en su dirección inicial. Dependiendo del mapa en el que nos encontremos los enemigos se moverán a una velocidad u a otra. Los monos son mas rapidos que las babosas.

```
public void colisionesBab(int y1, int y2, int y3, int x4, int x5, int x6) {
    rectEnem1 = new Rectangle(a + 2, y1 + 2, 21, 11);
    rectEnem2 = new Rectangle(b + 2, y2 + 2, 21, 11);
    rectEnem3 = new Rectangle(c + 2, y3 + 2, 21, 11);
    rectEnem4 = new Rectangle(x4 + 2, d + 2, 10, 19);
    rectEnem5 = new Rectangle(x5 + 2, e + 2, 10, 19);
    rectEnem6 = new Rectangle(x6 + 2, f + 2, 10, 19);
}
```

También en esta clase nos encontramos con los métodos colisionesBab y colisionesMono, que consisten en crear un rectángulo que rodee a los enemigos para asi poder gestionar las colisiones de los enemigos, se necesitan dos métodos, ya que, el tamaño del rectángulo de las babosas no es el mismo que el de los monos. A estos métodos les pasamos el valor de las coordenadas estáticas de los enemigos. También tenemos un método para actualizar las coordenadas de estos rectángulos para asi según se mueven los enemigos también se muevan los rectángulos.

Por ultimo, tenemos el método muere(), el cual devuelve un true cuando el personaje colisiona con alguno de los enemigos, es decir, el rectángulo del personaje intersecta con el rectángulo de algún enemigo.

```
//Este método devuelve true si el personaje colisiona con algún enemigo
public boolean muere() {
    if (col.getRectAnim().intersects(rectEnem1) || col.getRectAnim().intersects(rectEnem2)
            || col.getRectAnim().intersects(rectEnem3) || col.getRectAnim().intersects(rectEnem4)
            || col.getRectAnim().intersects(rectEnem5) || col.getRectAnim().intersects(rectEnem6)) {
        return true;
    } else {
        return false;
    }
}
```

Objetos:

Este método gestiona los objetos que nos encontramos en los mapas. En el constructor le pasamos las coordenadas de los objetos. En el método creaObjetos(). Inicia las imágenes botiquín, tarjeta1 y tarjeta2.

```
public void creaObjetos() throws SlickException {
    botiquin = new Image("./Objetos/spr_adrenalina.png");
    tarjeta1 = new Image("./Objetos/spr_tarjeta.png");
    tarjeta2 = new Image("./Objetos/spr_tarjeta2.png");
}
```

El método dibuja(), dibuja los objetos si las variables booleanas correspondientes a cada objeto están a true. Si están a false no las dibuja y además elimina el rectángulo que las rodea, para así evitar fallos de colisiones.

```
public void dibuja() throws SlickException {
    //Si los booleanos a, b y c están a true
    if (a) {
        botiquin.draw(xb, yb);
    }
    if (!a) {
        bot.setX(0);
        bot.setY(0);
    }
    if (b) {
        tarjeta1.draw(x1, y1);
        tarj1.setX(x1);
        tarj1.setY(y1);
    }
    if (!b) {
        tarj1.setX(0);
        tarj1.setY(0);
    }
```

```
public void colObj() {
    bot = new Rectangle(xb, yb, 10, 23);
    tarj1 = new Rectangle(x1, y1, 20, 20);
    tarj2 = new Rectangle(x2, y2, 20, 20);
}
```

El método colObj(), crea los rectángulos que rodean a los objetos para gestionar las colisiones.

Lugo, tenemos un método por cada colisión del personaje con el objeto, una para el botiquín, y otro para cada tarjeta. Este devuelve true si el personaje colisiona con alguno de los objetos, es decir, el rectángulo del personaje intersecta con el rectángulo de alguno de los objetos.

```
public boolean tar1Col() {
    if (col.getRectAnim().intersects(tarj1)) {
        return true;
    } else {
        return false;
    }
}
```

Sonido:

El objetivo de esta clase es gestionar el sonido del juego. Para empezar crea dos variables booleanas MusicaOn y SonidoOn, las cuales están a true y que se pondrán a false si se desactiva el sonido o la música.

El método iniciarSonidos(), inicia todos los efectos de sonido que se pueden dar a lo largo del juego.

```
public void iniciarSonidos() throws SlickException {
    puerta = new Sound("./Sonidos/abrir_puerta.ogg");
    tarjeta = new Sound("./Sonidos/coger_tarjeta.ogg");
    juliandead = new Sound("./Sonidos/julian_muerte.ogg");
    zacariasdead = new Sound("./Sonidos/zacarias_muerte.ogg");
    adrenalina = new Sound("./Sonidos/usar_adrenalina.ogg");
```

Y luego el método click() detecta cuando el usuario desea activar/desactivar la música y/o el sonido cuando se encuentra en los

```
public void click(GameContainer container, Music musica) {
    if (container.getInput().getMouseX() < 657 && container.getInput().getMouseX() > 625
            && container.getInput().getMouseY() < 32 && container.getInput().getMouseY() > 5) {
        if (musicaOn) {
            musica.stop();
            musicaOn = false;
        } else {
            musica.loop(1, 0.5f);
            musicaOn = true;
        }
    }
    if (container.getInput().getMouseX() < 892 && container.getInput().getMouseX() > 860
            && container.getInput().getMouseY() < 32 && container.getInput().getMouseY() > 5) {
        if (sonidoOn) {
            sonidoOn = false;
        } else {
            sonidoOn = true;
        }
    }
}
```

mapas. Este método detecta donde a clickado el usuario y si clicka donde la música pone la variable booleana de la música en el valor contrario al que estaba. Y lo mismo con el sonido.

Con los métodos get coge los sonidos de la clase cuando son necesarios.

Vidas:

Esta clase gestiona las vidas que tiene el personaje, y dibuja el numero de vidas que tiene en el juego. El constructor obtiene una variable int, que es el numero de vidas que tiene inicialmente. Tiene el método dibujar que dependiendo del numero de vidas que tenga dibuja el numero de vidas con corazones, y las vidas perdidas con corazones rotos.

```
public void dibujar(Graphics g) throws SlickException {
    //Inicia las vidas
    v1 = new Image("./Interfaz/spr_vida.png");
    v2 = new Image("./Interfaz/spr_vidamenos.png");

    if (vidas == 6) {
        v1.draw(260, 10);
        v1.draw(210, 10);
        v1.draw(160, 10);
        v1.draw(110, 10);
        v1.draw(60, 10);
        v1.draw(10, 10);
    } else if (vidas == 5) {
        v2.draw(260, 10);
        v1.draw(210, 10);
        v1.draw(160, 10);
        v1.draw(110, 10);
        v1.draw(60, 10);
        v1.draw(10, 10);
```

Mapa1, 2, 3 y 4:

Estas clases son los estados jugables, los estados 2, 3, 4 y 5. Son de tipo BasicGameState. Para empezar, inicializamos el mapa, el TiledMap y ponemos las coordenadas de inicio del personaje. También inicializamos todos los atributos necesarios para el desarrollo de la clase.

En el init iniciamos todo lo necesario para el estado. El mapa, el personaje y los enemigos, creamos la matriz con los limites del mapa, creamos los rectángulos de los enemigos para detectar las colisiones, creamos los objetos y sus respectivas colisiones, iniciamos los sonidos y la configuración de las imágenes. (En la mayoría de los casos utilizamos métodos de las clases explicadas anteriormente).

En el render, renderizamos el TiledMap, el cual hay que reducir su tamaño a la mitad para que entre dentro de las dimensiones del juego. Renderizamos los personajes y los enemigos, las vidas, los objetos, el texto y las imágenes del sonido. Las imágenes del sonido te dicen si esta activado o no, y puedes modificarlo haciendo click en ellas.

En el update, primero tenemos una variable booleana b que esta a false, entonces cuando hace el primer update del estado y la música esta activada empieza a sonar la música y luego la b se pone a true, asi nos aseguramos que la música empiece a sonar solo cuando se encuentre en el estado correspondiente.

```
//Si el boolean b es falso, que sirve para detectar cuando nos encontramos en este estado...
//... y la música está activada, iniciamos la música del Mapa1
if (!b && sonido.isMusicaOn()) {
    juliantheme.loop(1, 0.5f);
    b = true;
}
```

Posteriormente llamamos al método de Colisiones animDentro, el cual devuelve true si colisiona con la pared y lo que hace es retroceder

```
if (col.animDentro1(obstaculo, x, y)) {
    dentro = false;
    if (i == 6) {
        x = x - 1;
    }
    if (i == 4) {
        x = x + 1;
    }
    if (i == 2) {
        y = y - 1;
    }
    if (i == 8) {
        y = y + 1;
    }
}
```

al personaje un pixel dependiendo la dirección a la que va, para que así no se quede bloqueado y colisione con las paredes.

Le damos un valor a la variable int i, la cual nos dice la dirección a la que se esta moviendo el personaje dependiendo del numero que contenga y realiza el movimiento oportuno al personaje. Ponemos la varible booleana dentro a true, que significa que el personaje esta dentro de los limites del mapa. Realizamos los movimientos de los enemigos. Actualizamos las coordenadas x e y del personaje y las actualizamos en la clase Colisiones para gestionar las colisiones tanto del personaje como de los enemigos.

Cuando el personaje muere realiza el sonido oportuno, y lleva de nuevo el personaje a las coordenadas iniciales y vuelve a colocar los objetos que habían sido ya recogidos, reinicia el numero de tarjetas recogidas a 0 y le resta una vida. Si el numero de vidas es 0 ¡GAME OVER! y se dirige al estado 7.

```
//Cuando el personaje muere...
if (personaje.muere()) {
    //Si el sonido esta activado suena el sonido
    if (sonido.isSonidoOn()) {
        sonido.getJulianDead().play();
    }
    //Devuelve las coordenadas del personaje a las iniciales
    x = 49;
    y = 288;
    //Reinicia el estado de los objetos del mapa
    obj.setB(true);
    obj.setC(true);
    //Reinicia las tarjetas recogidas
    col.setTarjeta1(0);
    //Resta una vida al personaje
    vidas.setVidas(vidas.getVidas() - 1);
    //Si el número de vidas es 0 entonces para la música y le manda al estado 7, GAME OVER
    if (vidas.getVidas() == 0) {
        juliantheme.stop();
        game.enterState(7, entra, sale);
    }
}
```

También, gestiona cuando el personaje coge el botiquín y le falta alguna vida, entonces aumenta la vida en uno, y si coge las tarjetas aumenta el numero de tarjetas recogidas.

```
//Si el personaje coge el botiquin y no tiene todas las vidas aumenta una vida...
//... y elimina el botiquin del mapa
if (obj.botCol() && vidas.getVidas() < 6) {
    vidas.setVidas(vidas.getVidas() + 1);
    obj.setA(false);
    if (sonido.isSonidoOn()) {
        sonido.getAdrenalina().play();
    }
}

//Si el personaje coge la tarjeta 1 aumenta el número de tarjetas y la elimina del mapa
if (obj.tar1Col()) {
    col.setTarjeta1(col.getTarjeta1() + 1);
    obj.setB(false);
    if (sonido.isSonidoOn()) {
        sonido.getTarjeta().play();
    }
}

//Si el personaje coge la tarjeta 2 aumenta el número de tarjetas y la elimina del mapa
if (obj.tar2Col()) {
    col.setTarjeta1(col.getTarjeta1() + 1);
    obj.setC(false);
    if (sonido.isSonidoOn()) {
        sonido.getTarjeta().play();
    }
}
```

También debe gestionar si se activa o desactiva la música llamando al método de Sonido click.

```
//Cuando el personaje muere...
if (personaje.muere()) {
    //Si el sonido esta activado suena el sonido
    if (sonido.isSonidoOn()) {
        sonido.getJulianDead().play();
    }
    //Devuelve las coordenadas del personaje a las iniciales
    x = 49;
    y = 288;
    //Reinicia el estado de los objetos del mapa
    obj.setB(true);
    obj.setC(true);
    //Reinicia las tarjetas recogidas
    col.setTarjeta1(0);
    //Resta una vida al personaje
    vidas.setVidas(vidas.getVidas() - 1);
    //Si el número de vidas es 0 entonces para la música y le manda al estado 7, GAME OVER
    if (vidas.getVidas() == 0) {
        juliantheme.stop();
        game.enterState(7, entra, sale);
    }
}

//Si el personaje atraviesa el Mapa1 pasa al estado 3
if (col.cambiarMapa1()) {
    juliantheme.stop();
    if (sonido.isSonidoOn()) {
        sonido.getPuerta().play();
    }
    game.enterState(3);
}
```

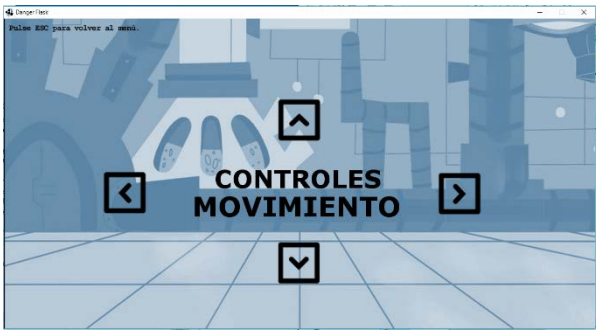
Si detecta con el método de colisiones cambiarMapa, en las clases Mapa1, 2 y 3 pasa al estado siguiente. Y en el Mapa4 pasa al estado del DialogoFinal, el estado 6.

GameOver:

En este estado entramos cuando perdemos todas las vidas. Es el estado 7 y es de tipo BasicGameState. Este estado tiene una imagen la cual renderiza y luego escribe los nombres de los creadores del videojuego a modo créditos. Si se pulsa el botón ESCAPE se vuelve al menú principal del juego, el estado 0.

Controles:

Este estado es al que entramos desde el menú. Es el estado 8 y es de tipo BasicGameState. Esta formado por una imagen que contiene los controles del juego. Renderizamos esta imagen y si pulsamos ESCAPE volvemos al menú.



Opciones:

Este estado es el estado 9, es al que entramos desde el menú y es tipo BasicGameState. Este estado tiene como función activar y desactivar la música y los efectos de sonido. Para ello tenemos una imagen de fondo y otra que rodea la opción seleccionada y cambia según se clickee en activar o desactivar.


```
if (container.getInput().isMouseButtonDown(0)) {
    //Si damos a activar la música el booleano de la música se pone a true
    if (container.getInput().getMouseX() < 930 && container.getInput().getMouseX() > 780
            && container.getInput().getMouseY() < 315 && container.getInput().getMouseY() > 180) {
        sonido.setMusicaOn(true);
        menutheme.loop(1f, 0.5f);
    }
    //Si damos a desactivar la música el booleano de la música se pone a false
    if (container.getInput().getMouseX() < 430 && container.getInput().getMouseX() > 280
            && container.getInput().getMouseY() < 315 && container.getInput().getMouseY() > 180) {
        sonido.setMusicaOn(false);
        menutheme.stop();
    }
    //Si damos a activar el sonido el booleano del sonido se pone a true
    if (container.getInput().getMouseX() < 1075 && container.getInput().getMouseX() > 953
            && container.getInput().getMouseY() < 482 && container.getInput().getMouseY() > 400) {
        sonido.setSonidoOn(true);
    }
    //Si damos a desactivar el sonido el booleano del sonido se pone a false
    if (container.getInput().getMouseX() < 252 && container.getInput().getMouseX() > 130
            && container.getInput().getMouseY() < 485 && container.getInput().getMouseY() > 403) {
        sonido.setSonidoOn(false);
    }
}
```

Cuando una opción es clickada cambia en valor booleano de MusicaOn y SonidoOn de la clase Sonido. Esto lo conseguimos en el update con el código indicado.

Para más información acerca del código, en el propio código del juego hay varios comentarios explicando las funcionalidades de este.

---

11. Diagrama de clases. UML