|  |
| --- |
| Palacios Limón, Guillermo Barchín Rubio, Álvaro Rodriguez Jañez, Eloy Matellanes Ramón, Miguel Pérez López, Xabier 2018-2019 |

|  |
| --- |
| Tecnología de Videojuegos |
| BEYOND SOFTWARE |
| Documento Técnico |



# Introducción

A continuación se explicarán las distintas clases que se han creado a lo largo del videojuego, así como las funciones cuyo funcionamiento no sea evidente.

Para facilitar la explicación y con el objetivo de mantener un orden, se catalogarán las clases en varios bloques:

* ***Historia***
* ***Introducción***
* ***Niveles***
* ***Jugador***
* ***Jefes finales***
* ***Enemigos***
* ***Objetos***

# Clases para la Historia y para la Introducción

## Historia1. java, Historia2. java, historia3. java, Historia4. java

### Función

El juego está formado por varios niveles entre los cuales se mostrará en pantalla una imagen en la que aparecerá una frase relacionada con el juego y, en alguna de ellas, algún consejo para el jugador

### Atributos

El único atributo que será necesario será el atributo *image*, que contendrá la imagen que se debe mostrar por pantalla entre cada nivel.

Será inicializada dentro de la función ***public void init();***

### Métodos

Los métodos pertenecientes a estas clases son los que se enumeran a continuación:

* ***public int getID():*** devuelve el número que identifica a cada estado para poder realizar la transición de éstos.
* ***public void init():*** inicializa los atributos que posea la clase. En este caso solo se inicializa el atributo *image*.
* ***public void render():*** esta función está destinada a imprimir texto e imágenes por pantalla. Dentro de esta función se llamará a las funciones *g.drawImage();* y *g.drawString();*
* ***public void update():*** puesto que la única funcionalidad de estas clases es mostrar una imagen por pantalla, cuando el jugador presione la tecla *ENTER*, se pasará al siguiente estado, que será el siguiente nivel según corresponda. Esta función será implementada dentro de este método.

## Historia5. java, Historia6. java, historia7. java, Historia8. java, Historia9. java

### Función

Una vez derrotemos al jefe final, estas clases, funcionando a modo de estados, darán una última explicación de la historia al jugador. Los atributos y los métodos serán los mismos que en las clases anteriores.

### Atributos

*Véase 2.1.2 Atributos*

### Métodos

*Véase 2.1.3 Métodos*

## Logo. java y Logo\_juego. java

### Función

La función de estas clases es mostrar por pantalla el logotipo del equipo de desarrollo (Logo.java) y el logo del videojuego (Logo\_juego.java), antes de mostrar el menú principal del videojuego.  
Los atributos y los métodos serán los mismos que en los casos anteriores con la salvedad de que se añade, dentro de *Logo\_juego.java* el atributo *historia*, de tipo *Music*

### Atributos

Además de los atributos ya mencionados (*Véase 2.1.2 Atributos*) la clase *Logo\_juego.java* incluye el atributo *private Music historia*.

### Métodos

Todos los atributos mantienen su función (*Véase 2.1.3 Métodos*) salvo el método ***public void init()*** que en que este caso también inicializa el atributo *historia* y es cuando pulsamos *ENTER* dentro de la función ***public void update();*** cuando comienza a reproducirse con la instrucción *historia.loop();*

## Menu. java

### Función

Esta clase mostrará por pantalla un fondo con 3 opciones que podrá elegir el usuario para comenzar una nueva partida, cargar partida o salir del juego.  
Al pulsar nueva partida, se mostrará la introducción del videojuego.

### Atributos

* ***private int selected:*** opción elegida por el usuario.
* ***private String[] options = new String[]{“ ”,” ”,” ”}:*** variable tipo string que almacena las opciones que puede pulsar el jugador. En este caso aparecerán vacías porque las opciones están impresas sobre la imagen de fondo.
* ***private StateBasedGame game***
* ***private Image image***
* ***private Sound sound:*** almacena el sonido que debe reproducirse cuando el jugador se desplace entre las distintas opciones.

### Métodos

* ***public int getID();***
* ***public void init():*** dentro se inicializan los atributos *image* y *sound*.
* ***public void render():*** imprime la imagen de fondo y dibuja el recuadro que se superpone a la opción que esté seleccionando el jugador en cada momento.
* ***public void update():*** dentro contiene una máquina de estados implementada con un switch-case. Con el primer estado se iniciará una nueva partida, con el segundo estado se cargará la partida en función del nivel en el que nos hayamos quedado y con el último estado el jugador saldría del juego.
* ***public void keyReleased():*** con esta función, pulsando las flechas arriba o abajo podemos desplazarnos entre las tres opciones disponibles.

## Intro1. java, Intro2. java, Intro3. java, Intro4. java, Intro5. java, Intro6. java, Intro7. java

### Función

Dan al jugador una breve introducción de la historia para ponerlo en contexto. Puesto que puede ser que no sea la primera vez que el jugador comienza el juego, se ha decidido dar la opción de que pulsando el botón *ESPACIO*, el jugador podrá saltar toda la introducción y saltar

### Atributos

Se añade a los atributos anteriores (*Véase 2.1.2 Atributos*) el atributo *nivel1*, de tipo *Music*.

### Métodos

Dentro del método ***public void update*** al pulsar *ESPACIO* (o *ENTER también en la clase Intro7.java)* se comenzará a reproducir la música del primer nivel.

# Clases para los Niveles

Todos los niveles tendrán la misma estructura y la misma lógica de programación por lo que, para no alargar el documento más de los necesario, se explicará tan solo el primer nivel.

## Atributos

* private Input entrada;
* private Jugador player;
* private SpriteSheet mapa;
* private ArrayList<Sala> salas;
* private int salaActual;
* private ControladorProyectiles proyectiles;
* private String[] options = new String[]{“Volver al juego”, “Controles”, “Menú Principal”};
* private int selected;
* private Boolean paused;
* private Image fondoPausa;
* private Image controles;
* private Music nivel1;
* private Sound select;

## Métodos

* ***public int getID()***
* ***public void init():*** en esta función se llevará a cabo la inicialización del nivel. En primer lugar, puesto que en la fase de diseño todas las habitaciones de un mismo nivel se dibujaron en una misma imagen, debemos crear a partir de dicha imagen las salas individuales por las que se moverá el jugador.   
  Para cada sala será necesario crear:
  + *Array de colisiones (ArrayList<Wall> walls1 = new ArrayList<>();)*
  + Array de puertas (*ArrayList<Puerta> puertas1 = new ArrayList<>();*)
  + Array de enemigos (*ArrayList<Enemigo> enemigos1 = new ArrayList<>();)*

Una vez creados los arrays, crearemos cada sala usando la siguiente expresión, cambiando los parámetros en función de cada sala:

* *Sala sala1 = new Sala(mapa.getSubImage(0, 1), walls1, puertas1, enemigos1, null, player, proyectiles);*

Tras crear las salas las añadiremos al array de salas usando la siguiente función:

* *salas.add(sala1);*

Para crear el resto de salas dentro del nivel seguiríamos el mismo procedimiento

* ***public void render():*** dentro de esta función se evaluará si el juego ha sido pausado o no. Cuando esté pausado, en pantalla se mostrará una imagen de fondo con un menú de pausa desde el que podremos volver al juego, ver los controles o volver al menú principal.  
  Mientras que el juego no esté en pausa, por pantalla se mostrará la sala en la que nos encontremos.
* ***public void update():*** dentro de esta función se llevarán a cabo varias tareas:
  + *Si el botón ESC es pulsado el juego entra en pausa, y si es pulsado de nuevo, se reanuda el juego.*
  + Mientras que el juego este pausado en función de la opción pulsada en el menú, podremos volver al juego, mostrar los controles o salir del juego pulsando *ENTER* estando sobre la opción deseada.
  + ***public void keyReleased():*** con esta función de Slick2D, estando en el menú de pause, pulsando sobre S y W podemos movernos entre las distintas opciones.

# Clase jugador

Esta clase, como su propio nombre indica será la que implemente al personaje que el jugador va a manejar.

## Atributos

* *private SpriteAnimado personaje*
* *private Rectangle hitbox*
* *private boolean up, down, r, l, stop, hit*
* private ControladorProyetiles proyectiles
* *private int vida, vidaTotal, cooldown, varitaActual, tiempoInvencibilidad, flickerTime*
* *private Image corazonVacio, corazonLleno, corazonMedio*
* *private Inventario inventario*
* *private Sound select*
* *private Sound pocion*

## Métodos

* ***public Jugador():*** esta función servirá como inicialización del personaje.
  + En primer lugar añadimos el Spritesheet del personaje y lo dividimos en filas que asociaremos a una dirección. De esta forma, con un bucle for que recorra la fila y el controlador de animaciones, podemos lograr que a medida que nos desplacemos el Sprite del personaje vaya cambiando y dé la sensación de movimiento.
  + A continuación se le ajusta una hitbox para saber cuando se choca o cuando es golpeado por algo.
  + En las siguientes líneas de código se inicializan todos los atributos anteriormente mencionados.
* ***public void draw():***
* ***public void drawCorazones():*** con este método se mostrarán por pantalla los corazones equivalentes a la vida restante del jugador.
* ***public void update():*** para obtener una velocidad constante y que sea independiente del equipo en el que se ejecuta el juego, se pasa un parámetro delta que almacena el tiempo que ha pasado entre cada llamada a la función.
* ***private void updateTeclado():*** en esta función se llevarán a cabo varias tareas:
  + *Cambio de varitas: pulsando CONTROL IZQUIERDO o ESPACIO cambiamos la varita seleccionada.*
  + *Uso de pociones:* pulsando los botones *E* y *R* usaremos una poción u otra.
  + *Movimiento del personaje:* pulsando los botones *W, A, S* y *D*, el personaje se desplazará en una dirección u otra. Además, si pulsamos el botón SHIFT IZQUIERDO, teniendo las botas, la velocidad del personaje incrementa.
* ***private void atacar():*** para evitar que el jugador pueda atacar de forma continua e indiscriminada, hemos optado por introducir un retardo usando el atributo *cooldown*, que variará en función de la varita que se esté usando. Independientemente de la dirección en la que nos estemos desplazando, pulsando las *FLECHAS* del teclado, dispararemos en la dirección deseada.
* ***public int getCooldownActual():*** en función de la varita elegida, el *cooldown* varía.
* ***public void crearProyectil():*** esta función se encarga de crear el proyectil con el sprite y el daño adecuados en función de la varita que se esté usando.
* ***public void alColisionar():*** evalúa con qué ha colisionado el personaje y actúa en consecuencia:
  + *Colisión con puerta:* el jugador se desplaza a la sala y posición indicadas.
  + *Colisión con proyectil:* el jugador se desplaza en la dirección contraria a la del proyectil para dar sensación de impacto.
  + *Colisión con objeto:* el objeto se añade al inventario.

# Clase Enemigo

## Función

La clase enemigo contendrá los métodos y atributos que heredarán las clases creadas para instanciar los distintos enemigos que intervienen en el juego.  
A continuación se listarán los atributos y métodos de dicha clase y para clase hija se comentarán los cambios realizados.

## Atributos

* *private SpriteAnimado sprite;*
* *private Rectangle hitbox;*
* *private Circle visionRange;*
* *private boolean up, down, r, l, collision, hit;*
* *private int vida, ataque cooldown, distanciaVision, movX, movY, velocidad, flickerTime, offsetX, offsetY, offsetHeight, offsetWidth;*
* *private Punto playerPosition;*

## Métodos

* ***Constructor 1 - public Enemigo(String filename, int ancho, int alto, int x, int y, int vida, int ataque, int velocidad):*** es el constructor del objeto. Toma los parámetros para crear el tipo de enemigo requerido y crea los sprites y animaciones correspondientes del mismo modo que hacíamos con la clase *Jugador.java*.   
  Se inicializan el resto de atributos, al igual que antes, y antes de salir de la función, se establece la regla de movimiento aleatorio que tendrán ciertos enemigos.
* ***Constructor 2 - public Enemigo(String filename, int ancho, int alto, int x, int y, int vida, int ataque, int velocidad, int offsetX, int offsetY, int offsetWidth, int offsetHeight):*** la función es igual que la anterior, con la salvedad de que añade la opción de variar el tamaño de la hitbox. Será empleado en enemigos como el Slime, el Murciélago o el Demonio.
* ***Constructor 3 - public Enemigo(String filename, int ancho, int alto, int x, int y, int distanciaVision, int vida, int ataque, int velocidad):*** mantiene las mismas funciones que el primer caso pero añade el atributo *distanciaVision* para aquellos enemigos que tengan la opción de disparar proyectiles.
* ***El resto de métodos son compartidos con la clase Jugador.java por lo que no se volverán a explicar para no alargar el documento más de lo necesario.***

## Esqueleto. java

### Función

El esqueleto se moverá de manera aleatoria, modificando su dirección con cada colisión y además comenzará a lanzar proyectiles cuando el jugador entre en su rango de visión.

### Modificaciones

* ***Constructor:*** se llama al *Constructor 3* de la clase padre y además inicializa los proyectiles y los sprites para la animación.
* ***Atacar:*** cuando el jugador entra en el rango de visión, el esqueleto toma como punto de referencia la posición del jugador y comienza a lanzar proyectiles con una cadencia que vendrá limitada por el *cooldown* elegido.

## Murcielago. java

### Función

El murciélago se desplazará de manera aleatoria a mayor velocidad por toda la sala, cambiando su dirección cuando colisione.

### Modificaciones

* ***Constructor:*** se llama al *Constructor 2* de la clase padre
* ***Avance:*** se analiza desde que dirección ha llegado la colisión y se le asigna una nueva dirección aleatoria. En función de esa dirección se mostrará la animación correspondiente.

## Slime

### Función

Se desplaza de manera aleatoria hasta colisionar, donde cambia de dirección.

### Modificaciones

No se realiza ninguna modificación, hereda los métodos de la clase padre usando el *Constructor 2*.